Notes de cours *M1 E3A*

Linux – Commandes et appels systèmes

Hugues Mounier[†]

[†] Laboratoire des Signaux et Systèmes
Université Paris Sud 11, CNRS, Supélec
3, rue Joliot Curie, 91192 Gif sur Yvette, France
e-mail : Hugues.Mounier@l2s.centralesupelec.fr
Hugues.Mounier@u-psud.fr

2017/2018– **Document en cours d'élaboration**

ver. 1.0N

LABORATOIRE DES SIGNAUX ET SYSTÈMES

Table des matières

Partie A

Notions de base

Partie B

Compléments de C, commandes & shell

I – Rappels et compléments sur le langage C

1 Complements sur les entrées/sorties

La gestion des entrées/sorties est un élément déterminant dans la maitrise de tous les langages et spécialement dans celle du langage C. Nous examinerons dans ce chapitre,

les possibilités de gestion des tampons d'entrées/sorties, les accès directs aux fichiers, et certaines fonctions particulières telles que les fonctions de formatage interne des données.

1.1 Gestion des tampons d'entrées/sorties

Le transfert d'information entre la mémoire de l'ordinateur et le disque peut s'effectuer soit directement, soit en passant par un tampon servant d'interface entre le disque et l'adresse mémoire où la donnée est à saisir ou à écrire. Les différentes fonctions d'entrées/sorties du langage C utilisent normalement des tampons qui sont alloués automatiquement à l'ouverture d'un fichier par le système d'exploitation.

Quand le transfert s'effectue sans tampon vers un fichier de sortie, l'information apparait dans le fichier dès qu'elle est écrite. Quand le transfert s'effectue avec tampon, deux cas peuvent se produire : si le tampon est créé en mode bloc, l'information est stockée dans le tampon jusqu'à saturation, et le tampon est alors transféré. Si le tampon est créé en mode ligne, il est rempli jusqu'à la rencontre du caractère '\n' (nouvelle ligne) et transféré alors. Par défaut, la sortie standard `stdout` et l'entrée standard `stdin` associées au terminal de connexion utilisent des tampons en mode ligne. En revanche, la sortie d'erreur `stderr` est sans tampon.

Le tampon de sortie associé à un fichier est vidé automatiquement par le système lorsqu'il est plein. L'utilisateur peut néanmoins en demander la purge à tout instant (même si le tampon n'est pas plein) par l'appel à la fonction `fflush()`.

Description

```
fflush(FILE *fp); /* fp = pointeur de fichier */
                  /* renvoie 0 si OK ou EOF en cas d'erreur */
```

La fonction provoque le transfert physique immédiat dans le fichier associé au pointeur fp, des données écrites dans le tampon. Cette fonction est en général utilisée pour s'assurer que toutes les informations à écrire sur un fichier ont été physiquement transférées. Il se peut en effet qu'un test conduise à arrêter l'exécution d'un programme, dans ce cas il peut être nécessaire de faire appel à `fflush()` pour transférer physiquement toutes les informations écrites vers le fichier. Appliquée au tampon d'un fichier ouvert en entrée (par exemple `stdin`) l'action de `fflush()` est indéfinie.

Exemple de code I.1

```
int flag;

if (flag == 0)
    fflush(stdout);  /* vide le tampon associe a stdout */
```

Toutes les informations qui avaient été écrites sur le fichier de sortie standard sont physiquement affichées sur l'écran.

1.2 Fichiers ouverts à la fois en lecture et en écriture

L'ouverture d'un fichier est réalisée par la fonction `fopen()` qui renvoie un pointeur valide ou le pointeur `NULL` en cas d'échec.

Description

```
FILE *fp;
fp = fopen(const char *nom, const char *type);
```

La fonction `fopen()` ouvre le fichier dont le nom est spécifié par le pointeur nom, et lui associe en cas de succès le pointeur `fp`. L'ouverture est réalisée selon le type précisé :

- `r` lecture seule permise.
- `w` écriture seule possible en début de fichier.
- `a` écriture autorisée à partir de la fin du fichier.
- `r+` lecture et écriture permises.
- `w+` écriture et lecture permises, le contenu précédent du fichier étant détruit.
- `a+` écriture et lecture permises, l'écriture étant autorisée à partir de la fin du fichier.

Les trois derniers types sont destinés à la mise à jour d'un fichier en utilisant un seul pointeur de fichier, au lieu de manipuler deux pointeurs différents, l'un contrôlant le fichier en lecture, et l'autre en écriture.

Il faut cependant être attentif au fait que lorsqu'un fichier est ouvert avec un seul pointeur, le même tampon est utilisé pour assurer l'écriture et la lecture. Il en résulte généralement des effets inattendus et indésirables. Pour éviter de tels effets, il faut obligatoirement que chaque écriture soit suivie d'un appel à `fflush()` ou à un appel à `fseek()` (voir plus loin). De même, toute lecture doit être suivie d'un appel à `fseek()` avant toute écriture. La séquence d'instructions suivante illustre cette difficulté :

Exemple de code I.2

```
#include <stdio.h>
main(void)
{
  FILE *fr;
  char s[80];
  fr = fopen("fich","r+"); /* ouvert en lecture/ecriture */
  fgets(s,80,fp);
  fseek(fp,0L,0);          /* pointeur au debut du fichier */
  if (fputs("*******",fp) == (char)EOF)
      printf("Erreur d'ecriture\n");
}
```

Cette séquence exécutera la lecture et l'écriture attendues. En revanche, si l'appel à `fseek()` est omis, l'écriture sera erronée car elle contiendra la chaîne demandée `"******"` précédée par l'information lue par `fgets()`. De plus, l'écriture ne sera pas effectuée au

début du fichier mais après un nombre d'octets égal à la taille du tampon d'entrée/sortie utilisé par le système d'exploitation Linux. En effet, lors de l'appel `fgets()`, Linux lit le fichier `"fich"` jusqu'à saturation du tampon (ou la rencontre de la fin de fichier) et déplace le pointeur à l'adresse correspondante dans le fichier. Comme il n'y a qu'un seul pointeur de fichier, l'écriture suivante est effectuée à cette adresse.

1.3 Accès direct

Fonction `fseek()`. Il est possible de se placer dans un fichier sur un octet donné. Cette opération s'effectue par la fonction `fseek()`.

```
int   origin;
FILE *stream;
long  offset, nptr;

int fseek(FILE *fp, long offset, int origin);
     /* fp = pointeur de fichier */
     /* offset = rang de l'octet à atteindre               */
     /* origin = origine pour calculer l'octet à atteindre */
     /* renvoie 0 normalement ou -1 en cas d'erreur       */
```

La fonction `fseek()` positionne le pointeur de fichier `fp` à l'octet de rang `nptr` (par rapport au début du fichier). L'opération est réalisée de la manière suivante :

Lorsque `origin = SEEK_SET`, alors `nptr = offset`

Lorsque `origin = SEEK_CUR`, alors `nptr = pointeur courant + offset`

Lorsque `origin = SEEK_END`, alors `nptr = fin de fichier + offset`

Les mnémoniques `SEEK_SET`, `SEEK_CUR`, et `SEEK_END` sont définis dans le fichier `stdio.h` et valent respectivement 0, 1, et 2.

Nota :

- L'offset doit être un entier long. Si la valeur est passée sous forme de constante numérique, il faut prendre garde de la convertir en entier long par un cast. Exemple : `fseek(fp, (long)512, 0);`
  Cette instruction positionne le fichier sur le $512^{\text{ième}}$ octet depuis l'origine du fichier, c'est à dire que le premier octet concerné par la prochaine entrée/sortie sera le $513^{\text{ième}}$.
- Lorsque l'origine est prise en fin de fichier (`origin` = 2), l'offset doit normalement être négatif car sa valeur est additionnée à la position du dernier octet du fichier.

Fonction `ftell()`. Il est possible de connaître la position courante du pointeur d'un fichier à l'aide de la fonction `ftell()` qui donne le rang de l'objet pointé présentement par le pointeur de fichier. Cette position est donnée par rapport à l'origine du fichier.

```
long int p;
p = ftell(FILE *fp); /* fp = pointeur de fichier    */
                     /* renvoie -1 en cas d'erreur */
```

La variable p doit être impérativement du type `long`. Lorsque le fichier vient d'être ouvert, `ftell()` renvoie 0, et lorsque `ftell()` suit un appel à `fseek()`, cette fonction renvoie la valeur spécifiée par le paramètre offset.

Exemple de code I.3

```
long p;
fseek(fp,(long)512,0):
p = ftell(fp);
  /* p a la valeur 512 */}
```

Fonction `rewind()`. Enfin, le pointeur d'un fichier peut être replacé à l'origine à tout moment en utilisant la fonction `rewind()` :
`void rewind(FILE *fp); /* fp = pointeur de fichier */`
`rewind(fp)` est équivalent à l'appel `fseek(fp, 0L, SEEK_SET)`.

1.4 Fonctions d'entrées/sorties particulières

Il est possible de lire des données à partir d'une chaîne de caractères, ou d'en écrire dans une chaîne.

Fonctions `sscanf()`, `sprintf()`.
`int sscanf(char *s,char *format,int *arg,...);`
Cette fonction est similaire à `scanf()` ou `fscanf()`, excepté qu'elle trouve la source de ces données dans la chaîne `s`. La fonction renvoie le nombre de variables lues.
`int sprintf(char *s,char *format,int arg,...);`
Cette fonction écrit à l'adresse pointée par `s` la chaîne obtenue par formatage des quantités passées en arguments. La fonction renvoie le nombre de variables converties.

Fonction `perror()`. Sous Linux, lorsqu'un processus provoque une erreur, cette erreur est repérée par un numéro. Il existe une fonction qui permet l'impression du message d'erreur associé au numéro de l'erreur. Cette fonction est `perror()`.
`perror(char *s); /* Imprime le message d'erreur et la chaine s */`
`perror()` imprime un message d'erreur sur le fichier standard d'erreur (`stderr`) décrivant la dernière erreur rencontrée lors d'un appel système. La chaîne `"s"` est un message d'erreur additionnel, éventuellement fourni par l'utilisateur. Cette chaîne permet notamment d'identifier la position de l'appel à `perror()` dans le programme (Il y a souvent plusieurs appels).

Pour mémoire, rappelons qu'il existent deux fonctions qui contrôlent la rencontre d'une fin de fichier ou celle d'une erreur. Ce sont respectivement `feof()` et `ferror()`.

2 Normalisation C ANSI

Depuis sa création en 1973, le langage C n'a pas connu de modifications considérables. La plupart des constructeurs qui ont implémenté ce langage sur leur machine se sont fondés sur la description du langage donnée par la première édition du livre de référence de B. Kernighan et D. Ritchie.

Cependant avec le développement de l'usage du C, et le développement d'Unix puis de Linux, certains ajouts et extensions sont apparus. Ces modifications, pour la plupart, n'ont pas apporté de changements importants et ont assuré une compatibilité ascendante, c'est à dire que les programmes "ancien style" peuvent être compilés sans erreur avec les nouveaux compilateurs. Le succès du langage C a rendu nécessaire une standardisation du langage original et de ses extensions. Cette standardisation a fait l'objet d'une norme ANSI rendue publique en 1989.

2.1 Modifications introduites par la norme ANSI

Nous résumons ici les différences existants entre la première définition du langage C et la norme ANSI C.

2.1.1 Le préprocesseur

Avant la normalisation, la fin de ligne terminait une pseudo-instruction commençant par `#`. Avec la norme ANSI, si la ligne est terminée par un \ elle se prolonge sur la ligne suivante.

Exemple I.1.

```
#define TAILLE \
 1000
```

Ceci est équivalent à :

```
#define TAILLE 1000
```

D'autre part, il y a deux nouveaux opérateurs désignés par `#` et `##` qui permettent l'équivalent de l'usage des quotes en shell (opérateur `#`) ou la concaténation de chaînes (opérateur `##`).

2.1.2 Modifications de syntaxe

De multiples détails de syntaxe ont été modifiés ou aménagés. Ils sont pour la plupart transparents pour l'utilisateur. Les principales modifications sont les suivantes :

– De nouveaux mots clés ont été introduits, à savoir `void`, `const`, `volatile`, `signed` et `enum`.

– La norme a élargi le nombre de suffixes pour forcer une constante numérique à être d'un certain type. On trouve :
  – Pour les entiers `U` (non signé) et `L` (long)
  – Pour les flottants `F` (simple précision), `D` (double précision)
– La liste des caractères non imprimables qui peuvent être décrits par une suite de un ou plusieurs caractères a été étendue. Figurent en plus, dans cette liste :

  `\a` BEL, sonnerie du terminal (`ctrl G`)

  `v` VT, tabulation verticale

  `xhh hh`, nombre hexadécimal
– Un caractère peut être désormais signé ou non signé, selon la déclaration. Il devient l'équivalent d'un entier sur 8 bits. Le préfixe `signed` est sans signification devant un autre type que `char`. Exemple :
```
signed char c1; /* caractère signé (-128 < c1 < +127) */
unsigned char c2; /* caractère non signé (0 < c2 < +255) */
```
– Le type `long double` a été introduit. Il définit normalement une précision supérieure à la précision du type `double`. Il est rarement implémenté actuellement.
– Le type `void *` a remplacé `char *` pour désigner un pointeur sur un type non défini. Par exemple, les fonctions d'allocation mémoire renvoient désormais un pointeur de type `void *` au lieu de `char *`.
– Les structures peuvent être désormais assignées, passées en argument à une fonction et retournées par une fonction.
– La nouvelle norme introduit la notion de déclaration de prototype de fonction, avec une déclaration des types de variables à l'intérieur des parenthèses. Cette nouvelle disposition permet la vérification des types d'arguments lors de l'appel d'une fonction. Les déclarations "ancien style" sont toujours autorisées (avec quelques restrictions).

Exemple de code I.4

```
main(void)
{
  char *fonct(int, char, float);    /* nouveau style */
  /* déclare une fonction avec 3 paramètres de types int, char */
  /* et float. La fonction renvoie un pointeur de type char   */
  double add();                     /* ancien style autorisé */
  ...
}
```

– Les tableaux, structures et unions peuvent être initialisés même lorsque ce sont des variables automatiques (auparavant, ils devaient être de classe `static` pour pouvoir l'être).

– Les tableaux de caractères déclarés explicitement avec une taille N peuvent être initialisés par une chaîne contenant N caractères (le `\0` n'est pas ajouté dans ce cas). auparavant, la chaîne ne devait pas dépasser N-1 caractères. Par exemple :

```
char ch[7] = "exemple";
```

n'entraine pas d'erreur de compilation.

2.2 Le fichier <sys/limits.h> et la norme POSIX

En dépit de toute normalisation, l'architecture d'un ordinateur ne peut être totalement transparente. Pour assurer la portabilité des codes, le fichier `<sys/limits.h>` définit des tables d'équivalence entre un mnémonique et une valeur limite liée à la machine. On trouve notamment dans ce fichier les mnémoniques qui suivent (les valeurs numériques indiquées sont les valeurs *minimales* garanties par la norme) :

Mnémonique	Valeur	Signification
`#define CHAR_BIT`	8	bits dans un `char`
`#define CHAR_MAX`	`UCHAR_MAX` ou `SCHAR_MAX`	maximum de `char`
`#define SCHAR_MAX`	127	maximum de `signed char`
`#define UCHAR_MAX`	255	max de `unsigned char`
`#define CHAR_MIN`	0 ou `SCHAR_MIN`	minimum de `char`
`#define SCHAR_MIN`	-127	minimum de `signed char`
`#define INT_MAX`	+32767	maximum de `int`
`#define INT_MIN`	-32768	minimum de `int`
`#define LONG_MAX`	+2147483647	maximum de `long`
`#define LONG_MIN`	-2147483648	minimum de `long`
`#define SHRT_MAX`	+32767	maximum de `short`
`#define SHRT_MIN`	-32768	minimum de `short`
`#define UINT_MAX`	65535	maximum de `unsigned int`
`#define ULONG_MAX`	4294967295	max de `unsigned long`
`#define USHRT_MAX`	65535	max de `unsigned short`

D'autre part, un certain nombre de paramètres implicitement ou explicitement manipulés en Langage C sont dépendants du système d'exploitation de la machine. Il s'agit par exemple du nombre de fichiers ouverts simultanément par un programme, ou encore de la longueur maximale autorisée pour un nom de fichier.

Sous Linux, la norme POSIX a définit des valeurs minimales pour tous ces types de paramètres (le constructeur de la machine peut bien entendu définir des valeurs plus grandes que les minima POSIX). Il est impossible ici de faire la liste exhaustive de tous ces paramètres, d'autant plus qu'un grand nombre de ces paramètres sont relatifs aux appels système Linux. Il est utile toutefois de connaitre la valeur standard des paramètres suivants :

Mnémonique	Valeur	Signification
`_POSIX_ARG_MAX`	4096	Nombre maximal de caractères pour les arguments d'un appel système.
`_POSIX_NAME_MAX`	256	Nombre de caractères maximal dans un répertoire pour un nom de fichier.
`_POSIX_OPEN_MAX`	16	Nombre maximal de fichiers ouverts simultanément.
`_POSIX_PATH_MAX`	255	Nombre de caractères maximal d'un nom de fichier.

3 La bibliothèque C

La bibliothèque du langage C est très riche, et contribue à donner au langage sa puissance et sa flexibilité. Nous avons regroupé ici les principales fonctions par type de traitement.

3.1 Fonctions de traitement de caractères

Les fonctions dont le nom commence par '`is`' testent si un caractère est d'un certain type. Elles renvoient 0 si c'est faux et une valeur différente de 0 si c'est vrai. Les fonctions dont le nom commence par '`to`' convertissent une valeur en un caractère. Elles renvoient la valeur ASCII de ce caractère. Pour utiliser ces fonctions, il faut déclarer :
`#include <ctype.h>`

Fonctions de test `isXXX()`

- `int isalnum(const char c);`
  Cette fonction teste si c est une lettre ou un chiffre.
- `int isalpha(const char c);`
  Teste si c est une lettre (minuscule ou majuscule).
- `int isascii(const char c);`
  Teste si le caractère c est un caractère ASCII (valeur de c comprise entre 0 et 127).
- `int iscntrl(const char c);`
  Teste si c est un caractère de contrôle, c'est à dire si c a une valeur comprise entre 0 et 31, ou 127.
- `int isdigit(const char c);`
  Teste si c est une valeur numérique.
- `int islower(const char c);`
  Teste si c une minuscule.

- `int isprint(const char c);`
  Teste si c est imprimable (compris entre 32 et 126).
- `int ispunct(const char c);`
  Teste si c est un signe de ponctuation.
- `int isspace(const char c);`
  Teste si c est un espace, un signe de tabulation, un retour chariot, un saut de page ou un saut de ligne.
- `int isupper(const char c);`
  Teste si c une majuscule.
- `int isxdigit(const char c);`
  Teste si c est une valeur numérique héxadécimale, c'est à dire appartenant à 0-9, a-f,ou A-F.

Fonctions de conversion `toXXX()`

- `char toascii(int i);`
  Convertit la valeur entière i en une valeur ASCII en ne conservant que les 7 bits de poids faible..Si i est une valeur ASCII il n'y a pas de changement.
- `char tolower(int i);`
  Convertit une majuscule en minuscule. Si i n'est pas une lettre majuscule, la fonction laisse le caractère inchangé.
- `char toupper(int i);`
  Convertit une majuscule en minuscule. Si i n'est pas une lettre minuscule, la fonction laisse le caractère inchangé.

3.2 Traitement des chaînes de caractères

Ces fonctions comparent, concatènent, copient des chaînes de caractères et comptent le nombre de caractères contenus ou encore convertissent en un format donné. En plus du fichier habituel `<stdio.h>`, il faut déclarer :

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

Fonctions de conversion de chaîne

- `double atof(const char *s);`
  Convertit la chaîne `s` et renvoie sa valeur comme un flottant de type `double`.
- `int atoi(const char *s);`
  Convertit la chaîne `s` et renvoie sa valeur comme un entier de type `int`.
- `int atol(const char *s);`
  Convertit la chaîne `s` et renvoie sa valeur dans un entier de type `long`.

Fonctions de comparaison, copie

Les fonctions qui suivent nécessitent la déclaration :
`#include <string.h>`

- `char *strcat(char *dest, char *source);`
  Cette fonction concatène les deux chaînes en ajoutant le contenu pointé par `source` à la suite de celui pointé par `dest` (dont elle retire '\0') et termine par un '\0'. La fonction renvoie le pointeur `dest`.
- `int strcmp(char *s1, char *s2);`
  La fonction compare les deux chaînes pointées par s1 et s2. Elle renvoie 0 si les deux chaînes sont identiques. Dans le cas contraire, elle renvoie la valeur de la première différence rencontrée (soustraction du caractère de s1 et du caractère correspondant de s2).
- `char *strcpy(char *s1, char *s2);`
  La fonction copie la chaîne s2 dans la chaîne s1 (d'une façon générale le résultat est toujours mis dans le premier paramètre). Les dépassements de capacité de s1 ne sont pas testés.
- `int strlen(char *s);`
  La fonction renvoie le nombre de caractère contenus dans la chaîne `s`.
- `char *strncat(char *s1, char *s2, int n);`
  La fonction ajoute à la chaîne s1 les n premiers caractères de la chaîne pointée par s2. Elle renvoie le pointeur s1.
- `int strncmp(char *s1, char *s2, int n);`
  La fonction compare les deux chaînes définies par les pointeurs s1 et s2 sur les n premiers caractères. Elle renvoie 0 si les deux portions sont identiques. Dans le cas contraire, elle renvoie la différence entre les valeurs ASCII du premier caractère différent (soustraction du caractère de s1 et du caractère correspondant de s2).
- `char *strncpy(char *s1, char *s2, int n);`
  La fonction copie les n premiers caractères de la chaîne s2 dans la chaîne s1, sans rajouter de '\0'. Si s2 est plus petite que le nombre n demandé, la copie s'arrête à la rencontre de '\0' (qui est inclus). Le résultat est retourné dans s1.
- `char *strchr(char *s1, char c);`
  La fonction renvoie un pointeur sur la première occurrence du caractère c dans la chaîne pointée par s1. En cas d'échec (caractère c non trouvé), elle renvoie le pointeur `NULL`.
- `char *strstr(char *s1, char *s2);`
  La fonction renvoie un pointeur sur la première occurrence de la chaîne pointée par s2 dans la chaîne pointée par s1. En cas d'échec, elle renvoie le pointeur `NULL`.
- `char *memcpy(char *s1, char *s2, n);`
  La fonction copie les n premiers caractères de la chaîne s2 dans la chaîne s1 et - à la différence de `strncpy()` rajoute un '\0'. Si s2 est plus petite que le nombre n

demandé, la copie s'arrête à la rencontre de '\0' (dans ce cas, le fonctionnement de `memcpy()` est identique à celui de `strncpy()`). Le résultat est retourné dans s1.

Il existe quelques autres fonctions de traitement de chaînes de caractères. Elles sont, pour la plupart redondantes avec celles décrites ici. Par exemple, la fonction `memcmp(char *s1, char *s2, int n)` est absolument identique à la fonction `strncmp(char *s1, char *s2, int n)`.

3.3 Les fonctions d'allocation dynamique de mémoire

Les fonctions d'allocation dynamique de mémoire sont très importantes dans le Langage C, car en association avec les pointeurs, elles donnent un grande souplesse à la programmation.

En effet, sans ces fonctions les pointeurs perdraient beaucoup d'intérêt, puisqu'ils ne réservent pas d'espace mémoire.

Il faut noter que toutes ces fonctions renvoient depuis la normalisation ANSI, un pointeur sur un objet non typé (`void`) alors qu'auparavant, elles retournaient l'adresse d'un octet (pointeur de type `char *`). Cela implique lors de leur emploi, que le programmation fasse précéder l'appel d'un fonction par une conversion de type vers le type désiré (voir l'exemple à propos de `malloc()`). En plus du fichier habituel `<stdio.h>`, il faut déclarer :

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

– `void *malloc(unsigned int taille);`
  Cette fonction alloue le nombre d'octets contigus de mémoire spécifié par `taille`. La fonction renvoie un pointeur sur l'adresse de départ de la zone mémoire allouée qui n'est pas initialisée. Elle renvoie `NULL` en cas d'échec.

Exemple de code I.5

```
#include <stdio.h>
#include <stdlib.h>
main(void)
{
  int *pt;
  pt = (int)malloc(1000 * sizeof(int));
  if (pt == (int *)NULL) {  /* ne pas oublier de tester la valeur renvoyee */
    perror("Allocation memoire impossible");
    exit(-1);
  }
  /* pt pointe une zone de 1000 entiers de type int */
}
```

– `void *calloc(unsigned int ne,unsigned int no);`
  La fonction alloue de la mémoire pour un tableau de `ne` éléments dont chacun occupe

`no` octets. Comme pour `malloc()`, la fonction renvoie un pointeur sur l'adresse de départ ou `NULL` en cas d'échec. La zone mémoire est initialisée à `0`.

- `void *realloc(void *ptr, unsigned int taille);`
  La fonction réalloue l'espace mémoire dont l'adresse de départ est fixée par le pointeur `ptr`, et dont la taille en octets est fournie par `taille`. Le contenu de la mémoire est laissé inchangé.
  La fonction renvoie normalement un pointeur sur l'adresse de départ sauf s'il y a une erreur (taille demandée excessive par exemple), auquel cas elle renvoie `NULL`.
- `void free(void *ptr);` La fonction libère l'espace dont l'adresse de départ est pointée par `ptr`. Ce pointeur est l'adresse qui a été retournée par l'une des fonctions `malloc()`, `calloc()`, ou `realloc()`. **Attention**, le résultat d'un `free()` d'un pointeur non initialisé est indéterminé.

3.4 Fonctions complémentaires

Il ne peut être question ici de faire une description exhaustive de toute la bibliothèque du Langage C. Cependant, il est utile de connaître les fonctions qui suivent.

3.4.1 Fonctions mathématiques

Pour accéder à la bibliothèque mathématique du système, il faut déclarer :

```
#include <math.h>
```

De plus, lors de la compilation sous Linux, il est impératif d'invoquer `gcc` (GNU C Compiler) avec l'option de chargement de la bibliothèque mathématique `-lm` (mathematical library).

Certaines fonctions mathématiques, appelées avec des arguments inadéquats peuvent conduire à des débordements de valeurs. Si le débordement est une valeur trop grande (usuellement $> 10^{38}$), alors la valeur de type `double` HUGE_VAL est retournée. Si le débordement est une valeur trop petite (en général $< 10^{-38}$), alors la fonction renvoie 0. Les angles pour les fonctions trigonométriques sont exprimés en radians ($360° = 2\pi$ radians).

- Fonctions usuelles `double sin(double x), double cos(double x), double tan(double x), double asin(double x), double acos(double x), double atan(double x), double sinh(double x), double cosh(double x), double tanh(double x), double exp(double x), double log(double x), double log10(double x).`
- `double pow(double x, double y);`
  Renvoie x^y. La fonction puissance provoque une erreur si $x = 0$ et $y \leq 0$ ou si $x < 0$ et y n'est pas un entier.
- `double sqrt(double x);`
  Renvoie $\sqrt{x}$, la racine carrée de x, pour $x \leqslant 0$

- `double fabs(double x);`
  Renvoie la valeur absolue de x.
- `int abs(int x);` Renvoie la valeur absolue de x.
- `long labs(long x);` Renvoie la valeur absolue de x (x au format entier long).

3.4.2 Fonctions diverses

Parmi les nombreuses fonctions de la bibliothèque C, citons :

- `int rand(void);`
  Renvoie un nombre entier aléatoire. La fonction `rand()` renvoie un entier compris entre 0 et $2^{31} - 1$. La périodicité du tirage est égale à 2^{32}. Le tirage est toujours le même sauf si la racine du tirage est modifiée par la fonction `srand()`.
- `void srand(unsigned int racine);`
  Change l'origine de la séquence de nombres aléatoires (random seed). La racine initiale est 1.
- `void exit(int status);`
  Arrêt du processus courant. Cette fonction (qui est un appel système sous Linux) permet l'arrêt du processus courant. La valeur du status dépend du système d'exploitation. Sous Linux, il s'agit d'un octet qui est retourné au processus père, et dont la valeur est 0 si la fin d'exécution est normale.

4 Compléments sur les pointeurs

Un pointeur est une variable qui contient l'adresse d'une autre variable. Pour un usage ordinaire nous renvoyons le lecteur à un cours de Langage C de base, et nous n'aborderons ici que les aspects avancés des pointeurs.

4.1 Les tableaux de pointeurs

4.1.1 Principes

Puisqu'un pointeur est une variable elle-même, il peut être stocké dans un tableau, comme tout autre type de variable. Or, un pointeur unique est très comparable à un tableau à une dimension. En effet, dans un tableau les éléments sont rangés en mémoire séquentiellement. Si donc, un pointeur est positionné sur l'adresse de début du tableau, et si on ajoute 1 au pointeur, par définition le pointeur se place sur l'élément suivant du tableau. De la même façon, si on ajoute au pointeur un nombre entier n, le pointeur pointe le $(n+1)^{\text{ième}}$ élément du tableau.

La seule différence entre un pointeur et un nom de tableau est que le compilateur réserve le tableau à une position fixée dans la mémoire, ce qui revient à dire que le pointeur sur le tableau est une *constante*. Par conséquent, on ne peut pas changer l'adresse d'un tableau en manipulant son nom. Rappelons que par convention, le nom d'un tableau est l'adresse de ce tableau. Ainsi :

Exemple de code I.6

```
char *ptab1, *ptab2, tab[20];
ptab1 = tab;
ptab2 = \&tab[0];  /* ptab1 et ptab2 sont égaux a l'adresse de tab[0] */
```

Pour un tableau `int tab[dim]` où `dim` est un `int`, on a :
`tab[i] == *(tab + i)`
du point de vue du compilateur.

Cette observation s'applique également aux tableaux à deux dimensions (et plus). En effet, en C, un tableau `int a[10][20]` est considéré comme un tableau à une dimension `a[10]`, dont chaque élément est lui même un tableau de 20 éléments. Dans ces conditions, on pourra écrire, de manière similaire à l'exemple précédent :

Exemple de code I.7

```
char *ptab1, *ptab2, tab[10][20];
ptab1 =  tab[i];      /* adresse de l'élément tab[i][0] */
ptab2 = &tab[i][0];
  /* ptab1 et ptab2 sont égaux à l'adresse de tab[i][0] */
```

Il est clair que l'analogie observée entre un pointeur et un tableau à une dimension se retrouve entre un tableau de pointeur et un tableau à deux dimensions.

4.1.2 Manipulation des tableaux de pointeurs

Exemple I.2.

Considérons les déclarations suivantes :

```
int  a[10][20];
int *b[10];
```

Syntaxiquement parlant, `a[3][4]` et `b[3][4]` sont des références correctes à un entier. Cependant, la première déclaration réserve 200 places mémoire pour des entiers, tandis que la seconde ne réserve que 10 places mémoires pour stocker des adresses qui ne sont *pas initialisées*. L'initialisation de ces pointeurs doit être faite explicitement à l'aide des fonctions d'allocation dynamique de mémoire présentées plus haut.

Supposons que chacun de ces 10 pointeurs soit initialisé et pointe sur un tableau de 20 entiers. Les deux déclarations précédentes deviendront similaires mais l'encombrement mémoire de la seconde sera plus grand puisqu'en plus des 200 entiers, il y aura en outre la place de 10 pointeurs.

Cet inconvénient minime du tableau de pointeurs est compensé par un avantage important : Dans un tableau rectangulaire le nombre de colonnes est fixe, tandis que chaque

pointeur peut pointer un tableau d'un nombre quelconque de colonnes. L'intérêt des tableaux de pointeurs par rapport aux tableaux à deux dimensions est particulièrement significatif dans la manipulation des chaînes de caractères de longueurs inégales.

Illustrons l'efficacité des tableaux de pointeurs dans l'exemple qui suit. Le programme se propose de classer alphabétiquement un tableau de noms. Traditionnellement, en utilisant un tri à bulle sur un tableau de caractères à deux dimensions, lorsque deux lignes sont mal rangées, on les échange à l'aide d'une mémoire auxiliaire. En employant un tableau de pointeurs, il suffit d'échanger les pointeurs de lignes, ce qui élimine le stockage auxiliaire difficile à gérer avec des chaînes de longueurs inégales. De plus, la vitesse d'exécution est sensiblement accrue, les chaînes de caractères n'étant plus déplacées.

Exemple de code I.8

```
/*-------------------------------------------------------*/
/* Classement de chaines de caractères (version tableau)
/*-------------------------------------------------------*/
#include <stdio.h>
#define   N1  100    /* nombre de lignes */
#define  N2  50    /* nombre de colonnes */

void classe_t(int n1, int n2, char tab[N1][N2])
{    /* classement d'un tableau de chaines de caractères */
  char *pt;
  int   i,j;

  for (i = 0; i < n1-1; i++)
    for (j = i+1; j < n1; j++)
      if(strcmp(tab[i],tab[j]) > 0)
      {
        pt = (char *)malloc(n2);
        strcpy(pt, tab[i]);
        strcpy(tab[i], tab[j]);
        strcpy(tab[j], pt);
      }
}

main(void)
{
  static char tab[N1][N2];
  int         i = 0, N = 0;

  while (gets(tab[N]) != NULL)  /* lecture des données */
      N++;
```

```
  classe_t(N,N2,tab);
  for (i = 0; i < N; i++)
      printf("\%s\n", tab[i]);
}
```

Exemple de code I.9

```
/*-------------------------------------------------------*/
/* Classement de chaines de caractères (version pointeur)
/*-------------------------------------------------------*/
#include <stdio.h>
#define  N1  100   /* nombre de lignes   */
#define  N2  50    /* nombre de colonnes */

void classe_p(int n1, char *tab[N1])
{    /* classement d'un tableau de chaines de caracteres */
  char *pt;
  int   i,j;

  for (i = 0; i < n1-1; i++)
    for (j = i+1; j < n1; j++)
      if(strcmp(tab[i], tab[j]) > 0)
        {
        pt     = tab[i];
        tab[i] = tab[j];
        tab[j] = pt;
        }
}

main(void)
{
  static char *tab[N1];
  char         s[N2];
  int          i = 0, N = 0;

  while (gets(s) != NULL) {  /* lecture des données */
      tab[N] = (char *)malloc(strlen(s));
      strcpy(tab[N], s);
      N++;
  }
  classe_p(N, N2, tab);
  for (i = 0; i < N; i++)
      printf("\%s\n", tab[i]);
```

```
}
```

4.1.3 Ecueils à éviter

Il est courant de transmettre l'adresse d'un tableau à une dimension qui est récupérée au niveau de la fonction par un pointeur.

Exemple de code I.10

```
main(void)
{
  int longueur(char *s);
  static char a[10] = "exemple";

  printf("Longueur de la chaine = \%d\n", longueur(a));
}

int longueur(char *s)
{
  int n = 0;
  while (*s++) n++;
  return n;
}
```

Il faut être conscient que cette disposition n'est guère applicable pour des tableaux à plus d'une dimension. En effet, l'adresse du tableau peut toujours être récupérée par un pointeur, mais pour accéder aux éléments, il est impératif de connaître leur rangement c'est à dire la taille de chaque dimension. Pour accéder à un élément donné, il faut recalculer l'adresse de l'élément par rapport au début du tableau. L'utilisation d'un pointeur perd alors tout intérêt.

D'autre part, considérons la déclaration suivante : `char **argv;` Cette déclaration signifie que `argv` pointe sur une variable contenant l'adresse d'un ou plusieurs caractères (pointeur sur pointeur). Il y a donc une grande analogie avec la déclaration : `char *argv[];` qui stipule que argv est un pointeur sur un tableau dont chaque élément est un pointeur sur une chaîne de caractères. Dans de nombreux cas, les deux déclarations sont interchangeables.

4.2 Pointeurs, structures et unions

Comme pour n'importe quel type, une variable peut contenir l'adresse d'une structure ou d'une union. Il est donc possible de définir des pointeurs sur des structures ou sur des unions.

```
struct st {
  int   i;
```

```
  float x;
} *pt;
```

L'accès à un membre de la structure peut s'écrire des différentes manières suivantes :

```
pt.x      /* adresse de la variable flottante */
(*pt).x   /* contenu de la variable flottante */
pt->x     /* équivalent à (*pt).x             */
```

Ces notations peuvent être utilisées pour une union. Une structure ne peut contenir un membre identique à elle-même. En revanche, un membre de structure peut être un pointeur sur une structure de même type. Cette possibilité est à l'origine de l'usage massif des pointeurs avec les structures.

Exemple I.3.

Considérons la représentation d'un arbre à division binaire. On peut le désigner par une structure du type suivant :

```
struct noeud {
  char *nom;            /* nom du noeud         */
  int compteur;         /* niveau de division   */
  struct noeud *droite; /* sous-noeud de droite */
  struct noeud *gauche; /* sous-noeud de gauche */
} var, *pt;
```

La structure noeud ne se contient pas elle-même. Droite et gauche sont seulement deux pointeurs sur des structures du type noeud. Ce type de définition imbriquée est souvent utilisé pour créer des listes chainées. L'usage des pointeurs avec les structures (ou avec les unions) entraînent des notations parfois compliquées qu'il faut bien distinguer. Avec la structure de l'exemple précédent, on peut notamment avoir les combinaisons des opérateurs `* .` et `->` suivantes :

- `var.nom` adresse de la chaîne de caractères qui correspond au premier membre de `var`.
- `*var.nom` premier caractère de cette chaîne.
- `*(var.nom)` idem que l'item précédent.
- `(*pt).compteur` valeur de compteur pour la variable `pt`.
- `pt->compteur` idem que l'item précédent.
- `pt->nom` adresse de la chaîne nom pour la variable `pt`.
- `*pt->nom` premier caractère de cette chaîne.
- `pt->droite` adresse de la structure droite.
- `pt->droite->compteur` valeur de `compteur` de la structure droite.

Rappelons que la notation `var.*nom` n'a pas de sens puisque nom n'est pas une variable mais un membre de structure.

4.3 Pointeurs sur fonctions

Il est possible en C de définir des pointeurs sur fonctions, qui peuvent être assignés à l'adresse d'une fonction, passés comme argument d'une fonction, ou retournés par une fonction. Un pointeur sur fonction est défini de la façon suivante :

```
type (*pt_fonc)();
```

où `type` est un tyope C valide. Le pointeur `pt_fonc` est une variable contenant l'adresse d'une fonction qui renvoie une valeur du type spécifié. Les parenthèses entourant `*pt_fonc` sont obligatoires sinon la définition indiquerait une fonction ordinaire `pt_fonc` retournant un pointeur sur une valeur du type spécifié. L'initialisation du pointeur `pt_fonc` s'effectue en l'assignant à un nom de fonction de la manière suivante :

Exemple de code I.11

```
float somme(float x, float y)
{
  return (x+y);
}
main(void)
{
  float (*ptf)();
  float a,b;

  scanf("%f%f", &a, &b);
  ptf = somme;
  y   = (*ptf)(a, b);
  ...
}
```

Un pointeur sur fonction se manipule donc de façon similaire à une fonction ordinaire. La différence essentielle réside dans le fait que le nom d'une fonction ordinaire contient une adresse constante (celle de la fonction), tandis que le pointeur contient l'adresse d'une fonction qui peut être changée par simple affectation en cours d'exécution. Les pointeurs sur fonction servent principalement à être passés comme paramètres à d'autres fonctions ou sont utilisés comme tables de décision.

Cela est en particulier vrai dans l'appel système Linux qui permet de contrôler les signaux d'interruptions des processus. La syntaxe est la suivante :

`signal(int sig, void (*fct)());` Cet appel définit l'action à exécuter par le noyau Linux lors de la survenue du signal `sig`. Le deuxième paramètre est un pointeur sur fonction. Ce pointeur doit être initialisé à la fonction choisie par l'utilisateur (cf section 1, p. 139).

5 Compilation et édition de liens

Pour tirer pleinement parti de la puissance du langage, l'utilisateur doit être informé d'un certain nombre de facilités offertes par le compilateur C.

Rappelons qu'un compilateur est un programme informatique qui transforme un code source écrit dans un langage de programmation (le langage source) en un autre langage (le langage cible). Dans la plupart des cas, et pour qu'il puisse être exploité par la machine, le compilateur traduit le code source, écrit dans un langage de haut niveau d'abstraction, facilement compréhensible par l'humain, vers un langage de plus bas niveau, un langage d'assemblage ou langage machine. Dans le cas de langage semi-compilé (ou semi-interprété), le code source est traduit en un langage intermédiaire, sous forme binaire (code objet ou bytecode), avant d'être lui-même interprété ou compilé. Inversement, un programme qui traduit un langage de bas niveau vers un langage de plus haut niveau est un décompilateur.

Un compilateur effectue les opérations suivantes : analyse lexicale, pré-traitement (préprocesseur), décomposition analytique (parsing), analyse sémantique, génération de code et optimisation de code. Quand le programme compilé (code objet) peut être exécuté sur un ordinateur dont le processeur ou le système d'exploitation est différent de celui du compilateur, on parle de compilation croisée.

La compilation est souvent suivie d'une étape d'édition des liens, pour générer un fichier exécutable. L'édition des liens permet de créer des fichiers exécutables ou des bibliothèques dynamiques ou statiques, à partir de fichiers objets. La compilation en fichiers objets laisse l'identification de certains symboles à plus tard. Avant de pouvoir exécuter ces fichiers objets, il faut résoudre les symboles et les lier à une bibliothèque. Le lien peut être

- statique : le fichier objet et la bibliothèque sont liés dans le même fichier exécutable.
- dynamique : le fichier objet est lié avec la bibliothèque, mais pas dans le même fichier exécutable ; les liens sont établis lors du lancement de l'exécutable.

5.0.1 Compléments sur le pré-processeur

Le préprocesseur a toujours comme objectifs de réaliser 3 actions : les substitutions, la compilation conditionnelle, et l'inclusion de fichiers. La norme a apporté quelques compléments à ces fonctions de base.

Opérateurs # et ## Considérons la déclaration de substitution suivante :

```
#define TAILLE 1000
```

Cette macro-définition entraine le remplacement de `TAILLE` par sa valeur partout dans le code qui suit, sauf dans les chaînes entre guillemets. Pour contourner cette disposition utile mais qui peut dans certains cas être fâcheuse, le préprocesseur fournit désormais l'opérateur `#`.

Cet opérateur, placé devant la chaîne à substituer, entraine la substitution, le résultat étant placé entre guillemets. Il est alors possible d'utiliser l'opérateur `#` pour effectuer des substitutions dans les chaînes.

Exemple I.4.

Par exemple, dans le code suivant :

```
#define imprime_ad(expr)  printf(#expr " = \%g\n",expr)
main(void)
{
  float x,y;
  printf("Entrer x et y: ");
  scanf("\%f\%f", &x, &y);
  imprime_ad(x+y);
}
```

l'instruction `imprime_ad(x+y)` sera convertie en

```
printf("x+y" "=%g\n",x+y);
```

Ce qui est équivalent à :

```
printf("x+y =%g\n",x+y);
```

Si la chaîne à substituer dans un `#define` contient l'opérateur `##`, le signe `##` est supprimé ainsi que les blancs qui l'entourent, et les opérandes de part et d'autre du `##` sont concaténés.

```
#define coller(debut, fin) debut ## fin
```

L'appel de la macro `coller(var, 10)` entrainera la formation de la chaîne `var10`.

Directive `#elif` Comme pour le Shell, le mot-clé `elif` a été ajouté. Il a la même signification que le succession `#else #if`

Exemple de code I.12

```
#ifndef  TAILLE
#define  TAILLE  1000
#elif TAILLE != 1000
  x = TAILLE*2;
#else
  x = TAILLE*3;
#endif
```

5.1 Compilateur gcc

Le compilateur `gcc` fait partie de la famille des compilateurs GNU. Elle comprend : `gcc` pour le langage `C`, `g++` pour `C++`, `gcj` pour `Java`, `gnat` pour `Ada`, `gccgo` pour `Go`.

Les compilation et édition de liens du fichier source `fich.c` en exécutable `fich` se fait typiquement avec l'option `-o` par :

```
gcc -o fich fich.c
```

Si l'on dispose de plusieurs fichiers sources `fich1.c` et `fich2.c` par exemple, on produit d'abord par compilation les fichiers objets avec l'option `-c` :

```
gcc -c fich1.c
gcc -c fich2.c
```

Ensuite, l'édition de liens se fait à partir des fichiers objets par :

```
gcc -o fichExe fich1.o fich2.o
```

Si l'un des fichiers utilise la bibliothèque mathématique, la commande précédente devient :

```
gcc -o fichExe fich1.o fich2.o -lm
```

Dans le cas d'une bibliothèque nommée `libnombibli.a`, l'édition de liens sera de la forme

```
gcc -o executable objet1 [objet 2 ...] -lnombibli
```

L'option `-g` inclut des informations utiles au déboguage. Enfin, l'option `-Wall` ("all Warnings") affiche tous les messages d'avertissements, ce qui peut être utile.

5.2 Arguments sur la ligne de commande

La fonction `main()`, point d'entrée de tout programme C, a deux arguments :

`int main(`	Point d'entrée du programme
`int argc,`	Nombre d'arguments + 1
`cahr *argv[])`	Tableau de la commande et des arguments

Le permier argument, `argc`, est le nombre d'arguments + 1, et le deuxième, `argv`, est constitué de `argv[0]`, le nom de la commande, `argv[1]`, le premier argument, `argv[2]`, le deuxième argument, ..., `argv[argc-1]`, le dernier argument. Considérons le programme suivant, de code source `exmain.c` :

Exemple de code I.13

```
#include <stdio.h>

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

    int i = 0;

    printf("Nombre d'arguments + 1 : %d\n", argc);
    for (i = 0; i < argc; ++i)
        printf ("Argument %d : argv[%d] == %s\n", i + 1, i, argv[i]);

    return 0;

}
```

La compilation se fera par exemple par :

Exemple de code I.14

```
gcc -o exmain exmain.c
```

Puis, l'appel de l'exécutable pourra se faire par exemple :

Exemple de code I.15

```
./exmain 1 altier 8.089
```

Alors, lors de l'exécution de la fonction `main()`, le premier argument, `argc`, correspondant au nombre d'arguments + 1, vaudra 4 ; le deuxième argument, `argv` sera composé de : `argv[0] == "exmain"`, `argv[1] == "1"`, `argv[2] == "altier"`, `argv[3] == "8.089"`.

5.3 Rappels sur la norme ANSI

5.3.1 Déclarations externes

Dans un programme C, la définition d'un objet, variable ou fonction est a priori "externe", c'est à dire que cet objet peut être manipulé par des unités de compilation séparées. Le fait de mettre le mot clé `static` devant un tel objet annule cette propriété, l'objet n'étant plus visible qu'à l'intérieur du fichier où il est défini.

La norme ANSI a apporté quelques nuances quant au sens des déclarations externes. La nuance entre définition et déclaration a été estompée. Désormais, une déclaration externe d'un objet qui comporte une initialisation est une **définition**.

Une déclaration externe qui ne comporte ni initialisation, ni le mot clé `extern`, est considérée comme un essai de définition. Dans une unité de compilation, si la définition d'un objet apparait, tous les autres essais de définitions de cet objet seront considérés comme des déclarations redondantes et ignorées. Si aucune définition de l'objet n'apparait, tous les essais de définitions aboutiront à une seule définition avec initialisation à 0 de l'objet.

Ces dispositions peuvent se résumer par une règle : un objet doit n'avoir qu'une définition. Pour les objets avec des liens internes, cette règle s'applique à l'intérieur d'un fichier de compilation. Pour les objets à liens externes, la règle s'applique au programme tout entier.

5.3.2 Visibilité et édition de liens

Il y a deux types de visibilité à considérer. La première qu'on désigne par visibilité lexicale est la région du programme dans laquelle un identificateur est connu. La seconde concerne les connections qui existent entre des identificateurs avec liens externes et qui sont compilés dans des unités séparées. Cette visibilité relève de l'édition de liens.

Visibilité lexicale La visibilité lexicale d'un objet dans une déclaration externe part de sa déclaration et se termine à la fin du fichier. La visibilité d'un paramètre de fonction commence au début du bloc définissant le corps de la fonction et s'achève à la fin de ce bloc. La visibilité d'un identificateur déclaré en début de bloc s'arrête en fin de bloc. La visibilité d'une étiquette est le corps de la fonction où elle apparait.

Si un identificateur est déclaré en tête d'un bloc (y compris celui constituant le corps d'une fonction), toute déclaration de cet identificateur à l'extérieur du bloc est suspendue jusqu'à la fin du bloc.

Dans la norme ANSI, le même identificateur peut être utilisé dans un même champ de visibilité pour désigner des objets différents à condition qu'ils n'appartiennent pas à la même famille d'objet. Ces familles sont :

- Variables, fonctions, noms de `typedef`, constantes symboliques d'énumération.
- Étiquettes
- Noms de structures, unions et énumération.
- Membres de structures et unions.

Il est cependant évident que, pour des raisons de lisibilité, l'usage d'un identificateur avec des sens différents est une très mauvaise pratique.

Edition de liens Dans un fichier source, toutes les déclarations relatives à un même objet avec un lien interne font référence à un objet unique à l'intérieur du fichier. Toutes les déclarations relative à un même objet avec un lien externe font référence à un objet unique pour l'ensemble du programme. A l'intérieur d'un bloc, un identificateur s'il n'est pas déclaré `extern` n'a pas de lien et est unique au bloc.

5.4 L'outil splint

`splint`, abbréviation de "Secure Programming Lint", est un outil de vérification statique de code C. Il effectue une analyse approfondie du code à compiler ; il produit tous les messages et avertissements fournis par le compilateur C standard, auxquels il ajoute de nombreux avertissements quant à des problèmes potentiels. Son utilité majeure se situe dans le contrôle qu'il exerce sur les définitions et déclarations désignant un même objet à travers de multiples fichiers. `splint` permet ainsi de découvrir des erreurs qui seraient difficiles de trouver autrement.

`splint` effectue essentiellement trois types de contrôles : la cohérence dans l'usage des objets manipulés, la portabilité des codes et la mise en évidence de constructions suspectes. `splint` a enfin la possibilité d'interpréter des annotations spéciales du code source, qui donne une vérification plus forte qu'il n'est possible par simple inspection du code source.

5.4.1 Tests de cohérence

Ces tests contrôlent notamment l'adéquation des types de variables, les valeurs retournées par les fonctions, et les variables ou fonctions qui sont définies et non utilisées (et vice versa).

Exemple I.5.

Considérons l'exemple suivant

```
   /* fichier f1.c */
1  int f(int x, int y)
2  {
3    return x+y;
4  }
   /* fichier f2.c */
1  int main()
2  {
3    int *x;
4    int y, z;
5    extern int f(int x, int y);
6    z = f(1,x);
7  }
```

La compilation par `splint` donne, entre autres, le diagnostic (le détail peut dépendre de la machine) :

```
f2.c:6:11: Function f expects arg 2 to be int gets int *: x
  Types are incompatible. (Use -type to inhibit warning)
```

5.4.2 Constructions suspectes

Exemple I.6.

Considérons Le code suivant, dans le fichier `exsplint.c`

```
 1  #include <stdio.h>
 2  int main()
 3  {
 4      char c;
 5      while (c != 'x');
 6      {
 7          c = getchar();
 8          if (c = 'x')
 9              return 0;
10          switch (c) {
11          case '\n':
12          case '\r':
13              printf("Newline\n");
14          default:
15              printf("%c",c);
16          }
```

```
17      }
18      return 0;
19  }
```

La sortie de `splint` sur le code précédent est la suivante :

```
exsplint.c:5:12: Variable c used before definition
  An rvalue is used that may not be initialized to a value on some
  execution path. (Use -usedef to inhibit warning)
exsplint.c:5:12: Suspected infinite loop.  No value used in loop
                   test (c) is modified by test or loop body.
  This appears to be an infinite loop. Nothing in the body of the
  loop or the loop test modifies the value of the loop test. Perhaps
  the specification of a function called in the loop body is missing
  a modification. (Use -infloops to inhibit warning)
exsplint.c:7:9: Assignment of int to char: c = getchar()
  To make char and int types equivalent, use +charint.
exsplint.c:8:13: Test expression for if is assignment expression:
  c = 'x' The condition test is an assignment expression. Probably,
  you mean to use == instead of =. If an assignment is intended, add
  an extra parentheses nesting (e.g., if ((a = b)) ...) to suppress
  this message. (Use -predassign to inhibit warning)
exsplint.c:8:13: Test expression for if not boolean, type char: c = 'x'
  Test expression type is not boolean. (Use -predboolothers to inhibit
  warning)
exsplint.c:14:17: Fall through case (no preceding break)
  Execution falls through from the previous case (use /*@fallthrough@*/
  to mark fallthrough cases). (Use -casebreak to inhibit warning)
```

Le code corrigé est le suivant

```
#include <stdio.h>
int main()
{
    char c = (char) 0;  // Added an initial assignment definition.

    while (c != 'x') {
        c = (char) getchar();  // Type-cast to char.
        if (c == 'x') // Fixed the assignment error to make it a
                      // comparison operator.
            return 0;
        switch (c) {
        case '\n':
        case '\r':
            printf("Newline\n");
            break;  // Added break statement to prevent fall-through.
        default:
```

```
            printf("%c",c);
            break;  // Added break statement to default catch, out of
                    // good practice.
        }
    }
    return 0;
}
```

L'examen exhaustif de toutes les propriétés de `splint` serait fastidieux et inutile car les messages sont le plus souvent explicites. Nous nous sommes bornés ici à donner un aperçu de l'intérêt de ce vérificateur.

5.5 L'outil make

L'utilitaire `make` est un outil pour compiler et charger un programme fait de multiples modules. Il a pour objectif de vérifier que tous les fichiers nécessaires au chargement existent et sont à jour. Il créé le module exécutable à partir d'un fichier décrivant les opérations à faire et les relations existant entre les divers fichiers. Ce fichier de description est appelé par convention `makefile`, ou `GNUmakefile` ou `makefile` ou encore `Makefile` (la recherche est faite dans cet ordre dans le répertoire courant). L'option `-f` suivie d'un nom de fichier permet d'utiliser un autre nom pour le fichier de description.

5.5.1 Un exemple simple

Pour illustrer l'usage de `make`, considérons l'exemple suivant.

Exemple I.7.

Soit un programme nommé `prog` chargé à partir de 3 fichiers sources `x.c`, `y.c`, `z.c` avec la bibliothèque mathématique `libm`. On suppose que `x.c` et `y.c` ont des déclarations communes dans un fichier `defs.h`, c'est à dire qu'ils contiennent :

```
#include "defs.h"
```

Le fichier `makefile` aura l'allure suivante :

```
# Ce fichier makefile est destiné à créer le module exécutable prog

prog:  x.o  y.o  z.o
    gcc -o  prog  x.o  y.o  z.o -lm
x.o  y.o:  defs.h
```

La commande est exécutée en tapant simplement `make` en ligne de commande. La première ligne est un commentaire. `make` ignore tous les caractères qui suivent un `#` jusqu'à la fin de la ligne courante. Un fichier de commande peut contenir des lignes vides comme cela est le cas à la deuxième ligne. Elles sont ignorées par `make`. La troisième ligne indique que l'exécutable `prog` dépend des trois fichiers `.o` désignés, et

> la ligne suivante décrit comment charger le programme. La dernière ligne significative du fichier spécifie que `x.o` et `y.o` dépendent de `defs.h`.
>
> `make` fait les vérifications adéquates de date et recompile ce qui doit l'être. Par exemple si les dates de `x.o` ou `y.o` sont antérieures à la dernière modification de `defs.h`, `make` recherche les fichiers `x.c` et `y.c` et les recompile avant d'exécuter la ligne 4.

5.5.2 Fonctionnement général

Le fichier de configuration `makefile` décrit des cibles, matérialisant des actions à éxécuter, (ces cibles sont souvent des fichiers, mais pas toujours) décrit de quelles autres cibles elles dépendent, et par quelles actions (quelles commandes) y parvenir. Afin de reconstruire une cible spécifiée par l'utilisateur, `make` va chercher les cibles nécessaires à la reconstruction de cette cible, et ce récursivement. Les règles de dépendance peuvent être explicites (noms de fichiers donnés explicitement) ou implicites (via des motifs de fichiers ; par exemple, `*.o` dépend de `*.c`, si celui-ci existe, via une recompilation).

`make` peut être appelé avec une liste de fichiers cibles par la ligne de commande suivante :

```
make CIBLE [CIBLE ...]
```

Si aucun argument n'est passé, make construira la première cible specifiée dans le `makefile`. Traditionnellement, cette cible est appelée `all`. La date de modification du fichier cible permet à make de déterminer si celui-ci est à jour. Alors la cible n'est pas reconstruite.

5.5.3 Les règles

Un `makefile` est constitué de règles. La forme la plus simple de règle est la suivante :

```
cible [cible ...]: [composant ...]
[tabulation] commande 1
  ...
[tabulation] commande n
```

La cible est le plus souvent un fichier à construire, mais elle peut aussi définir une action (effacer, compiler...). Les composants sont des pré-requis nécessaires à la réalisation de l'action définie par la règle. Autrement dit, les composants sont les cibles d'autres règles qui doivent être réalisées avant de pouvoir réaliser cette règle. La règle définit une action par une série de commandes. Ces commandes définissent comment utiliser les composants pour produire la cible. Chaque commande doit être précédée par un caractère de tabulation. Les commandes sont exécutées par un shell distinct. Voici un exemple de makefile :

```
all: prog
prog:  x.o  y.o  z.o
    gcc -o  prog  x.o  y.o  z.o -lm
x.o  y.o:  defs.h
clean:
    rm -f x.o y.o z.o
```

À l'exécution de ce ficher makefile par l'intermédiaire de la commande `make all` ou de la commande `make`, la règle '`all`' est exécutée. Elle nécessite la réalisation du composant

'prog' qui est associé à la règle 'prog'. Cette règle sera donc exécutée automatiquement avant la règle 'all'. En revanche, la règle clean n'est pas exécutée et la commande make clean n'exécute que la règle 'clean'. À noter qu'une règle ne comporte pas nécessairement de commande.

Les composants ne sont pas toujours des cibles vers d'autres règles, ils peuvent aussi être des fichiers nécessaires à la construction du fichier cible :

```
sortie.txt: fichier1.txt fichier2.txt
        cat fichier1.txt fichier2.txt > sortie.txt
```

L'exemple de règle ci-dessus construit le fichier sortie.txt en utilisant les fichiers fichier1.txt et fichier2.txt. En exécutant le makefile, make vérifie si le fichier sortie.txt existe et s'il n'existe pas, il le construira à l'aide de la commande définie dans la règle.

Les lignes de commande peuvent avoir un ou plusieurs des trois préfixes suivants :

- Un signe moins -, spécifiant que les erreurs doivent être ignorées.
- Un arobase @, spécifiant que la commande ne doit pas être affichée dans la sortie standard avant d'être exécutée.
- Un signe plus +, la commande est exécutée même si make est appelé dans un mode "ne pas exécuter".

5.5.4 Règle multiple

Lorsque plusieurs cibles nécessitent les mêmes composants et sont construites par les mêmes commandes, il est possible de définir une règle multiple. Par exemple :

```
all : cible1, cible2

cible1 : texte1.txt
        echo texte1.txt
cible2 : texte1.txt
        echo texte1.txt
```

peut être remplacé par :

```
all : cible1, cible2

cible1, cible2 : texte1.txt
        echo texte1.txt
```

À noter que, dans ce cas, la cible 'all' est obligatoire, sans quoi seule la cible 'cible1' sera exécutée.

Pour connaître la cible concernée, il est possible d'utiliser la variable $@, ainsi par exemple :

```
all : cible1.txt, cible2.txt

cible1.txt, cible2.txt : texte1.txt
        cat texte1.txt > $@
```

créera deux fichiers, cible1.txt et cible2.txt, ayant le même contenu que texte1.txt.

5.5.5 Les macros

la commande `make` reconnait les macros définitions de la forme :

```
CHAINE1 = Chaine2
```

Dans cette syntaxe voisine de celle du shell, `CHAINE1` joue le rôle de nom de variable, et `Chaine2` apparait comme le contenu. Après une telle définition, `make` remplace chaque occurence de `$(CHAINE1)` par `Chaine2`. Les imbrications sont autorisées jusqu'à 10, la plus interne étant traitée en premier. Ainsi, `$($(CHAINE1))` sera remplacé d'abord par `$(Chaine2)`, et le contenu de `Chaine2` (s'il existe) fournira la chaîne de caractères finale. Il n'y a pas de parenthèses à mettre après le `$` si la macro n'a qu'un seul caractère :

```
$z    # invocation de la macro z
$(z)  # identique à la ligne précédente
$2    # invocation de la macro 2
```

`make` possède quelques macros internes, notamment `$*`, `$@`, `$<`, et `$?` :

- `$*` est remplacé par le nom du fichier source (sans suffixe).
- `$@` est remplacé par le nom complet du fichier cible exécutable.
- `$<` est remplacé par tous les fichiers sources dont les modules relogeables ne sont pas à jour. Par exemple `gcc -c $<` demande de recompiler tous les sources dont les relogeables sont périmés.
- `$?` est remplacé par tous les noms qui ont été trouvés être plus récents que le fichier cible exécutable.

Les variables d'environnement sont également disponibles sous forme de macros. Les macros dans un makefile peuvent être écrasées par les arguments passés à `make`. La commande est alors :

```
make MACRO="valeur" [MACRO="valeur" ...] CIBLE [CIBLE ...]
```

Les macros peuvent être composées de commandes shell en utilisant l'accent grave ` :

```
DATE = `date`
```

Il existe plusieurs manières de définir une macro :

- Le `=` est une affectation par référence (on parle d'expansion récursive)
- Le `:=` est une affectation par valeur (on parle d'expansion simple)
- Le `?=` est une affectation conditionnelle. Elle n'affecte la macro que si cette dernière n'est pas encore affectée.
- Le `+=` est une affectation par concaténation. Elle suppose que la macro existe déjà.

5.5.6 Les règles de suffixes

Ces règles permettent de créer des makefile pour un type de fichier. Il existe deux type de règles de suffixes : les doubles et les simples.

Une règle de double suffixes est définie par deux suffixes : un suffixe cible et un suffixe source. Cette règle reconnaitra tout fichier du type « cible ». Par exemple, si le suffixe cible est `.txt` et le suffixe source est `.html`, la règle est équivalente à `*.txt : *.html` (où `*` signifie n'importe quel nom de fichier). Une règle de suffixes simples n'a besoin que

d'un suffixe source. La syntaxe pour définir une règle de double suffixes est :

```
.suffixe_cible.suffixe_source :
```

Attention, une règle de suffixe ne peut pas avoir de composants supplémentaires. La syntaxe pour définir la liste des suffixes est :

```
.SUFFIXES: .suffixe_cible .suffixe_source
```

Un exemple d'utilisation des règles de suffixes :

```
.SUFFIXES: .txt .html

# transforme .html en .txt
.html.txt:
        lynx -dump $<   >   $@
```

La ligne de commande suivante transformera le fichier `fich.html` en `fich.txt` :

```
make -n fich.txt
```

L'utilisation des règles de suffixes est considérée comme *obsolète car trop restrictive*.

5.5.7 Exemple plus complet

Voici un exemple plus complet :

Exemple de code I.16

```
# Indiquer quel compilateur est à utiliser
 CC      ?= gcc

# Spécifier les options du compilateur
CFLAGS  ?= -g
LDFLAGS ?= -L/usr/openwin/lib
LDLIBS  ?= -lX11 -lXext

# Commande de nettoyage
RM      ?= rm -f

# Reconnaître les extensions de nom de fichier *.c et *.o
#   comme suffixe
SUFFIXES ?= .c .o
.SUFFIXES: $(SUFFIXES) .

# Nom de l'exécutable
PROG  = life

# Liste des sources
SRC  = main.c window.c Board.c Life.c BoundedBoard.c
# Liste de fichiers objets nécessaires pour le programme final
OBJS  = main.o window.o Board.o Life.o BoundedBoard.o
```

```
# cible par défaut
all: $(PROG)

# Étape de compilation et d'éditions de liens
# ATTENTION, les lignes suivantes contenant "$(CC)" commencent
# par un caractère TABULATION et non pas des espaces
$(PROG): $(OBJS)
	$(CC) $(CFLAGS) $(LDFLAGS) $(LDLIBS) -o $(PROG) $(OBJS)

.c.o:
	$(CC) $(CFLAGS) -c $*.c

clean:
	$(RM) $(PROG) $*.o

zip:
    tar -zcvf $(PROG).tar.gz $(SRC) Makefile
```

De nos jours, les fichiers Makefile sont de plus en plus rarement écrits à la main par le développeur mais construits automatiquement à partir d'outils tels qu'autoconf ou cmake qui facilitent la génération de Makefile complexes et spécifiquement adaptés à l'environnement dans lequel les actions de production sont censées se réaliser.

5.6 Le générateur de makefile CMake

Spécificités

`CMake` est un système de construction logicielle. C'est un logiciel libre (licence BSD), multilangage et multiplate-forme.

À partir du fichier descriptif du projet (nommé `CMakeLists.txt`) et des informations sur la configuration logicielle de la machine effectuant la compilation (listées dans le fichier `CMakeCache.txt`, prérempli lors de la phase d'analyse de la configuration), `CMake` est capable de générer les types de fichiers suivants :

- un fichier Makefile ;
- un fichier de projet d'environnement de développement intégré ;
- ou une combinaison des deux (pour les environnements supportant les fichiers Makefile).

Alors que d'autres systèmes de construction ne sont utilisables que sous un ensemble restreint de systèmes d'exploitation, `CMake` est disponible aussi bien sous GNU/Linux que sous Windows et MacOS, pour ne citer qu'eux. Plus spécifiquement, `CMake` est utilisable avec les environnements suivants : `Eclipse` (avec le plugin `CMakeBuilder`), `KDevelop 4`, `QtCreator`, `NetBeans`, `Code::Blocks`, `Visual Studio`.

`CMake` gère les langages `C`, `C++` et `Java`.

De plus, comme vous pourrez vous-même en juger dans la suite de ce cours, `CMake` est très facile à utiliser (ce qui est loin d'être le cas de tous les outils de cette catégorie) et la syntaxe du fichier `CMakeLists.txt` est à la fois simple et efficace.

Exemple simple

Considérons avoir un projet constitué des fichiers sources suivants : `main.c`, `display.h`, `display.c`, `smooth.h`, `smooth.c`, et un exécutable `smooth_exe`.

La syntaxe du fichier `CMakeLists.txt` est on ne peut plus simple. Elle est exclusivement constituée d'appels de commandes. Les arguments à passer aux commandes sont placés entre parenthèses et séparés par des espaces ou des sauts de ligne. Les différents appels de commandes sont séparés de la même façon :

```
nom_de_la_commande(argument1 argument2 argument3)

nom_d_une_autre_commande(
        argument1
        argument2
)
```

À noter qu'aucun saut de ligne n'est toléré entre le nom de la commande et la parenthèse ouvrante.

Si un argument doit lui-même contenir des espaces, il pourra être entouré de guillemets doubles :

```
commande(argument "argument avec des espaces")
```

Les commentaires sont marqués par un `#` en début de ligne :

```
#ceci est un commentaire
commande(argument1 argument2)
```

Considérons le code suivant dans le fichier `CMakeLists.txt` du projet :

Exemple de code I.17

```
#Déclaration du projet
project(SmoothProject)

#Déclaration de l'exécutable
add_executable(
        smooth_exe
        src/display.h
        src/display.c
        src/main.c
        src/smooth.c
        src/smooth.h
)
```

La commande `project()` est utilisée pour déclarer un projet.

- Le premier paramètre est le nom du projet.
- Cette commande prend également un second paramètre optionnel. Il s'agit du langage dans lequel le code du programme à compiler est écrit. Dans la majorité des cas, vous pouvez simplement ignorer cet argument, `CMake` étant capable de reconnaître lui-même le langage en analysant les sources.

Il faut ensuite définir les cibles (ensemble des exécutables et des bibliothèques à générer). Dans l'exemple précédent, il s'agit de créer un unique exécutable à partir du code source. Pour ce faire, on utilise la commande `add_executable()`. Le premier paramètre est le nom de l'exécutable. Les arguments suivants constituent la liste des fichiers sources, y compris les fichiers include.

Dans le dossier racine du projet, la commande

```
cmake . -G"generateur"
```

générera la fichier correspondant au générateur fourni en deuxième argument. Le premier paramètre désigne le répertoire où se situe le fichier `CMakeLists.txt` du projet, ici le dossier courant.

5.7 L'outil `valgrind`

`valgrind` est un outil permettant de déboguer, effectuer du profilage de code et mettre en évidence des fuites mémoires. Ce logiciel est modulaire ; parmi les modules on trouve `memcheck`, qui permet de débusquer les failles dans un programme au niveau de l'utilisation de la mémoire. `memcheck` vérifie entre autres :

- Que l'on n'utilise pas de valeurs ou de pointeurs non initialisés
- Que l'on n'accède pas à des zones mémoire libérées ou non allouées
- Que l'on ne libère pas deux fois une zone mémoire
- Que l'on n'oublie pas de libérer la mémoire allouée. Des options permettent de connaître avec précision les zones de mémoire qui sont perdues.
- Que l'on passe des arguments valides à certaines fonctions de la bibliothèque standard comme la fonction memcpy().

Exemple I.8.

Considérons le code exemple suivant, dans le fichier `ptrBug.c` :

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

int main(void)
{
    int *tableau;
    int  i;
```

```
    for(i = 0; i < 10; i++)
      tableau[i] = 2*i;
}
```

Après avoir compilé avec les informations symboliques :

```
gcc -g -o ptrBug ptrBug.c
```

La sortie de `valgrind ./ptrBug` contient notamment les lignes suivantes :

```
==4785== Use of uninitialised value of size 4
==4785==    at 0x80483D1: main (ptrError.c:10)
==4785==
==4785== Invalid write of size 4
==4785==    at 0x80483D1: main (ptrError.c:10)
==4785==  Address 0x0 is not stack'd, malloc'd or (recently) free'd
```

Voici un autre exemple avec une lecture en dehors des bornes d'une zone mémoire allouée dynamiquement :

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

int main(void)
{
    int *tableau;
    int  i;

    tableau = (int *)calloc(10, sizeof(int));
    if (tableau == NULL) {
      perror("initialisation impossible");
      exit(1);
    }

    for(i = 0; i < 10; i++)
      tableau[i] = 2*i;

    for(i = 0; i < 11; i++)
      printf("tableau[%d] = %d\n", i, tableau[i]);

}
```

La sortie de `valgrind` contient entre autres

```
tableau[1] = 2
tableau[2] = 4
tableau[3] = 6
tableau[4] = 8
tableau[5] = 10
tableau[6] = 12
```

```
tableau[7] = 14
tableau[8] = 16
tableau[9] = 18
==4795== Invalid read of size 4
==4795==    at 0x8048502: main (ptrErrorBis.c:19)
==4795==  Address 0x41a4050 is 0 bytes after a block of size 40 alloc'd
==4795==    at 0x402425F: calloc (vg_replace_malloc.c:467)
==4795==    by 0x80484A0: main (ptrErrorBis.c:9)
==4795==
tableau[10] = 0
==4795==
==4795== HEAP SUMMARY:
==4795==     in use at exit: 40 bytes in 1 blocks
==4795==   total heap usage: 1 allocs, 0 frees, 40 bytes allocated
==4795==
==4795== Searching for pointers to 1 not-freed blocks
==4795== Checked 56,548 bytes
==4795==
==4795== LEAK SUMMARY:
==4795==    definitely lost: 40 bytes in 1 blocks
```

Il existe des interfaces graphiques aux sorties de `valgrind`, comme `valkyrie`.

5.8 L'outil de mise au point gdb

Il existe plusieurs "debuggers" pour les programmes écrits en Langage C. Nous décrivons ici l'un des plus courants : `gdb`.

`gdb` peut être utilisé dans deux situations, soit après la mort d'un processus en examinant le fichier core généré lors de l'arrêt du processus par le système, soit pour étudier le comportement d'un programme qui se déroule. Nous décrirons principalement l'usage de `gdb` dans ce dernier cas, de loin le plus fréquent et le plus utile.

5.8.1 Appel de gdb

Pour utiliser `gdb` dans les meilleures conditions, il est nécessaire d'avoir réalisé la compilation avec l'option `-g` :

```
gcc -g -o prog prog.c
```

L'option `-g` permet de stocker dans le fichier exécutable des informations symboliques facilitant le travail de `gdb` telles que le nom des variables et les lignes du code source où elles apparaissent. Pour examiner le code exécutable, il suffit alors de taper

```
gdb prog
```

5.8.2 Commandes interactives

Lorsque `gdb` est lancé, il répond en affichant `(gdb)` pour indiquer qu'il est en attente de commandes. Les principales commandes sont les suivantes :

- `run (ou r)`. La commande débute l'exécution du programme. Si des arguments sont à passer, il faut les taper sans rappeler le nom du programme. L'exécution est bien entendu très lente par rapport à l'exécution normale du programme.
- `print [expr]` affiche la valeur de l'expression. L'expression peut comprendre des constantes et des variables du programme. Les opérateurs utilisables sont ceux du langage C, excepté la division entière qui est notée `div`, et le calcul du modulo qui est noté `mod` au lieu de `%`. Les valeurs sont affichées dans le format correspondant au type déclaré de la variable.
- `break`. La commande permet de poser des points d'arrêt. La syntaxe a deux formes :

  `break <numéro de ligne>`
  qui définit un point d'arrêt à la ligne spécifiée du fichier source et
  `break <func>` qui spécifie un arrêt à chaque appel de la fonction `func()`.
- `cont`. L'instruction permet de continuer l'exécution après un arrêt, jusqu'au prochain point d'arrêt ou jusqu'à la fin du programme.
- `step`. Comme pour `cont`, il s'agit de relancer le programme après un arrêt, mais ici, `gdb` exécute seulement l'instruction correspondant à la ligne suivante du programme source. Notez qu'on n'entre pas dans les fonctions appelées.
- `next`. On exécute une instruction, comme pour `step`, mais en entrant dans les fonctions appelées.
- `backtrace`. Cette commande permet d'afficher la pile d'exécution, indiquant à quel endroit l'on se trouve au sein des différents appels de fonctions.
- `delete <n>`. La commande supprime le point d'arrêt placé auparavant et repéré sous le numéro n.
- `where`. Cette commande affiche les noms des fonctions actives.
- `list <n1>, <n2>` liste le fichier source de la ligne n1 à n2.
- `quit (ou q)`. Cette commande abandonne `gdb`.

II – Commandes Linux

1 Notions de base

1.1 Introduction

Linux est un système multi-tâches et multi-utilisateurs. Linux est de plus, un système d'exploitation interactif. Cela signifie que lorsqu'un utilisateur tape une commande, le système vérifie la commande, l'exécute, visualise les réponses appropriées et se remet en attente d'autres commandes. La plupart des fonctions de base sont regroupées dans une partie monolithique appelée noyau (le terme anglais est kernel). Les deux autres parties principales de Linux sont constituées par le Système de Fichiers et une couche de logiciel servant d'interface pour l'utilisateur appelée le **shell**. Le shell ne fait pas partie à proprement parler du système Linux mais c'est ce que voit l'utilisateur quand il travaille sous Linux directement. Le shell a une double fonction : c'est d'une part un interpréteur de commandes, c'est à dire qu'il transcrit les requêtes de l'utilisateur en actions ; c'est d'autre part un véritable langage de programmation permettant de réaliser des applications de gestion du système. On désigne habituellement par le mot script le code généré en langage shell.

Le noyau Linux comprend deux parties : le sous-système de gestion de fichiers et le sous-système de gestion de processus. Il comprend également une section appelée appels système qui se situe à la lisière entre les programmes utilisateurs et le noyau.

1.2 Comptes utilisateur – Super utilisateur

Tout utilisateur désirant se connecter à un système Linux doit posséder un compte utilisateur dans le système. Il s'agit d'une identification choisie par l'administrateur du système. Un compte est caractérisé par :

- un nom de connexion unique dans le système.
- un mot de passe (facultatif mais fortement conseillé)
- un espace de travail sur le disque matérialisé par un point d'entrée dans le système de fichiers et qui est appelé "**home directory**" ou répertoire de travail.

Un système d'exploitation tel que Linux implique par sa complexité, la définition d'un personnage chargé de gérer les ressources de l'installation informatique, que nous appellerons super utilisateur ou "**super-user**". La notion de super-user est matérialisée par l'existence d'un compte particulier sur le système Linux dont le nom est prédéfini et obligatoire. Ce nom est `root`. Il en résulte que le super-user est souvent désigné sous le nom de `root`. Le super-user dispose de privilèges particuliers lui permettant d'avoir une visibilité totale sur l'activité de la machine. Il peut également agir sur la quasi totalité des travaux exécutés par Linux. Il dispose donc d'un ensemble de commandes et de droits spécifiques lui donnant les moyens d'assurer sa tâche. L'utilisation du compte `root` permet l'usage des commandes réservées au super-user, cependant il autorise aussi toutes les possibilités fournies par un compte normal.

Il est également possible d'exécuter une commande en tant que super utilisateur, en faisant précéder la dite commande de la commande `sudo` (abréviation de "substitute user do", en anglais : exécuter en se substituant à l'utilisateur). Par exemple, pour arrêter le système, on peut taper la commande `/sbin/halt` qui demande le mot de passe `root`. Une alternative est :

```
sudo /sbin/halt
```

1.3 Connexion – Déconnexion

Connexion

Lorsqu'on se connecte, on peut accéder aux commandes Linux par le biais d'une fenêtre de commandes ou terminal (souvent le programme `xterm` ou ses dérivés).

Une fois cette fenêtre lancée, le système affiche un sigle qui sera envoyé à chaque fois que Linux sera en attente d'une commande de votre part. Ce sigle qu'on nomme `prompt` est par défaut le caractère `$`. En général, il est configuré pour faire apparaître le nom du compte et éventuellement le nom sur le réseau du système utilisé, ainsi que le répertoire courant (dans lequel on se trouve). Par exemple : `evariste@maMachine:~`
Vous êtes alors reconnu comme un utilisateur valide du système et vous êtes installé dans votre répertoire de travail (home directory) repéré par le symbole `~`.

Quand une connexion a déjà été réalisée avec succès, il est possible de créer une nouvelle connexion avec un autre utilisateur. Cette nouvelle connexion ne supprime pas la précédente. Il faut taper `su` suivi du nouveau nom de login. Dans ce cas, la procédure

de connexion est à nouveau invoquée, et en particulier une demande le mot de passe correspondant au nouveau nom de login. Il faut noter cependant les points suivants :

- Quand la connexion est achevée, Linux renvoie le prompt du premier nom de login (qui reste donc inchangé) et l'utilisateur reste dans le même espace de travail. Si cette situation n'est pas souhaitée, il convient d'utiliser la commande `su -` suivi du nouveau nom de login qui donne un résultat identique à une connexion directe
- Si on tape `su` sans argument, c'est le compte `root` qui est pris par défaut. C'est sous cette forme que ce type de connexion est utilisé le plus souvent, car cela permet au super-user d'intervenir sur n'importe quel terminal sans déconnecter aucun utilisateur.
- Lorsque la requête de déconnexion est formulée, Linux rétablit la connexion précédente dans l'état ou elle était avant la commande `su` (en particulier pour la zone courante de travail).

Déconnexion

La procédure de déconnexion est extrêmement simple, il suffit de taper : `CTRL/D` ou bien la commande `exit`. D'une manière générale en effet, sur Linux `CTRL/D` signifie "fin de saisie". C'est ce qu'il faut taper en particulier lors de la saisie de données au clavier pour matérialiser la fin de l'entrée pour certaines commandes. Dans ce cas, le `CTRL/D` ne déconnecte pas. En revanche, lorsque le système est en attente de commande, le `CTRL/D` provoque la fin d'exécution du shell courant entraînant ainsi la déconnexion de l'utilisateur. La commande `exit` termine plus explicitement le shell.

1.4 Arrêt du système

La procédure d'arrêt a pour objet de stopper tous les travaux en cours d'exécution sur le système et de mettre à jour sur le disque l'état du système de fichiers. Cette procédure a pour nom `/sbin/shutdown`. Il faut disposer des privilèges du super-utilisateur pour pouvoir l'invoquer. Pour arrêter le système correctement, il faut, sauf urgence particulière, s'assurer que tous les utilisateurs sont déconnectés, et qu'aucune tâche n'est encore en cours d'exécution. La procédure d'arrêt est invoquée de la façon suivante :

```
# shutdown -h now
```

Cela signifie qu'une demande d'arrêt immédiat est émise (`now`) suivi d'une procédure de mise hors tension (`-h` pour "halted"). Un argument différent de `now` peut être donné en argument, sous forme temporelle : `+m` où `m` est le nombre de minutes ou `hh:mm` spécifiant l'heure d'arrêt.

2 Notions fondamentales

2.1 Les types de fichiers

Pour le système Linux, tout fichier est une suite d'octets sans aucune hypothèse sur la structuration ou la nature des informations stockées. A l'exception des fichiers exécutables (fichiers contenant le code binaire des commandes) pour lesquels Linux s'attend à une structure prédéfinie, toute organisation interne des données contenues dans un fichier est laissée au choix de l'utilisateur du système, c'est à dire en réalité aux applications. Tout fichier Linux a les attributs suivants :

- un nom (pas nécessairement unique) comprenant 256 caractères au plus.
- un numéro unique de repérage pour le système appelé inode. une taille en octets.
- l'identification du propriétaire (compte utilisateur) et du groupe auquel appartient celui-ci.
- la date de dernière modification des caractéristiques du fichier (dite abusivement date de création).
- la date de dernière modification (des données).
- la date de dernier accès.
- un système de permissions d'accès.

Bien que tous les fichiers aient ces mêmes attributs, la nature des tâches à effectuer sur eux conduit le système Linux à distinguer plusieurs types de fichiers.

2.2 Fichiers répertoires

La fonction d'un répertoire (le terme anglais est directory) est de fournir la correspondance entre la désignation d'un fichier par l'utilisateur, c'est à dire le nom, et la désignation par le système Linux lui-même, c'est à dire un numéro appelé inode. Chaque utilisateur possède un répertoire de ses propres fichiers et peut créer des sous-répertoires. Chaque répertoire contient en outre, deux entrées particulières :

. qui contient le nom du répertoire courant

.. qui contient le nom du répertoire parent.

Le système Linux a besoin pour pouvoir fonctionner d'un certain nombre de répertoires qui seront mentionnés plus loin. Un répertoire est donc un point d'attache de fichiers et/ou de sous-répertoires, ces derniers pouvant être eux mêmes les points d'attache d'autres fichiers et/ou d'autres répertoires, etc. L'ensemble des fichiers est ainsi constitué d'un squelette de répertoires, attachés les uns aux autres selon une arborescence, car un répertoire peut avoir plusieurs fichiers et/ou répertoires attachés à lui, mais ne peut avoir qu'un seul père qui est le répertoire auquel il est attaché et dont le nom est contenu dans le fichier ...

2.3 Fichiers ordinaires

Dans l'arborescence qui vient d'être évoquée, si les répertoires forment les branches de l'arbre, les fichiers ordinaires constituent ses feuilles. L'utilisateur manipule essentiellement des fichiers ordinaires. Nous distinguerons les fichiers ordinaires selon leur utilisation :

- les fichiers *textes* qui sont composés de chaînes de caractères sans signification particulière pour le système Linux. Le texte de ce cours par exemple constituerait un fichier de type texte.
- les fichiers *programmes* ou *sources* sont des séquences de mots qui correspondent à des instructions pour l'ordinateur. De façon générale, ces fichiers sont créés par un éditeur de texte. Les instructions peuvent être directement exécutables par la machine, il s'agit alors de langages interprétés (par exemple, programmes `python`, procédures shell), ou doivent d'abord être traduites en codes exécutables, il s'agit alors de langages compilés (par exemple, codes `C`, `C++`, etc.).
- les fichiers *exécutables* résultent de la traduction par un compilateur approprié de fichiers sources et de leur chargement. Ces fichiers ont une structure qui est reconnue par Linux, notamment au niveau de l'en-tête et de la fin du fichier.
- les fichiers de *données* contiennent des informations exploitables par les programmes (ex : les fichiers créés par les systèmes de gestion de base de données). Ces fichiers se séparent en deux groupes :
  - les fichiers *binaires* où les nombres sont écrits selon leur représentation binaire.
  - les fichiers *ASCII* où les nombres sont écrits sous forme de chaînes alphanumériques.

2.4 Fichiers spéciaux

Les fichiers spéciaux sont de plusieurs types, on trouvera ci-dessous les types les plus importants pour l'utilisateur :

Fichiers spéciaux d'entrée/sorties

La gestion des entrées/sorties a été conçue dans le système Linux de telle sorte que soit abolie toute différence entre fichiers disque et unités périphériques. Ces fichiers spéciaux sont des noms de fichiers disque servant à désigner soit :

- des unités périphériques (comme par exemple les terminaux).
- des lignes de communication.
- des tubes de communication.
- des organes internes tels que la mémoire principale par exemple.

Ces fichiers spéciaux sont traités comme des fichiers disque, mais les opérations de lecture et d'écriture effectuées sur ces fichiers permettent d'activer le dispositif qui leur est associé.

Fichiers liens

Les fichiers liens contiennent un nom d'un fichier de type quelconque vers lequel le fichier lien renvoie. Lors de l'utilisation d'un lien, le système trouve le nom du fichier pointé par le lien et utilise en fait ce fichier pointé. Le lien permet donc d'accéder de façon transparente pour l'utilisateur à un fichier ou un répertoire de plusieurs points placées de façon quelconque dans l'arborescence des fichiers.

2.5 Le système de fichiers

Tous les fichiers manipulables sous Linux sont regroupés dans une structure hiérarchique et cohérente qu'on désigne par un *système de fichiers*. Les caractéristiques principales d'un système de fichiers Linux sont les suivantes :

- La taille maximale d'un fichier donné peut aller jusqu'à 2 à 16Go (2Go sur le système de fichiers `ext2`, 16Go sur `ext4`).
- Il n'y a pas de structuration (liée à Linux) du contenu des fichiers, en particulier, il n'y a pas de marque physique de fin de fichier. C'est la taille en octets qui permet de déterminer la fin de fichier.
- Les fichiers disque et les unités périphériques sont interfacées de façon identique.
- Un même fichier physique peut avoir plusieurs références utilisateur (c'est à dire être désigné par des noms différents). Ainsi plusieurs programmes d'application par exemple, peuvent utiliser un même fichier de données qui sera repéré avec des noms différents et des localisations dans l'arborescence différentes.
- le système de fichiers principal a une origine (répertoire) dont le nom est fixé, c'est le répertoire `root`. Comme ce nom est impératif, il est systématiquement omis dans les désignations de fichiers. Il en résulte que pour le désigner, on utilise le seul symbole `/`.
- Dans l'arborescence qui va du global au particulier, un même nom de fichier peut être utilisé à divers niveaux du système de fichiers. En revanche, le chemin qui permet d'accéder au travers de différents répertoires et sous-répertoires à un fichier donné doit être unique.

Les systèmes de fichiers les plus courants sont la `fat` (disquettes et clefs usb), `ntfs` (Windows), `Ext2`, `Ext3` et `Ext4` (Linux), `iso 9660` (cd) et `udf` (dvd).

Il est possible d'avoir plusieurs systèmes de fichiers simultanément. Ces différents systèmes peuvent être réunis ensembles pour ne former qu'une seule entité. Un système de fichiers peut être monté par une commande adéquate (la commande `mount` ; cf. sous section 1.6, p. 116), c'est à dire attaché à un répertoire de l'arborescence initiale. De même, il est possible de démonter un système de fichiers (par la commande `umount`). En d'autres termes, on peut faire apparaître ou disparaître tout un ensemble de fichiers, si cet ensemble est structuré en un système de fichiers.

Le système de fichiers dont l'origine est root (et dont le nom, rappelons le est omis, de sorte qu'il apparaît comme `/`) contient les répertoires et fichiers nécessaires au fonctionnement de Linux. Ce système de fichiers est donc obligatoire.

Chaque système de fichiers est supporté physiquement par une partition d'un disque dur. Les systèmes de fichiers ne peuvent pas utiliser de l'espace disque à l'extérieur de cette partition. Considérons le répertoire `/media`, contenant le sous répertoire `apt`, comme représenté en Figure II.1 en sortie de la commande `tree` qui affiche les répertoires en forme graphique primitive.

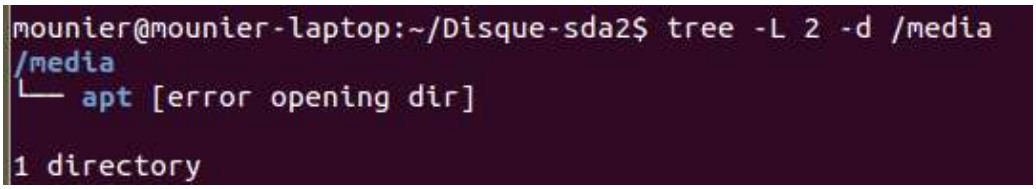

FIGURE II.1 – Exemple de répertoire `/media` avant montage.

Après montage du système de fichiers `Recovery partition`, on obtient la situation de la Figure II.2

FIGURE II.2 – Le répertoire `/media` après montage.

Dans l'exemple de cette figure, le répertoire qui sert de point d'attache du nouveau système de fichiers est `/media`. Ce répertoire est dit être le point de montage. En dessous de `media`, les fichiers et répertoires sont placés sur ce nouveau système de fichiers, c'est à dire sur une partition séparée d'un des disques durs du système.

2.6 Accès et partage des fichiers

Notion de propriété

Pour le système Linux, le créateur d'un fichier devient automatiquement le propriétaire du fichier. L'identité du propriétaire est écrite dans la structure de contrôle du fichier sous la forme d'un nombre. Ce nombre est celui que Linux associe au nom de connexion pour repérer un utilisateur. Le propriétaire d'un fichier peut transmettre à n'importe quel utilisateur son droit de propriété par la commande `chown`. Seuls le propriétaire et le super-user peuvent effectuer ce transfert de propriété.

Accès au fichiers

Chaque fichier Linux (répertoire, fichier ordinaire ou fichier spécial) possède un mécanisme de protection définissant les conditions d'accès aux informations contenues dans le fichier. Sur le système Linux, l'accès à un fichier est conditionné par un jeu de permissions spécifiques au fichier. Cependant dès lors que les conditions d'accès sont satisfaites, tout utilisateur peut accéder aux données contenues dans un fichier. En particulier, il faut noter que :

- Le propriétaire du fichier n'a aucune priorité d'accès (à permissions égales).
- Plusieurs utilisateurs, en nombre a priori illimité, peuvent accéder simultanément à un fichier donné. Pendant longtemps même, le système Linux n'a fourni aucune possibilité d'avoir un accès exclusif à un fichier (excepté par le jeu des permissions). Il est possible aujourd'hui par programme de bloquer l'accès à tout ou partie d'un fichier.
- Le super-user a un accès sans restriction à tout fichier et à tout répertoire. Aucune interdiction ne peut lui être opposée, y compris celles qu'il définit lui-même sur ses propres fichiers.

Droits d'accès

Le système Linux distingue trois types de permissions d'accès :

`r` (read)	lecture autorisée. Permet de lire un fichier, ou de consulter le contenu d'un répertoire.
`w` (write)	écriture autorisée. Permet de modifier le contenu d'un fichier. Dans le cas d'un répertoire cela représente le droit de créer et de détruire des fichiers dans ce répertoire.

`x` (execute) exécution autorisée. Droit permettant de lancer les fichiers exécutables. Dans le cas d'un répertoire cela donne le droit de rechercher un fichier dont on connaît le nom. En d'autres termes, le répertoire peut être utilisé dans un chemin d'accès seulement si ce droit est présent. Ce droit est indépendant du droit de lecture. En particulier si l'exécution est autorisé mais pas la lecture, on peut consulter un fichier dont on connait le nom mais on ne peut pas lire les noms des autres fichiers du répertoire.

Ces permissions sont associées à chacune des trois classes d'utilisateurs :

`u` (user) propriétaire du fichier
`g` (group) groupe auquel appartient le propriétaire (propriétaire excepté)
`o` (other) le public (tous les autres utilisateurs)

Au total, il y a donc un jeu de 9 permissions binaires (oui ou non) déterminant les droits d'accès à chaque fichier et ce, pour tous les utilisateurs possibles.

Note II.1. Les permissions présentées ici sont les plus importantes d'un point de vue utilisateur. Mais le système Linux manipule des droits supplémentaires plus utiles à l'administration. Ces droits sont expliqués au chapitre d'administration système, sous section 1.3, p.114.

Nommage des fichiers

Le nom d'un fichier est une chaîne de 256 caractères au plus. Il n'y a pas de caractère interdit par le système Linux lui-même. Cependant, les caractères non imprimables sont évidemment déconseillés pour des raisons de bon sens, de plus, certains caractères ont une signification particulière pour le shell ce qui conduit à proscrire leur emploi. Ces caractères sont les suivants :

```
/ * ? - $ [ ] { } ( ) | ; & " ’ ‘ espace tabulation return
```

Note II.2. Le shell possède des mécanismes d'escape des caractères spéciaux qui permettent d'accéder à des fichiers contenant des caractères spéciaux. D'autre part, il faut veiller sur les claviers français à ne pas utiliser de lettres accentuées qui peuvent également être mal interprétées.

En résumé, il est conseillé pour les noms de fichiers de n'utiliser que les lettres non accentuées, les chiffres, le point et le signe souligné (_). Bien que cela ne soit pas une obligation, il est d'un bon usage de commencer un nom de fichier par une lettre et de n'utiliser que des minuscules, les majuscules étant plutôt réservées aux noms de variables.

Exemples de noms corrects :

```
fichier        f10        fic_3.c
```

Exemples de noms incorrects ou déconseillés :

```
%fic-*        $true        papa/maman        quoi?        251
```

Repérage dans le système de fichiers

L'identification d'un fichier dans le système Linux est donnée par le chemin d'accès. Le chemin d'accès (en anglais pathname) est une suite de noms de répertoires séparés par le caractère / et terminée par le nom du fichier à identifier (fichier ordinaire, répertoire ou fichier spécial). Il n'y a pas de limite dans le nombre de répertoires traversés pour arriver au nom de fichier terminal. Il est possible de repérer un fichier par rapport à l'origine du système de fichiers ou par rapport à une origine courante définie préalablement. Le chemin d'accès absolu est la suite des répertoires qu'il faut franchir en partant de la racine (root) du système de fichiers pour aboutir au fichier désiré. Il est dit absolu car il désigne explicitement le fichier choisi indépendamment de la position où l'on peut se trouver à un instant donné. Puisque le nom `root` est systématiquement omis, le chemin absolu commence toujours par le caractère /. Exemple de désignation d'un fichier par le chemin absolu :

```
/user/dir1/dir2/fichier
```

Le chemin d'accès relatif est la suite de noms de répertoires qu'il faut franchir à partir du répertoire où l'on se trouve, qui est appelé répertoire courant, pour arriver au fichier désiré. Par suite, le chemin relatif ne commence jamais par le caractère /. Linux considère que la désignation d'un fichier est faite par le chemin absolu ou par le chemin relatif sur le seul critère que la chaîne de caractères commence par / ou non. Dans la désignation par chemin relatif, il est possible d'utiliser les deux répertoires suivants :

- `.` repérage du répertoire courant.
- `..` repérage du répertoire parent.

Dans ces conditions, on peut avoir 3 types de chemins relatifs :

- Le chemin commence implicitement dans le répertoire contenant le répertoire `dir1` :
  ```
  dir1/dir2/fichier
  ```
- Le chemin commence explicitement dans le répertoire contenant `dir1` :
  ```
  ./dir1/dir2/fichier
  ```
- Le chemin commence dans le répertoire parent du répertoire où l'on se trouve :
  ```
  ../dir_frere/fichier
  ```

Pour construire des fichiers de commande, il existe deux commandes d'extraction de nom dans le chemin d'accès :

`basename` Cette commande extrait du chemin d'accès le nom de fichier après le dernier /.

`dirname` Cette commande est inverse de la précédente, c'est à dire qu'elle enlève du chemin d'accès le nom de fichier après le dernier /.

Exemples :

```
basename /other/evariste/jeu  ---> jeu
dirname /other/evariste/jeu   ---> /other/gilles
```

Notion de modèle de fichier

De manière à faciliter la manipulation des noms dans le système de fichiers, et surtout pour faciliter les recherches, il existe quelques symboles qui permettent de créer un modèle, le shell générant alors automatiquement tous les noms correspondant à ce modèle.

`*` ce caractère représente un nombre quelconque de caractères (y compris l'absence de caractère).

`[c1c2cn]` permet la substitution par `c1`, puis `c2`, puis `cn` du caractère correspondant à cette position.

`[!c1c2cn]` remplace n'importe quel caractère, sauf ceux placés dans la liste.

`[c1-c2]` permet la substitution par tous les caractères de la table ASCII de `c1` à `c2` inclus du caractère correspondant à cette position.

`?` permet la substitution par tous les caractères possibles du caractère courant.

Ce mécanisme de substitution est appelé recherche à partir de modèles (en anglais "pattern matching").

Exemple II.1.

Supposons que notre répertoire contienne les fichiers suivants :

```
chap1,   chap2,   chap1.1, chap1.2, chap1.3
chap2.1, chap2.2, chap2.3
```

Pour imprimer les paragraphes 3 de chaque chapitre, nous pouvons utiliser la commande

```
lp chap1.3 chap2.3
```

mais en utilisant la notion de modèle on tapera :

```
lp chap?.3
```

Si maintenant nous voulons uniquement les paragraphes 1 et 3, nous pouvons demander :

```
lp chap1.1 chap1.3 chap2.1 chap2.3
```

ou bien

```
lp chap?.[1,3]
```

l'impression de tous les fichiers se fera par

```
lp chap*
```

Ce mécanisme fonctionne avec toutes les commandes de recherche et de manipulation de fichiers (attention aux suppressions globales de fichiers !) ainsi que dans certains utilitaires tels les éditeurs de texte. On peut également l'utiliser avec un chemin d'accès, par exemple :

```
lp /tmp/liste*
```

Note II.3. Le caractère . (point) en début de nom de fichier n'est pas trouvé par un modèle de fichier (voir la commande ls et les fichiers cachés). Par exemple pour imprimer tous les fichiers d'un répertoire, y compris ceux commençant par un point il faut taper :

```
lp .* *
```

2.7 Répertoires de travail du système

La norme sur la hiérarchie des systèmes de fichiers (FHS ou Filesystems Hierarchy Standard) définit une organisation logique standard concernant l ?organisation des répertoires. La plupart des systèmes Linux y adhèrent. En voici le premier niveau d'arborescence :

Répertoire	Signification	Contenu
`/`		Racine du système, hiérarchie primaire
`/bin`	"binaries"	Exécutables des commandes essentielles
`/boot`	démarrage	Fichiers statiques du chargeur d ?amorçage
`/dev`	"device"	Fichiers spéciaux des périphériques
`/etc`	"editing text config"	Fichiers textes de configuration
`/home`	maison	Répertoires personnels des utilisateurs
`/lib`	"librairies"	Bibliothèques partagées essentielles et modules du noyau
`/media`		Contient les points de montages pour les médias amovibles
`/mnt`	"mount"	Point de montage pour monter temporairement un système de fichiers
`/proc`	processus	Répertoire virtuel pour les informations système (états du noyau et des processus système)
`/root`	racine	Répertoire personnel du super-utilisateur
`/sbin`	"super binaries"	Exécutables système essentiels
`/srv`	"services"	Données pour les services du système
`/tmp`	"temporary"	Fichiers temporaires
`/usr`	"Linux system resources"	Hiérarchie secondaire, pour des données en lecture seule par les utilisateurs. Ce répertoire contient les applications usuelles des utilisateurs et leurs fichiers
`/usr/local`		Hiérarchie tertiaire. Emplacement où les utilisateurs doivent installer les applications qu'ils compilent.

`/var`	“variable”	Données variables et diverses
`/opt`	“optionnels”	Emplacement pour des applications installées hors gestionnaire de paquets (logiciels optionnels)

3 Commandes

3.1 Principes généraux

Les commandes Linux sont utilisées pour lancer l'exécution d'un programme utilisateur ou d'un programme système. Chaque commande est lue caractère par caractère par un interpréteur, que nous avons vu être le shell et qui détermine les actions à exécuter. shell recherche le programme correspondant à la tâche à effectuer et lance son exécution. La ligne de commande n'est prise en compte qu'au moment où la touche `<return>` est appuyée. Plusieurs commandes peuvent être entrées sur la même ligne. Dans ce cas elles doivent être séparées par des *points virgules* (`;`). La syntaxe générale d'une commande est la suivante :

```
cmd [-options] [arguments ou noms de fichiers]
```

Nota

les crochets `[ ]` indiquent que l'argument est optionnel.

- Par convention, les noms de commandes (cmd) sont tapés en minuscules.
- Les options sont reconnues comme telles si elles sont précédées du signe moins (-) ou plus (+) dans certains cas. Ces options sont définies par une lettre minuscule ou majuscule suivant les cas.
- Les arguments sont des noms de fichiers ou des variables prédéfinies ou définies par l'utilisateur (voir le shell plus loin).

Les quelques commandes de base qui suivent illustrent les diverses situations qui peuvent se produire.

3.2 Le code de retour

Lorsqu'une commande se termine, elle positionne un nombre codé sur un octet. Cet octet est le code de retour de la commande. Si la commande s'est déroulée sans erreur, le code de retour est fixé par convention à 0. Dans le cas contraire il est différent de 0. L'interpréteur de commande shell récupère ce code de retour, ce qui permet à l'utilisateur de tester éventuellement sa valeur.

3.3 Quelques exemples de commandes

3.4 Commande : `echo`

Cette commande affiche sur l'écran le texte passé en argument.

Syntaxe

```
echo [-n] [texte]
```

Description

Affiche à l'écran le texte indiqué.
L'option `-n` annule le passage à la ligne automatique en fin de texte

3.4.1 Commande `ls`

Cette commande permet de lister à l'écran le contenu du répertoire dans lequel vous vous trouvez (plus de détails sont donnés en sous-section **??**, p. **??**)

Syntaxe

```
ls [-option] [rep1 rep2 ...]
```

Description

Liste à l'écran le contenu des répertoires `rep1`, `rep2`, fournis en arguments 2, 3, etc. En tapant la commande avec l'option `l` (long) qui demande des informations détaillées, soit

```
ls -l
```

on obtient un affichage du type :

```
total 2
-rw-r-----    1    stage1    users    449    Dec 19 10:10    truc
drwxr-xr-x    2    stage1    users    512    Dec 19 10:19    dir
```

3.4.2 Commande `cat`

Cette commande permet de lister et éventuellement de concaténer, le contenu d'un ensemble de fichiers à l'écran. Elle est détaillée dans la section 4.

3.5 Les entrées/sorties des commandes

Les appels d'entrées/sorties ont été conçus de manière à éliminer les différences d'accès existant entre les diverses unités périphériques.

Notion de périphérique standard

On appelle périphérique standard, tout dispositif que l'utilisateur emploie pour communiquer avec le système et qui lui est attribué sans spécification particulière de sa part. Les exemples typiques sont l'écran et le clavier.

La plupart des commandes reçoivent leur entrée de l'entrée standard et envoient leur sortie sur la sortie standard. Sur le système Linux, il y a 3 périphériques standard attribués à chaque utilisateur, ce sont :

– l'entrée standard (`stdin`) qui correspond au clavier

– la sortie standard (`stdout`) qui correspond à l'écran
– la sortie d'erreurs standard (`stderr`) qui correspond aussi à l'écran (bien que ce soit un fichier différent de la sortie standard). Toutes les erreurs sont affichées sur cette sortie.

Redirections simples

Linux donne la possibilité à l'utilisateur de modifier les affectations par défaut des organes d'entrées/sorties standard. Cette opération s'appelle rediriger les entrées sorties standard. Divers aiguillages sont possibles. Les informations peuvent être prises ou rangées à l'aide d'un fichier spécial (associé à un périphérique) en sortie ou en entrée d'un autre programme. Ceci est permis par le fait que les fichiers Linux ne sont pas figés suivant un critère d'accès (séquentiel, direct), ni selon une allocation d'espace (taille des fichiers).

Pratiquement, il existe trois symboles de redirection des entrées/sorties correspondant aux divers objectifs de l'aiguillage :

`<` : prendre l'entrée dans

`>` : mettre la sortie au début de

`>>` : mettre la sortie à la fin de

L'argument qui suit l'un de ces trois symboles est un nom de fichier (fichier ordinaire disque ou fichier spécial). La syntaxe pour opérer une redirection a donc l'allure suivante dans sa version la plus simple :

```
commande > nom_fichier
```

Note II.4. – Il peut y avoir une ou plusieurs redirections dans la même commande. Comme chaque commande dispose de 3 entrées/sorties, on peut faire 3 redirections au maximum
– les redirections sont liées à la commande elle-même, c'est à dire que les entrées/sorties standard sont restaurées automatiquement dans leur destination par défaut dès que la commande est achevée.
– en sortie, si le fichier de redirection existe, il est pris et donc, réinitialisé si on a utilisé l'option `>`. Si le fichier n'existe pas, il est automatiquement créé même avec l'option `>>`.
– la redirection en sortie présente donc un risque d'écrasement des fichiers existants, on peut éviter cela en positionnant une option du shell de la façon suivante :
`set -o noclobber`

Exemples II.1.

(*i*) La liste de fichiers du répertoire est placée dans le fichier fich :
```
ls -l > fich
```
(*ii*) Le fichier `fic2` est copié dans le fichier `fic3` à partir de la fin de ce dernier :
```
cat fic2 >> fic3
```

(*iii*) Le programme `a.out` prendra les données qu'il aurait normalement lues sur le clavier, dans le fichier `fichdata`.

```
a.out < fichdata
```

La redirection de l'entrée et de la sortie standard se fait implicitement sans préciser explicitement en particulier s'il s'agit de l'entrée ou de la sortie. Il est également possible de *rediriger la sortie d'erreurs standard*. Dans ce cas il est évidemment nécessaire de la préciser explicitement puisqu'elle coïncide avec la sortie standard. La désignation est par convention la suivante :

entrée standard	`0`
sortie standard	`1`
sortie d'erreurs standard	`2`

Pour rediriger la sortie standard on tapera donc :

```
commande 2> fich_erreur
```

Si on désire rediriger simultanément la sortie d'erreurs et la sortie standard dans le même fichier, la syntaxe est la suivante :

```
commande >fich 2>&1
```

Il est également possible (moins utile) de rediriger la sortie standard vers la sortie d'erreur par la commande : `1>&2`.

Note II.5.
- Le nombre qui désigne l'organe standard doit être collé au symbole de redirection sinon le shell l'interprète comme un argument de la commande.
- Les redirections peuvent apparaître n'importe où dans la ligne, la désignation de la redirection est faite uniquement par les caractères $>$ et $<$ et non par la position, ainsi par exemple :

```
<entrée >sortie sort
```

Note II.6. La commande sort utilisée sans argument lit l'entrée standard, trie les lignes lues par ordre alphabétique, puis affiche le résultat sur la sortie standard. Le fichier sortie contient donc le résultat du tri du fichier entrée.

- Les redirections sont évaluées de la gauche vers la droite, ce point a une importance lors de la redirection de la sortie d'erreur. Pour rediriger la sortie d'erreur dans la sortie standard il faut d'abord rediriger cette dernière.
- Une commande contenant les redirections suivantes a donc pour effet de rediriger la sortie d'erreur standard dans la sortie standard en l'occurrence 1 puisque la redirection vers fichier n'a pas été lue. La sortie standard est ensuite envoyé vers fichier, mais les erreurs arriveront toujours à l'écran :

```
2>&1 1>fichier
```

Enchaînement de commandes : tubes

La sortie standard d'une commande peut servir d'entrée standard à une autre commande. Pour cela, le système ouvre et gère un fichier de type FIFO (first in first out) qui servira de tampon de communication entre les deux commandes. C'est ce type de connexion qu'on appelle *tube* (en anglais pipe). Physiquement, le tube est représenté par le symbole `|` qui signifie créer un tube entre les deux commandes de part et d'autre du `|` (i.e. relier la sortie standard de la première commande à l'entrée standard de la seconde).

La syntaxe est la suivante :

```
commande1 | commande2
```

On peut ainsi chaîner plusieurs commandes avec le signe `|`, les résultats intermédiaires n'étant plus affichés au terminal. Seul le résultat final est visualisé, sauf s'il est lui même redirigé.

Exemple II.2.

> Trier à l'écran alphabétiquement ligne à ligne un fichier. On tapera la commande :
> `cat fichier | sort`
>
> `cat` est la commande d'impression
> `fichier` est le fichier à trier
> `sort` est la commande de tri
>
> Le résultat s'affichera directement sur la sortie standard (l'écran). C'est bien entendu, cette sortie qui est triée, le fichier reste, lui, inchangé.

4 Commandes de manipulation de fichiers

4.1 Commande `man`

Le système Linux fournit en ligne une documentation précise sur toutes les commandes reconnues par le système avec les explications des diverses options possibles. On accède à cette documentation en ligne grâce à la commande `man`. ***Ne faites confiance qu'au manuel issu de la commande*** `man`. En effet, chaque distribution Linux est différente et seule `man` vous fournira les informations propres à la distribution sur laquelle vous êtes en train de travailler.

Syntaxe

```
man [-options] [section] cmd
```

Description

`man` affiche à l'écran la nature de la commande `cmd` et précise les éventuelles options avec leur signification. Il est important de noter que lorsque les symboles sont soulignés dans l'affichage, cela signifie que ce sont des variables. Les options de la commande man varient d'un système à l'autre, on consultera le manuel de la machine utilisée pour l'utilisation approfondie de la commande. L'option `section` indique la section du manuel au sein de laquelle on cherche la description de la commande (ou fonction, fichier, etc.) `cmd`. Les différentes sections du manuel sur une distribution Ubuntu sont les suivantes :

Section	Description
1	Programmes exécutables ou commandes de l'interpréteur de commandes (shell)
2	Appels système (fonctions fournies par le noyau)
3	Appels de bibliothèque (fonctions fournies par les bibliothèques des programmes)
4	Fichiers spéciaux (situés généralement dans `/dev`)
5	Formats des fichiers et conventions. Par exemple `/etc/passwd`
6	Jeux
7	Divers (y compris les macropaquets et les conventions)
8	Commandes de gestion du système (généralement réservées au superutilisateur)
9	Sous-programmes du noyau (hors standard)

Exemples

```
man man
```

Explique l'usage de la commande man.

```
man kill
```

Explique l'usage de la commande `kill` (commande qui permet de tuer un processus).

```
man 2 kill
```

Explique l'usage de l'appel système `kill()` (fonction C qui permet de tuer un processus).

4.2 Accès à un répertoire : commande cd

Cette commande a pour but de se déplacer dans l'arborescence et de définir le nouveau répertoire courant ; `cd` signifie "Change Directory".

Syntaxe

```
cd [répertoire]
```

Description

`cd` permet de changer de répertoire et d'avoir comme nouveau répertoire celui qui est spécifié en argument de la commande. Si ce paramètre est omis, le shell considère que c'est le répertoire de travail de l'utilisateur ("home directory"). Si on utilise `".."` comme paramètre, le shell l'interprète comme étant le répertoire de niveau immédiatement supérieur au répertoire courant. Le répertoire de destination peut être précisé de façon absolue (à partir de la racine du système de fichiers), ou relative (à partir du répertoire courant).

Exemples

Chemin absolu :

```
cd /usr/stage1/babar
```

chemin relatif à partir d'un répertoire de même niveau que `babar` :

```
cd ../babar
```

Notations spéciales

On dispose de l'argument `"-"` qui permet de revenir au répertoire précédant le dernier changement.

Exemple

`cd -` permet le retour au répertoire précédent.

Si le répertoire débute par un tilde, les substitutions suivantes ont lieu :

`~/` est remplacé par le répertoire courant de l'utilisateur

`~compte/` est remplacé par le répertoire courant de l'utilisateur compte.

Nota

L'usage du tilde n'est pas limité à la commande cd, on peut utiliser cette notation pour tout argument d'une commande.

Exemples

`cd /rep` est équivalent à `cd /home/stage1/rep` si `/home/stage1` est le répertoire de connexion de stage1. `cat stage2/essai` liste le fichier essai placé dans le répertoire de connexion de l'utilisateur `stage2`.

4.3 Commande `pwd`

Cette commande imprime le nom du répertoire courant ; `pwd` signifie "Print Working Directory".

Syntaxe

```
pwd
```

Description

Avant de lancer les opérations de destruction de fichiers surtout lorsqu'elles sont globales (du type `rm *`), il est fortement conseillé de s'assurer du répertoire l'où on se trouve ... qui n'est peut être pas celui où l'on croit être !

4.4 Création d'un répertoire : commande `mkdir`

Cette commande créé un répertoire ; `mkdir` signifie “MaKe DIRectory”.

Syntaxe

```
mkdir répertoires
```

Description

`mkdir` créé le ou les sous-répertoires spécifiés si l'utilisateur possède le droit d'écrire dans le répertoire d'accueil. Les entrées `"."` et `".."` du nouveau répertoire sont automatiquement créées. Si aucun chemin n'est spécifié, le sous-répertoire est créé dans le répertoire courant.

Exemple

Si on se trouve dans le répertoire `/home/evariste/papa`, `mkdir fiston` va créer le répertoire `/home/evariste/papa/fiston`.

4.5 Destruction d'un répertoire : commande `rmdir`

La destruction comme la création d'un répertoire ne peut être réalisée que par le système lui-même. La commande `rmdir` est la commande de destruction ; `rmdir` signifie “ReMove DIRectory”.

Syntaxe

```
rmdir reps
```

Description

Pour que la suppression du répertoire puisse être exécutée, le répertoire doit être vide. Il est impossible de détruire un répertoire dans lequel on se trouve, non plus qu'un répertoire en amont. Si le chemin n'est pas précisé, le répertoire à détruire sera supposé avoir pour père celui dans lequel on se trouve.

4.6 Visualisation du contenu d'un répertoire : commande `ls`

Cette commande liste le contenu d'un répertoire ; `ls` signifie “LiSt”.

Syntaxe

```
ls [-options] [noms]
```

Description

Pour chaque nom de répertoire, `ls` en liste le contenu. Si le nom est celui d'un fichier ordinaire, `ls` liste les informations relatives au fichier en accord avec les options demandées. Si aucun nom n'est passé à la commande, c'est le répertoire courant qui est listé. Enfin, si des noms de fichiers ordinaires et des noms de répertoires sont passés en argument, `ls` imprime d'abord les informations relatives aux fichiers ordinaires, puis liste le contenu des répertoires.

Options

Il y a une quantité d'options. En standard, `ls` donne seulement le nom du fichier, avec un classement alphabétique. Les options les plus utilisées sont :

- `-a` Liste tous les fichiers. Normalement les noms de fichiers commençant par un point (.) ne sont pas listés.
- `-C` Liste seulement les noms avec une impression multi-colonnes (option par défaut).
- `-d` Liste les répertoires comme les fichiers
- `-i` Affiche le numéro d'inode en début de chaque ligne pour chaque fichier.
- `-L` Suit les liens au lieu de les afficher.
- `-l` Donne les informations les plus complètes.
- `-R` Descend récursivement dans les répertoires.
- `-r` Ordre alphabétique inverse.
- `-s` Affiche seulement le nom et la taille de chaque fichier.
- `-t` Classe selon la date de dernière modification du fichier (la plus récente en premier) au lieu d'un classement alphabétique.
- `-u` Classe selon la date de dernier accès au fichier.

Exemple

Si on suppose que le répertoire courant posséde deux entrées : `fich` (un fichier ordinaire) et `dir` (un répertoire), l'exécution de `ls -l` donnera :

```
drwxr-x--  2   evariste  group  272  Apr 5 14:33  dir
-rw-r--r-- 1   evariste  group  403  Apr 1 13:26  fich
```

De la gauche vers la droite les informations visualisées sont :

- les permissions d'accès,
- le nombre de liens,
- le propriétaire de l'entrée,
- le groupe du propriétaire,
- la taille en octets de l'entrée,
- la date de dernière modification de l'entrée,
- le nom de l'entrée.

4.7 Visualisation : commande `cat`

Outre les éditeurs classiques (`nano`, `gedit`, `vi`, `emacs`), on dispose de plusieurs commandes permettant de visualiser un fichier ASCII sous des formes diverses. Les principales sont `cat, more et od`.

Syntaxe

```
cat [-options] [fich1 fich2...]
```

Description

La commande lit les fichiers spécifiés et affiche le contenu à l'écran. Si aucun fichier n'est spécifié c'est l'entrée standard qui est prise.

Options

`-u` La sortie est sans tampon (la sortie est normalement faite à travers un tampon mémoire ("buffer" en anglais)).

`-s` Élimine les messages relatifs aux fichiers non existants.

`-v` Imprime les caractères non imprimables (exceptés les tabulations, les sauts de page et les retours à la ligne) avec les conventions suivantes :

- imprime les caractères de contrôle comme `^x` (`ctrl/x`)
- imprime `DEL` (la touche `Suppr` ou `Delete`) comme `^?`
- imprime les caractères non ASCII (sur 8 bits) comme `M-x` où x est le caractère spécifié par les 7 bits de poids faible.

`-t` imprime les caractères de tabulation comme `^I` (effectif seulement si l'option `v` est positionnée).

`-e` imprime un `$` à la fin de chaque ligne (avec `-v`)

4.8 Commande `more`

Cette commande visualise un fichier paginé à l'écran.

Syntaxe

```
more [-ns] [+ligne] [+/modèle] [f1 f2..]
```

Description

More permet de lister un fichier en s'arrêtant à chaque page. L'absence de fichier permet d'utiliser la commande derrière un tube pour afficher page par page le résultat de n'importe quelle commande. Pour obtenir la page suivante, il faut presser la touche `<espace>`. La touche `b` permet de revenir en arrière d'une page (sauf pour l'utilisation sur entrée standard). En appuyant sur la touche `q`, l'utilisateur peut terminer l'affichage. Enfin, en tapant `<return>`, more avance d'une ligne. La liste de toutes les commandes est donné par la touche `?` ou bien `h`.

Options

`+ligne` Le listing commence à la ligne spécifiée par son numéro.

`+/modèle` Le listing commence 2 lignes avant la première occurence de la chaîne de caractères `modèle`.

`-n` définit le nombre `n` de lignes par page (22 par défaut).

`-s` supprime les lignes blanches lors de l'impression.

La commande more vérifie que les fichiers spécifiés sont bien visualisables (fichiers ASCII), ce qui n'est pas le cas de `cat`.

4.9 Commande od

Cette commande a pour but d'afficher le contenu d'un fichier sous forme octale, hexadécimale, ou ASCII selon l'option choisie (le nom de la commande est l'abréviation anglaise de "octal dump").

Syntaxe

```
od -bcdox [fich] offset.b
```

Description

`od` imprime octet par octet ou mot de 16 bits par mot de 16 bits le contenu du fichier `fich` à partir d'une origine fixée par `offset.b`. Si la variable `offset` est présente, elle indique que l'impression ne commence qu'à partir de l'octet de rang égal à `offset` exprimé en octal. Si on désire que `offset` soit interprété en décimal, il faut le faire suivre du point. Enfin, si on veut que la valeur `offset` soit interprétée en blocs de 512 octets, il faut la faire suivre de `.b`.

Options

Les options déterminent le type d'affichage. L'option par défaut est l'option `-o`. Plusieurs options sont possibles, dans ce cas les diverses traductions sont données en parallèle.

- `-b` affichage octet par octet, chaque octet étant interprété en numération octale.
- `-c` idem à l'option `-b` mais l'interprétation est faite en ASCII. Certains caractères non imprimables apparaissent comme des "escape", les autres comme des nombres octaux sur 3 caractères.
- `-d` affichage par mots de 16 bits qui sont interprétés en numération décimale.
- `-o` idem à l'option `-d` mais avec une interprétation octale.
- `-x` idem mais avec une interprétation hexadécimale.

Exemple

```
od -c /etc/passwd 18
```

fournit l'affichage caractère par caractère du fichier `/etc/passwd` à partir du 18ème octet.

4.10 Déplacement, changement de nom : commande mv

Cette commande renomme des fichiers ou des répertoires. Elle peut également déplacer des fichiers d'un répertoire à un autre. `mv` signifie "MoVe".

Syntaxe

```
mv [-i] ancien_nom nouveau_nom
mv [-i] fich1 fich2 fich3.... répertoire
```

Description

Dans la première syntaxe, `mv` change le nom du fichier qui s'appelait `ancien_nom` et qui s'appelle désormais `nouveau_nom`. Si le fichier `nouveau_nom` existait déjà, il sera supprimé et remplacé par le fichier `ancien_nom` sous sa nouvelle appellation `nouveau_nom`. Dans la deuxième syntaxe, un ou plusieurs fichiers sont déplacés dans le répertoire dont le nom est passé comme dernier argument de la commande. `mv` ne permet pas de déplacer un fichier sur lui-même, c'est à dire que le répertoire d'arrivée doit être différent du répertoire de départ. Les droits d'accès nécessaires sont seulement le droit d'écriture dans le répertoire source et arrivée même si le fichier destination est écrasé. En effet il y a suppression du fichier destination avant recréation, ce qui ne demande pas de droit d'écriture sur le fichier.

Exemples

```
mv fich1 fich2
```

Le fichier `fich1` devient le fichier `fich2`.

```
mv fich /tmp
```

Le fichier `fich` disparaît du répertoire courant et va se placer dans le répertoire `/tmp`.

Options

`-i` demande interactive de confirmation avant l'écrasement d'un fichier existant (répondre par `y` ou `n`)

Nota

La syntaxe `mv ancien_nom nouveau_nom` peut servir à changer de nom un répertoire à condition de ne pas le déplacer de répertoire. Le fichier déplacé ne change pas de caractéristiques, en particulier de propriétaire.

Exemple

```
mv rep nouveau_rep
```

4.11 Copie : commande cp

Cette commande duplique des fichiers ou des répertoires complets dans l'arborescence.

Syntaxe

```
cp [-p] fich1 fich2
cp [-p]fich1 [fich2 fich3...] répertoire
cp [-p]-r répertoire_source [fichiers répertoires] rep_dest
```

Description

Dans la première syntaxe, `fich1` est copié dans `fich2`. Bien entendu, les noms de fichiers peuvent être donnés en absolu ou en relatif. Dans la deuxième syntaxe, tous les fichiers ordinaires spécifiés sont recopiés dans le répertoire (qui doit être le dernier argument de la commande) avec les mêmes noms respectifs. La troisième syntaxe avec l'option `-r` permet de copier des répertoire complets récursivement. Un nouveau répertoire du nom du répertoire source est placé dans le répertoire destination.

Il faut noter que dans toutes les circonstances, les fichiers d'origine restent intacts. Le fichier copié appartient au compte réalisant la commande. Les droits nécessaires sont la lecture sur les fichiers sources, l'écriture sur les fichiers qui sont écrasés et l'écriture sur les répertoires destination pour les nouveaux fichiers.

Options

`-p` les fichiers copiés conservent les attributs de droits d'accès et dates du fichier d'origine

`-r` copie récursive de tout le répertoire source.

Exemples

```
cp f1 sous_rep/f2
cp * sous_rep
cp -r /tmp .
```

4.12 Destruction : commande `rm`

Cette commande supprime des fichiers.

Syntaxe

```
rm [-rif] fich1 fich2
```

Description

`rm` supprime les fichiers spécifiés dans le répertoire où la commande a été lancée. Si l'utilisateur de la commande ne dispose pas du droit d'écriture sur le fichier, un message de confirmation d'écrasement est demandé. Une réponse commençant par `y` permet de supprimer le fichier. Le système interdit la suppression uniquement si l'utilisateur n'a pas le droit d'écriture dans le répertoire.

Options

`-r` permet de supprimer tous les fichiers du répertoire et le répertoire lui-même, ce qui suppose un nom de répertoire passé en paramètre. Il est impossible de supprimer le répertoire dans lequel la commande est lancée, et a fortiori un répertoire en amont.

`-i` `rm` est alors interactif, c'est à dire qu'il demandera l'autorisation de supprimer chaque fichier avant de le faire. Cette option est vivement conseillée pour toute destruction massive. Il faut noter en effet qu'il n'existe aucun moyen simple de récupérer un fichier qui vient d'être purgé (sauf bien sûr si on en possède une sauvegarde).

`-f` supprime les demandes interactives lorsque les droits d'écriture ne sont pas présents. Supprime également les messages d'erreurs pour les fichiers inexistants (a utiliser par exemple dans les fichiers de commande).

Nota

La récupération des fichiers supprimés est très difficile. La commande `fsdb` (voir un cours d'administration) permet dans certains cas d'obtenir des résultats.

4.13 Création de liens physiques : commande `ln`

A chaque chemin d'accès de l'arborescence correspond un fichier et un seul. Inversement, un fichier écrit sur le disque est usuellement repéré par un chemin d'accès unique. Cependant, il est possible de définir plusieurs chemins d'accès logiques qui pointent vers le même fichier physique. On dit alors que le fichier a plusieurs liens.

Ce concept s'explique facilement si on se souvient qu'un répertoire est une table d'équivalence entre le nom d'un fichier et son numéro d'inode. Rien n'empêche qu'un même numéro d'inode soit attaché à un nom dans des répertoires différents.
Dans ce cas, il est bien évident qu'une modification apportée à un tel fichier en le manipulant à partir de l'un de ses noms se retrouve si on le manipule à partir d'un autre nom puisqu'il s'agit du même fichier physique.
La commande `ln` établit un lien sur un fichier.

Syntaxe

```
ln nom1 nom2
```

Description

`nom1` est le nom d'un fichier qui doit obligatoirement exister. `nom2` est créé par la commande, mais il ne s'agit que d'un nom désignant le même fichier que `nom1`. Cette procédure permet de manipuler le même fichier à partir de plusieurs répertoires du système de fichiers. Il est également possible de l'utiliser pour désigner dans un même répertoire un fichier par des noms différents (ce qui peut être nécessaire si ce fichier doit être manipulé par des outils logiciels n'ayant pas les mêmes conventions de nomination).

L'existence de liens et leur nombre sont visualisés avec la commande `ls -l`. Le nombre de liens est placé dans la deuxième colonne, après les droits d'accès, pour les fichiers ordinaires. Ce nombre est `1` quand la commande ln n'a pas été utilisée.

La destruction (avec la commande `rm`) d'un fichier muni de plusieurs liens ne provoque en réalité que la destruction du nom correspondant et non celle du fichier lui-même. Le nombre de liens apparaissant sur la ligne d'informations (avec `ls -l`) des autres noms de ce fichier est alors décrémenté de `1`. Lorsque le compteur de liens est égal à 0, le fichier est alors supprimé.

Nota

Il est interdit de créer des liens :

– sur un répertoire

– entre deux systèmes de fichiers

Il résulte de ces deux limites que le lien physique est peu utilisé sur les systèmes modernes, on lui préfère le lien symbolique, décrit ci-après.

4.14 Création de liens symboliques : commande `ln -s`

Comme le lien physique, le lien symbolique permet de définir plusieurs chemins d'accès logiques qui pointent vers le même fichier physique. Le lien symbolique est un fichier spécial différent de l'objet pointé, il contient le nom d'un fichier de type quelconque vers lequel le fichier lien renvoie. Lors de l'utilisation d'un lien, le système trouve le nom du fichier pointé par le lien et utilise en fait ce fichier pointé. Le lien est donc utilisable entre répertoires ou entre systèmes de fichiers différents.

Dans ce cas, il est bien évident qu'une modification apportée à un tel fichier en le manipulant à partir de l'un de ses noms se retrouve si on le manipule à partir d'un autre nom puisqu'il s'agit du même fichier physique. La commande `ln -s` créé un lien symbolique.

Syntaxe

```
ln -s nom1 nom2
```

Description

`nom1` est le nom absolu ou relatif de l'objet vers lequel le lien symbolique va pointer. `nom2` est le nouveau fichier spécial lien crée par la commande.

Nota

La destruction (avec la commande `rm`) d'un lien symbolique détruit le lien lui même et non le fichier pointé. Le type lien et le contenu du fichier lien (le nom de l'objet pointé) sont donnés par la commande `ls -l`. Le lien sur un répertoire permet de "descendre" dans l'arborescence, mais pas de "remonter", de plus l'indication donnée par la commande `pwd` ne correspond plus avec le nom du lien donné à la commande `cd`.

La commande ne vérifie pas l'existence de l'objet vers lequel le lien pointe, si l'objet n'existe pas l'erreur se produit à l'utilisation. Les droits d'accès du liens ne sont jamais utilisés, le contrôle à lieu sur l'objet pointé.

Exemple

```
ls -l /bin  lrwxrwxrwx   1   root root   9 Jul 5 14:33 bin -> ./usr/bin
```

4.15 Nature d'un fichier : commande `file`

Cette commande a pour objet de déterminer le type de l'information contenue dans un fichier.

Syntaxe

```
file fichier1 fichier2 ...
```

3 niveaux de droits	u (user)	g (group)	o (others)
3 types d'autorisation	r w x	r w x	r w x
Codage	4 2 1	4 2 1	4 2 1

TABLE II.7 – Codage numérique des droits d'accès.

Description

La commande fait une série de tests sur chaque fichier afin de classifier la nature des informations contenues. Les principales réponses de la commande sont les suivantes :

`ASCII text`	Fichier texte ou programme source
`commands text`	Fichier de commande shell
`executable`	Fichier binaire exécutable
`directory`	Répertoire
`data`	Fichier de données
`block special`	Fichier spécial type bloc
`character special`	Fichier spécial type caractère
`symbolic link`	Fichier spécial lien symbolique

5 Commandes agissant sur les droits d'accès

Les commandes agissant sur les permissions utilisent parfois un codage sous forme octale qui est explicité ci-dessous. Les droits de lecture, écriture et exécution associés au 3 cas utilisateur, groupe ou public donne un jeu de 9 permissions vraies ou fausses. Pratiquement, ces permissions sont codées dans 9 caractères binaires groupés 3 par 3. Chaque classe d'utilisateur a ses droits d'accès fixé par un chiffre qui peut varier de 0 à 7. La Table II.7 illustre ce codage.

5.1 Permissions par défaut : commande `umask`

Linux fixe par défaut les droits d'accès à 644, c'est à dire lecture et écriture autorisées pour le propriétaire et lecture seule pour le groupe et les autres. Cette disposition n'est pas nécessairement souhaitable comme permissions standard.

Pour pouvoir affecter un autre jeu de permissions automatiquement sans que l'utilisateur ait à le préciser à chaque fois, le système Linux fournit une commande de positionnement des droits par défaut lors de la création d'un fichier ou d'un répertoire.

La commande `umask` fixe les permissions d'accès à un fichier à la création.

Syntaxe

```
umask [permission]
```

Description

`permission` est un entier de 3 chiffres, qui représentent chacun la restriction des droits d'accès demandée par rapport à `666` pour les 3 classes d'utilisateurs, à savoir de gauche à droite, le propriétaire, le groupe et le public. Pour obtenir le nouveau positionnement par défaut, il faut soustraire de la valeur `666` la permission fixée par `umask`.

Exemple

```
umask 022
```

à partir de cette commande tous les fichiers seront créés avec les droits correspondant à `644` (puisque `0666 & ~022 = 0644`, i.e., `rw-r-r-`), soit lecture seule autorisée pour tous les utilisateurs sauf pour le propriétaire qui conserve la lecture et l'écriture.

Nota

- `umask` est une commande interne du shell (`bash` sous Ubuntu).
- Si on fait plusieurs appels successifs à `umask`, il faut toujours raisonner à partir du codage `666`.
- Si le paramètre `permission` est omis, la commande retourne la valeur courante des permissions (fixée lors du dernier appel à `umask`).

5.2 Changement des permissions : commande `chmod`

Cette commande permet de modifier les droits d'accès d'un fichier.

Syntaxe

```
chmod [-R] mode fichier1 ...
```

Description

`chmod` permet de redéfinir les droits d'accès d'un ou plusieurs fichiers. La commande n'est exécutée par Linux que si l'utilisateur est le propriétaire des fichiers spécifiés, ou s'il est le super-user. Il y a deux façons de définir le paramètre

Mode symbolique

```
chmod [qui] [op] [quoi] fichier
```

avec

qui :	`u` (propriétaire)	`g` (groupe)	`o` (public)	`a` (tous)
op :	`+` (ajouter)	`-` (enlever)	`=` (nouveau)	
quoi :	`r` (lecture)	`w` (écriture)	`x` (exécution)	

Options

`-R` Permet de changer les propriétaires récursivement en parcourant les répertoires rencontrés.

Exemple

```
chmod g-r,o-w fich
```

supprimera dans le fichier `fich`, le droit de lecture pour le groupe et le droit d'écriture pour le public.

Mode absolu

En utilisant le codage défini en Table II.7, p. 71, on précisera le mode à l'aide de 3 ou 4 chiffres octaux, le premier indiquant éventuellement à Linux que la valeur est octale, le 2ème désignant la permission du propriétaire, le 3ème celle du groupe et le dernier, celle du public.

Exemple

```
chmod 0777 fich
chmod 0600 fich
```

tous droits pour tous lecture et écriture pour le propriétaire.

Nota

On remarquera que les commandes `umask` et `chmod` ne vérifient pas que les droits attribués sont compatibles avec la nature du fichier manipulé. Elles autorisent aussi bien l'attribution du droit de lecture sur un fichier binaire, que le droit d'exécution sur un fichier texte ...

5.3 Transfert de propriété d'un fichier : commande `chown`

La commande `chown` donne la possibilité de transférer les droits de propriété d'un fichier à un autre utilisateur.

Syntaxe

```
chown [-R] nouveau_propriétaire fichiers ...
```

Description

La commande change les propriétaires des fichiers et répertoires donnés en argument. Elle n'est acceptée par Linux que si elle est manipulée par le propriétaire des fichiers spécifiés ou par le super-user. Certains systèmes sont configurés pour réserver l'usage de cette commande au super-user.

Options

`-R` Permet de changer les propriétaires récursivement en parcourant les répertoires rencontrés.

5.4 Transfert de propriété d'un fichier : commande `chgrp`

Syntaxe

```
chgrp [-R] groupe fichiers ...
```

Description

La commande `chgrp` permet d'affecter les fichiers spécifiés à un autre groupe, celui indiqué par le 2ème paramètre. L'option et les restrictions d'emploi sont les même que pour la commande `chown`.

Nota

Les commandes `chmod`, `chown` et `chgrp` sont particulièrement utiles lorsqu'on manipule les commandes de copie, notamment `cp`. Dans ce cas en effet, les droits d'accès sont transmis par défaut et le propriétaire est celui qui réalise la copie. Il est donc parfois nécessaire d'adapter les droits et fixer un nouveau propriétaire.

6 Outils de traitement de fichiers

Les commandes présentées sont une sélection des commandes de traitement de fichiers ascii les plus utilisées. Ces outils sont employés directement au clavier ou bien lors de la réalisation de petits programmes utilitaires écrits en shell.

6.1 Commande `wc`

Comptage du nombre de lignes, de mots et/ou de caractères d'un fichier.

Syntaxe

```
wc [-lwc] [fichiers]
```

Description

Donne le nombre de lignes, de mots et de caractères pour la liste des fichiers ou pour l'entrée standard si les fichiers sont omis. Donne aussi le total pour chaque fichier.

Options

`-l` donne seulement le nombre de ligne.

`-w` donne seulement le nombre de mots.

`-c` donne seulement le nombre de caractères

Les trois options peuvent être utilisées dans n'importe quelle combinaison.

6.2 Recherche de chaînes : commande `grep`

Recherche d'une chaîne de caractères dans des fichiers.

Syntaxe

```
grep [options] expression [fichiers]
```

Description

La commande sort les lignes des fichiers contenant la chaîne de caractères "expression". Une expression peut contenir des caractères spéciaux, nommés expressions régulières, sur le modèle des recherches de chaînes de caractères sous les éditeurs `vi` ou `ed`. Par exemple le caractère `^` représente le début de la ligne, tandis que `$` représente la fin de ligne. S'il y a plus d'un fichier à analyser, la commande sort également les noms des fichiers.

Par défaut l'entrée standard est utilisée. Il est prudent de placer entre guillemets simples une expression contenant des caractères spéciaux. Le code de retour est 0 dans le cas où

la chaîne est trouvée, 1 dans le cas contraire et 2 en cas d'erreur de syntaxe ou de fichier inaccessible.

Expressions régulières

Les expressions régulières sont des suites de caractères et d'opérateurs sur caractères permettant de faire des sélections. Voici les opérateurs les plus courants :

- `^` début de ligne
- `.` un caractère quelconque
- `$` fin de ligne
- `x*` zéro ou plus d'occurrences du caractère x
- `x+` une ou plus occurrences du caractère x
- `x?` une occurrence unique du caractère x
- `[...]` plage de caractères permis
- `[^...]` plage de caractères interdits
- `\{n\}` pour définir le nombre de répétition `n` du caractère placé devant

Exemples

L'expression `[a-z][a-z]*` désigne une chaîne d'au moins un caractère en minuscule ; L'expression `^[0-9]\{4\}$` désigne, du début à la fin de la ligne, les nombres ([0-9]) de 4 chiffres (`\{4\}`).

Options

- `-v` affiche les lignes qui ne contiennent pas la chaîne.
- `-c` imprime seulement le nombre de lignes contenant la chaîne.
- `-i` ignore la différence entre majuscules et minuscules.
- `-l` seuls les noms de fichiers contenant la chaîne sont affichés.
- `-n` la ligne est précédée par son numéro de ligne dans le fichier.

6.3 Substitution de caractères : commande `tr`

La commande `tr` filtre de traduction des caractères transmis.

Syntaxe

```
tr [-ds] [chaine1 [ chaine2 ] ]
```

Description

La commande `tr` copie l'entrée standard vers la sortie standard en effectuant une traduction ou une suppression de certains caractères. Les caractères lus et correspondant à ceux placés dans `chaine1` sont remplacés par le caractère de rang égal dans `chaine2`. Si la première chaine est trop longue par rapport à la seconde, les caractères supplémentaires ne sont pas traduits.

Les abréviation suivantes sont valides :

`[a-z]` suite de caractères suivant la table ascii.

`[a*n]` répétition du caractère `a` `n` fois.

Comme pour toute chaîne contenant des caractères spéciaux pour le shell, on peut encadrer les chaînes par des guillemets simples.

Options

`-d` supprime les caractères précisés dans chaine1.

`-s` ramène à une seule occurence les caractères multiples précisés dans chaine2.

Exemples

```
tr -d '[0-9]' <f1 >f2
```

supprime les chiffres du fichier `f1`, place le résultat dans `f2`.

```
tr '[A-Z]' '[a-z]'
```

transforme les majuscules en minuscules.

```
ps -ef | tr -s ' '
```

supprime les blancs en double dans la sortie de la commande `ps`.

6.4 Extraction de champs : commande cut

Extraction de champs de fichiers texte.

Syntaxe

```
cut -cliste [fichier1 fichier2 ......]
cut -fliste [-dc] [-s] [fichier1 fichier2 ......]
```

Description

La commande extrait des colonnes des fichiers de la ligne de commande. Les colonnes sont des positions en caractères dans la ligne ou des champs délimités par des tabulations par défaut. La commande est un filtre s'il n'y a pas de fichiers en argument. La liste `liste` est une suite de numéros de champs ou colonnes séparés par des virgules ou des `-`. Par exemple `1,3,5` ou `1-10,12`.

Options

`-cliste` position en caractère.

`-fliste` position en champ (les lignes sans séparateur sont transmises).

`-dc` définition de `c` comme caractère séparateur, par exemple pour le blanc `-d" "`.

`-s` suppression des lignes sans séparateur (option `-f`).

Nota

Ne pas mettre de blanc à la suite des options `c`, `f` et `d` (contrairement à `tr`).

Exemple

```
cut -d: -f1,3 /etc/passwd
```

Obtenir la liste des noms d'utilisateurs et des numéros associés :

6.5 Sélection de lignes : commande `tail`

Affiche la fin d'un fichier en commençant à une ligne donnée.

Syntaxe

`tail [ ±[nombre] [lbc] ] [fichier]`

Description

La commande `tail` copie le fichier placé en argument vers la sortie standard en commençant à l'endroit précisé par les options. Par défaut l'entrée standard est lue. Par défaut la sortie commence 10 lignes avant la fin du fichier. Le nombre précise la position où commencer, `+` signifie à partir du début, `-` à partir de la fin. La position peut être donnée en lignes, (ajouter `l`), en blocs (ajouter `b`) ou en caractères (ajouter `c`).

Exemple

`tail -20c fich`

Sort les 20 derniers caractères de `fich`

7 Commandes diverses

7.1 Commande `clear`

Cette commande efface l'écran du terminal ou de la fenêtre active.

Syntaxe

`clear`

7.2 Commande `sleep`

Cette commande permet de faire une pause dans les programmes écrits en shell.

Syntaxe

`sleep n`

Description

La commande attend n secondes avant de terminer.

7.3 Commande `times`

Syntaxe

`times`

Description

Affiche le temps cumulé utilisateur et système pour les processus lancés par le shell courant (ne pas confondre avec la commande `time` qui donne les statistiques pour une commande)

8 Mémento de commandes Linux

Aide sur les commandes	
`man ls`	Appel de l'aide pour la commande `ls`
`-h ou -help`	Demande d'aide pour une commande
`ls -help`	Demande d'aide pour la commande ls
Impression	
`lpr -Phpv prog.ps`	Impression du fichier prog.ps sur hpv
`lpq`	File d'attente sur l'imprimante par défaut
`lpq -Plp`	File d'attente sur l'imprimante lp
Manipulation des fichiers	
`ls`	Liste des fichiers du répertoire
`ls -l`	Liste détaillée des fichiers du répertoire
`cd`	Déplacement dans l'arborescence des fichiers
`cd /etc`	Déplacement dans le répertoire etc
`pwd`	Nom du répertoire courant
`cd ..`	Positionnement sur le répertoire parent
`mkdir prog`	Création du répertoire prog
`cd prog`	Déplacement dans le sous-répertoire prog
`rmdir prog`	Effacement du répertoire prog
`cp prog1.c prog2.c`	Copie du fichier prog1.c dans prog2.c
`rm prog1.c`	Effacement du fichier prog1.c
`mv prog1.c prog2.c`	Renommage ou déplacement du fichier prog1.c en prog2.c
`file prog.c`	Type du fichier prog.c
`wc prog.c`	Nombre de lignes, de mots, de caractères, du fichier prog.c
`cat prog.c`	Liste du contenu du fichier prog.c
`cat a.txt >> b.txt`	Copie du fichier a.txt au bout du fichier b.txt
`more prog.c`	Liste du contenu du fichier prog.c, arrêt en bas d'écran
`less prog.c`	Liste du contenu du fichier prog.c, amélioration de more
`grep main prog.c`	Affiche toutes les lignes du fichier prog.c contenant main
`vi prog.c`	édition avec l'éditeur vi du fichier prog.c
`emacs prog.c`	édition avec l'éditeur emacs du fichier prog.c
`chmod a+r fich.htm`	Permission de lecture pour tous du fichier fich.htm
`sort fich.txt`	Tri du fichier fich.txt
`cmp a.txt b.txt`	Compare deux fichiers
`diff a.txt b.txt`	Affiche les différences entre les deux fichiers
`touch fich.txt`	Crée un fichier vide de ce nom s'il n'existe pas, sinon change la date de dernière modif. du fichier
Combinaisons de commandes	
`echo "Bonjour" > fich.txt`	Ecriture de "Bonjour" dans le fichier fich.txt
`ls > liste.txt`	Envoi de la liste des fichiers du répertoire dans liste.txt

`ls >> liste.txt`	Idem mais copié au bout de liste.txt
`|`	Envoi de la sortie d'une commande dans l'entrée de la suivante
`ls | wc -l`	Nombre de fichiers du répertoire en cours (wc -l compte les lignes affichées par ls)
Gestion de la session	
`passwd`	Changement du mot de passe
`who`	Utilisateurs connectés
`w`	Utilisateurs connectés et action en cours
`whoami`	Userid de la session en cours
`id`	uid et gid (numéro d'utilisateur et de groupe)
`h`	Historique des commandes
`"`	Commande précédente
`echo "Bonjour"`	Affichage d'une chaîne de caractères
`echo $PATH`	Affichage du chemin d'accès aux commandes
`printenv`	Affichage des variables d'environnement
`alias`	Liste des alias
`tty`	Nom du terminal
`locale`	Affiche les options locales de langue
`exit`	Quitte le shell (ou la session)
`logout`	Idem
`Ctrl-d`	Idem (Ctrl = touche contrôle)
Compression et archivage	
`tar tzvf prog.tar.gz prog`	Liste (v) de la table (t) des fichiers de l'archive prog.tar.gz
`tar czf prog.tar.gz prog`	Création (c) d'un fichier archive (f) prog.tar.gz comprimé (z) à partir de tous les fichiers de l'arborescence prog
`tar xzf prog.tar.gz prog`	Extraction (x) des fichiers de l'archive prog.tar.gz
`gzip fich.txt`	Compression du fichier fich.txt en fich.txt.gz
`gunzip fich.txt.gz`	Décompression du fichier fich.txt.gz en fich.txt
`gzip -d fich.txt.gz`	Idem
Gestion des processus	
`ps auxr`	Liste des process en cours d'exécution
`ps aux|more`	Liste de tous les process
`top`	Suivi de l'activité de la machine
`&`	Mise en arrière plan d'un processus
`prog &`	Lancement de prog en arrière plan
`fg`	Mise en avant plan d'un processus stoppé
`jobs`	Liste des jobs en arrière plan
`kill %1`	Tue le job d'arrière plan [1]
`kill 1492`	Tue le processus de PID 1492
`free`	Espace mémoire disponible
Gestion de l'espace disque	
`df`	Espace occupé/disponible sur les disques montés

`mount`	Liste des disques montés
`mount /cdrom`	Montage d'un cd-rom
Compilation	
`gcc -o prog prog.c`	Compilation C du fichier prog.c, exécutable dans le fichier prog
`gcc -o prog prog.c lm`	Idem avec recherche de fonctions dans la librairie mathématique
`gcc prog.c`	Compilation C du fichier prog.c, code objet dans le fichier a.out
`gcc -c prog.c`	Compilation C du fichier prog.c, code objet dans le fichier prog.o
`./prog`	Exécution du programme prog
`g++ -o hello hello.C`	Compilation C++ du fichier hello.C
`ar crv libamoi.a sub1.o`	Rangement d'un code objet dans une librairie personnelle
`ar t libamoi.a`	Liste des fichiers objet d'une librairie personnelle
`gcc -o prog prog.c L. -lamoi`	Compilation C avec recherche de sousprogrammes dans la librairie libamoi.a du répertoire courant
`make`	Exécution des commandes du fichier Makefile
`ldd prog`	Librairies partagées appelées par le programme exécutable prog
`nm prog`	Symboles du programme exécutable prog
`gdb prog`	Appel du débogueur pour la recherche des erreurs du programme exécutable prog
`strace date`	Trace des appels systèmes de la commande date
`strip prog`	Enlève les symboles du programme exécutable prog

III – Le shell

1 Introduction

Le shell est un programme dont la fonction est de servir d'interface entre les utilisateurs et le noyau. Le shell interprète les commandes tapées interactivement par les utilisateurs et lance les exécutions demandées. Conformément à la philosophie du système Linux, le shell peut également lire les commandes depuis un fichier disque au lieu de lire depuis l'entrée standard du terminal.

Le shell est donc dans ce cas un interpréteur de commande qui peut être utilisé comme un langage de programmation de haut niveau. Le langage de programmation shell permet

de disposer de variables, de structures de contrôles et de redirections pour le traitement des entrées/sorties.

Le terme “shell” est un terme générique, en fait il existe plusieurs interpréteurs de commandes différents sur les machines Linux récentes. Les plus utilisés sont :

- Le "Bourne shell", `/sbin/sh`, écrit dès le début de la vie d'Linux (l'ancêtre de Linux) par S. Bourne et a été distribué sur tous les systèmes Linux depuis 1977. Ce shell était et est toujours utilisé pour l'écriture des fichiers de commandes de la plupart des systèmes Linux.
- Le "bash", `/usr/bin/bash`, présent sur Linux. Il est compatible avec le Bourne shell. Le rappel et la manipulation des commandes se fait à l'aide des flèches du clavier.

Dans ce manuel nous présentons à la fois les commandes du Bourne shell `sh` de façon à pouvoir lire ou modifier les programmes anciens et bien entendu les fonctionnalité les plus étendues du `bash` qui est désormais le shell le plus répandu.

2 L'interface shell-Linux

2.1 Commandes et processus

L'utilisation du shell de manière avancée nécessite la compréhension du mécanisme à la base du lancement des programmes sous Linux. Dans tous les cas le shell lit la commande demandée par l'utilisateur, deux cas se présentent alors :

- La commande peut être traitée directement par le shell sans faire appel à un programme extérieur. La commande est dite **interne**.
- La commande n'est pas connue du shell, qui la recherche dans un fichier du système. La commande est dans ce cas dite **externe**.

Dans le cas de commande interne, il n'y a besoin d'aucune tâche supplémentaire, et le résultat est très rapide. De plus certaines fonctionnalités doivent par principe être traitées par le shell lui-même comme par exemple la commande `cd` qui modifie une information du shell. Le `bash` dispose d'environ 40 commandes internes.

Dans le cas d'une commande externe, il y a lancement d'un programme, ce qui se traduit par la création d'une nouvelle tâche sur le système dite sous Linux un **processus**. Un processus n'est pas créé directement par le système, c'est toujours un processus déjà existant qui est à l'origine du nouveau processus.

On dit que le nouveau processus est le **fils** du processus antérieur, tandis que celui-ci est le **père** du nouveau processus. Le processus fils peut être le même programme (le même code) que le processus père ou non. Dans tous les cas les deux processus sont différents pour le système, en particulier ils portent un numéro différent.
Le shell qui lance une commande externe est le père de cette tâche, et le code du père et du programme fils sont donc toujours différents (sauf si on lance un nouveau shell explicitement depuis le shell courant).

2.2 Visualisation des processus : commande ps

La commande `ps` affiche l'état des processus en cours d'exécution. Elle permet donc de retrouver par exemple les numéros des processus que l'on utilise, le shell et les processus lancés par celui-ci.

Syntaxe

```
ps [-ef]
ps [-aux]
```

Description

Le choix des processus visualisés est fonction des options de la commande. Sans options, `ps` rapporte les processus associés au terminal de la commande, soit les processus lancés par l'utilisateur.

Options

- Pour voir chaque processus du système avec une syntaxe standard :
  ```
  ps -e
  ps -ef
  ps -eF
  ps -ely
  ```
- Pour voir chaque processus du système, avec une syntaxe dite BSD :
  ```
  ps ax
  ps axu
  ```
- Pour afficher un arbre des processus :
  ```
  ps -ejH
  ps axjf
  ```
- Pour avoir des informations à propos des threads :
  ```
  ps -eLf
  ps axms
  ```
- Pour obtenir des informations de sécurité :
  ```
  ps -eo euser,ruser,suser,fuser,f,comm,label
  ps axZ
  ps -eM
  ```
- Pour voir les processus tournant avec les droits de `root` (ID réel et effectif) :
  ```
  ps -U root -u root u
  ```

Valeurs d'état des processus (champ `STAT` ou `S`) :

`D` Sommeil ininterruptible (généralement dû à une E/S)

`R` Actif ou prêt (dans la file d'attente des tâches prêtes)

`S` Sommeil interruptible (en attente de complétion d'un évémenement)

`T` Stoppé par un signal ou en train d'être tracé

`X` Mort (ne devrait pas être vu)

`Z` Processus défunt ("zombie"), terminé, mais pas encore recueilli par son parent.

- `<` Haute priorité
- `N` Basse priorité (Nice to others)
- `L` Possède des pages verouillées (Locked) en mémoire (pour du temps réel et des E/S sur mesure)
- `s` Leader de session
- `l` Multi threadé (utilisant `CLONE_THREAD`, comme les threads `NPTL`)
- `+` Dans le groupe de processus d'avant plan

2.2.1 Commandes internes

Les commandes internes sont utilisables de façon transparente pour l'utilisateur ; en particulier il est possible de rediriger les entrées/sorties des commandes internes, sous réserve que la redirection ait un sens pour la commande. La liste complète des commandes interne se trouve dans le manuel du shell (`man bash`), on trouvera ici pour mémoire une liste de commandes internes déjà présentées. D'autre commandes utiles seront données au fil de ce support. Quelques commandes internes : `cd echo pwd umask times`

2.2.2 Commandes externes

Lorsque le shell de connexion exécute une commande externe lue sur son entrée standard, clavier ou fichier, il crée un nouveau processus fils qui contient le code de la commande à exécuter. Le shell de connexion est donc le père de tous les processus de l'utilisateur connecté. Le nom de la commande peut être donné d'une façon absolue, c'est à dire que le chemin d'accès commence par un `/`. Il peut aussi être donné de façon relative avec un `/` dans le chemin.

Dans les deux cas, le shell recherche le fichier correspondant à la commande seulement dans le répertoire précisé par le chemin. On utilisera ce mode par exemple pour avoir la certitude d'exécuter un fichier dans un répertoire particulier. Cette méthode permet également d'appeler une commande externe de même nom qu'une commande interne. En effet le shell exécute toujours les commandes internes en priorité.

Exemple

Pour exécuter une commande externe ayant (par erreur) le nom de la commande interne **test** du shell.

```
./test
```

En général et conformément à la syntaxe d'une commande présentée ci-dessus l'utilisateur donne le nom de la commande directement sans préciser le chemin d'accès. Le shell recherche alors ce fichier dans un certain nombre de répertoires par défaut. On choisit la plupart du temps les répertoires où se trouvent les commandes système conjointement à son répertoire courant. On verra dans la suite de l'exposé que l'utilisateur peut préciser les répertoires qu'il souhaite voir analyser par le shell au moyen de la variable `PATH` (voir la sous-section 2.3.9, p. 89). Par défaut la recherche se fait successivement dans les répertoires suivants : `/usr/bin`, répertoire courant.

Il existe deux cas de figure lors du lancement d'une commande externe par le shell. La commande peut correspondre à un fichier exécutable pur. Il s'agit dans ce cas d'un programme compilé par un utilisateur ou faisant partie des commandes livrées avec le système. Le second cas correspond à un fichier de commande destiné à être lu par le shell.

Commandes exécutables. Les commandes exécutables sont directement lancées par le shell. Celui-ci attend la fin de l'exécution de cette commande fille, renvoie le "prompt" (invite du système) et lit la commande suivante. Dans ce cas il n'y a qu'un seul processus fils créé par commande. La plupart des commandes système sont des exécutables.

Procédures shell. Dans ce cas le shell père crée un shell fils qui va lire le contenu du fichier à exécuter, interpréter les commandes et créer lui-même pour chaque commande un processus fils. Lorsque la liste des commandes est épuisée le fils se termine. Le père peut lire la commande suivante comme dans le cas précédent. Pour l'utilisateur la distinction entre une procédure shell et un exécutable est totalement transparente.

Toutefois l'utilisation de procédures shell génère un grand nombre de processus, ce qui ralentit la machine. D'autre part, l'interprétation des commandes est par nature plus lente que l'exécution d'un programme compilé. On notera que les procédures de commandes peuvent être lues pour analyser leur fonctionnement et même éventuellement modifiées par le responsable du système.

2.2.3 Création du shell courant au login

Le shell est un processus qui s'exécute sur le système au même titre que tous les autres programmes. L'interpréteur de commande shell dont on dispose après une connexion réussie est donc le fils d'un processus père qui est chargé de réaliser la connexion et de créer le shell pour l'utilisateur. Sur les premiers systèmes Linux, ce père était commun à toutes les connexions : il s'agissait du processus `init`, dont le numéro d'identification (`PID`) est `1`. `init` est à l'origine de tous les processus sur une machine Linux.

Sur les systèmes récents, le contrôle des connexions est généralement réalisé par un processus spécialisé dit "Port Monitor". Ce processus lance une tâche `login` de vérification de la connexion. C'est cette tâche qui est généralement le père du shell de connexion. Dans tous les cas il suffit de lancer la commande `ps -ef` pour suivre l'arborescence des processus père et fils et connaître les noms des taches en amont de son shell de travail.

2.3 Notion d'environnement

2.3.1 Généralités

Lorsque qu'un processus shell s'exécute, il conserve un certain nombre d'informations qui sont regroupées ici sous le nom d'environnement du shell. Lorsque le shell est utilisé interactivement sur un terminal, son environnement est en fait celui de l'utilisateur.

Par exemple les variables maintenues par le shell, qu'elles soient créées par le shell ou gérées automatiquement par celui-ci, font partie de l'environnement. Les différentes composantes de l'environnement sont détaillées dans ce qui suit.

2.3.2 Les variables utilisateur

Les variables utilisateur ; sont créées par chaque utilisateur en fonction des besoins. Ces variables contiennent des chaînes de caractères. La variable est connue grâce à un nom choisi arbitrairement. Le nom doit toutefois contenir seulement des lettres, chiffres et le signe `_` ; de plus il doit débuter par une lettre ou le signe `_` (souligné). Il n'y a pas de limitation théorique de la longueur du nom ou du contenu. En utilisation courante on peut manipuler des chaînes de plus de 128 caractères.

On accède au contenu de la variable en plaçant un `$` devant le nom (sans blanc). Par exemple si on suppose que la variable nom contient la chaîne de caractère `Paris`, la commande :

```
echo \$nom
```

fera afficher non pas la chaîne `$nom` mais le contenu de la variable `nom`, c'est à dire `Paris`.

2.3.3 Affectation d'une variable

Pour affecter ou assigner une variable, c'est à dire définir son contenu, il faut placer le signe `=` entre nom et contenu. Par exemple on tapera :

```
nom=Paris
```

Si la variable `nom` n'existait pas, elle est créée par l'affectation. On notera l'**absence de blancs de part et d'autre du signe égal** ; ceci est impératif pour affecter une variable. Il est possible de placer plusieurs assignations sur la même ligne de commande. Il est alors important de noter que le shell réalise ces assignations en séquence **de la droite vers la gauche**. Par exemple le ligne de commande suivante assigne la chaîne de caractère `2` à la variable `var1` :

```
var1=$var2 var2=2
```

En effet on place d'abord `2` dans `var2` puis `$var2` dans `var1`. Bien entendu, il est facile de réutiliser une variable par simple assignation, toutefois il existe une commande interne `unset` permettant de détruire des variables.

2.3.4 La commande `unset`

Syntaxe

```
unset [nom ...]
```

Description

Pour chaque `nom` retire les variables ou fonctions définies pour le shell courant (en pratique le nom demeure mais le contenu est effacé). Les variables en lecture seule (voir la commande `readonly`) et les variables `PATH`, `PS1`, `PS2`, `IFS` ne peuvent être retirées.

2.3.5 Règles de construction des chaînes de caractères

Dans le paragraphe précédent on a utilisé des chaînes simples sans blancs ni caractères spéciaux. Si l'on souhaite par exemple construire des chaînes de caractères contenant des blancs il faut utiliser une notation spéciale. En effet le shell utilise les blancs pour délimiter les arguments d'une commande.

On dispose pour cela de deux caractères ayant une signification particulière pour le shell. Il s'agit des guillemets simples ou apostrophes (') et des guillemets double ("). Tous les caractères placés entre deux guillemets simples perdent leur signification particulière pour le shell (le guillemet simple lui même ne peut pas être placé dans la chaîne). Entre guillemets double, seuls les caractères suivants gardent leur sens spécial : $, \, ‘ et ". L'anti-slash permet de retrouver le sens littéral des 4 caractères précédents. Devant tout autre caractère, il est pris dans son sens littéral et conservé dans la chaîne.
Voici un exemple d'utilisation des guillemets :

```
var1=papa var2=maman
var="$var1 $var2"
```

La variable `var` contient dans ce cas la chaîne : `papa maman` ; il y a eu interprétation du caractère $, et substitution du nom de variable par son contenu. Dans le cas suivant le $ n'est pas interprété, la chaîne est conservée telle quelle :

```
var='$var1 $var2'
```

On peut également concaténer les chaînes de la façon suivante :

```
var='la'
var="$var variable"
var="$var est une chaîne de caractères"
```

On obtient dans la variable la chaîne suivante :
`la variable est une chaîne de caractères`
Il est aussi possible d'utiliser le caractère \ (anti-slash) pour faire perdre au caractère qui le suit son sens particulier pour le shell. Par exemple on placera un guillemet dans une phrase de la façon suivante :

```
echo les guillemets sont une source d\'erreur
```

L'utilisation des guillemets et de l'anti-slash n'est pas limitée aux chaînes de caractères pour l'affectation des variables. Tout argument d'une ligne de commande peut être construit à l'aide de ces mécanismes. Par exemple on utilisera le caractère \ juste avant le retour chariot pour éviter qu'il ne déclenche la lecture de la commande. De cette façon il est possible de taper une ligne très longue.

La construction de chaînes de caractères contenant à la fois du texte et des noms de variables amène rapidement au problème de la délimitation du nom de variable par rapport aux autres caractères. Pour résoudre ce point, on dispose des accolades ouvrantes et fermantes qui peuvent encadrer et délimiter tout nom de variable.

Exemple

```
tout="${deb}e la ligne de test"
```

`tout` contient la chaîne suivante : `ceci est le debut de la ligne de test`.
Lorsque l'on omet les accolades et qu'il n'y a pas de blancs pour délimiter, il en résulte généralement une variable non définie, dont le contenu est par défaut une chaîne nulle. Avec certaines options du shell, une erreur est signalée. Dans le cas de l'exemple, on aurait obtenu dans la variable `"tout"` la chaîne suivante : `la ligne de test`

2.3.6 Lecture de variables : commande `read`

Dans le but de construire des programmes shell interactifs, il existe une commande de lecture et d'affectation de chaînes de caractères, `read`. Cette dernière lit une ligne à partir de l'entrée standard et assigne successivement les chaînes lues aux variables citées en argument.

Syntaxe

```
read var1 var2 .....
```

Description

Si le nombre de mots de la ligne dépasse le nombre de variables, la dernière variable contient tous les mots en excès. Si plusieurs mots doivent être entrés dans une même variable, il doivent être entourés de doubles guillemets, conformément à la syntaxe du shell.

Exemple

```
read prenom nom profession
```

ligne entrée : `Evariste Galois professeur`
La commande précédente affecte les variables de la façon suivante :

```
nom=Galois
prenom=Evariste
profession=professeur
```

2.3.7 Substitution de commande

La substitution de commande permet de placer le résultat d'une commande à l'intérieur d'une chaîne de caractères. Le shell exécute la commande, il en résulte une chaîne de caractères obtenue sur la sortie standard de la commande. C'est cette chaîne résultat qui est substituée au libellé initial de la commande.

Deux syntaxes sont possibles :

– Syntaxe traditionnelle du Bourne shell.
La commande à exécuter est placée entre guillemets inversés (` correspondant souvent à la touche `Alt Gr/7`). Par exemple pour obtenir la date et l'heure courante dans une variable on tapera :

```
jour=‘date‘
```

Cette commande assigne dans la variable `jour` le résultat de la commande `date`. On prendra garde à la différence de comportement entre une chaîne placée entre guillemets simples et guillemets inversés. Dans le premier cas la chaîne reste telle quelle, dans le second cas on exécute une commande.

Il est possible d'avoir plusieurs niveaux de substitution de commande. Pour cela on fera précéder les guillemets intérieurs d'un anti-slash. Par exemple :

```
bonjour=‘banner "Vous êtes dans le répertoire \‘pwd\‘"‘
```

On pourra afficher en gros le texte en faisant echo `$bonjour`.

– Syntaxe normale du `bash`
La substitution est précédée d'un `$` comme pour les variables et la commande est placée entre parenthèses : `$(commande)`

Exemple

```
var=$(date)
```

2.3.8 Variables protégées en écriture

Il est possible d'empêcher l'écriture dans une variable utilisateur qui a déjà été définie et affectée. Pour cela on utilisera la commande `typeset -r` sous `bash` ou `readonly` sous `sh`.

Syntaxe

```
readonly [noms....]
typeset -r [noms....]
```

Description

Les variables placées dans la liste deviennent protégées en écriture. Dans le cas où la commande est tapée sans argument, on obtient une liste de toutes les variables en lecture seulement.

2.3.9 Variables du shell

Ces variables sont aussi appelées variables d'environnement. Ces variables ont des noms fixés et écrits en majuscule par convention. L'administrateur du système ou l'utilisateur peuvent modifier ces variables de la même façon que les variables utilisateur.

Ces variables servent à configurer le shell lui même ou d'autres programmes d'application comme par exemple l'éditeur. Les noms de ces variables sont fixés et écrits en majuscule par convention.

Voici une liste des principales variables shell :

`PATH` Cette variable définit les répertoires où le shell va chercher les fichiers exécutables correspondant au nom des commandes tapées par l'utilisateur. Elle est importante puisqu'elle conditionne les commandes qui sont accessibles directement par leur nom simple.
Son contenu est une liste de noms complets de répertoires séparés par des deux points (:). Le répertoire courant est spécifié par une chaîne nulle. L'ordre de recherche est celui des répertoires dans la variable, il peut donc être changé par l'utilisateur.
Par défaut, en Bourne shell la valeur de `PATH` est `PATH=/bin:/usr/bin`. Sous Linux, elle est définie par l'administrateur à l'installation du `bash`.

`HOME` Cette variable contient le répertoire de connexion de l'utilisateur (*home directory*). Elle est initialisée au moment de la connexion et est disponible pour le shell durant toute la session. La commande `cd` sans argument utilise par exemple cette variable. On peut utiliser le contenu de cette variable afin d'éviter l'écriture du nom complet du répertoire. Dans une procédure shell, son utilisation rend le programme indépendant du répertoire de l'utilisateur.

`LOGNAME` `LOGNAME` contient le nom de l'utilisateur qui est connecté. Cette variable est utile pour adapter par exemple une procédure en fonction de l'utilisateur.

`PS1` Variable définissant la chaîne de caractères utilisée par le shell en attente de commande (prompt). Par défaut le shell initialise cette variable à `$` pour un utilisateur normal et à `#` pour le super utilisateur.

`PS2` Seconde chaîne pour une attente de la suite d'une commande sur une seconde ligne (second prompt). Par défaut `PS2` contient `>` . Le shell utilise cette seconde chaîne pour les commandes dont la syntaxe impose le passage à la ligne, par exemple les structures de contrôle utilisées interactivement.

`PWD` Contient le chemin du répertoire courant, c.à.d. Le répertoire dans lequel on se trouve actuellement.

2.3.10 Exportation des variables

Les variables utilisateur ou variables shell sont maintenues en interne par le shell courant. Cela signifie que lorsque ce shell crée un shell fils ces variables internes ne sont pas transmises.
Par exemple si l'on lance un nouveau shell par la commande `bash`, dans ce nouveau shell, on ne dispose plus des variables qui ont été définies dans le père. De même, si le shell fils exécute une procédure de commande, dans la procédure les variables du père ne sont pas définies.

Pour transmettre les variables aux shells fils il existe la commande `export` en Bourne shell ou `typeset -x` en bash qui déclare les variables qui seront transmises à tous les fils. Les variables exportées feront donc automatiquement partie de l'environnement des shells fils.

2.3.11 Les commandes `export` et `typeset -x`

Syntaxe

```
export [noms[=valeur]....]
typeset -x [noms[=valeur]....]}
```

Description

Exporte les variables qui sont placées dans la liste des arguments. On peut initialiser les variables au moment de l'exportation. Sans nom de variables, la commande liste l'ensemble des variables exportées du shell courant. Toute variable exportée par un shell père est elle-même du type `export` pour le shell fils. Si celui-ci génère à son tour un troisième shell, celui-ci disposera de l'environnement exporté par son grand père et aussi des variables éventuellement exportées par son père.

Pour être complet en ce qui concerne la transmission de l'environnement d'un shell à l'autre il est important de noter que le passage de l'environnement se fait toujours du père vers le fils. Il n'y a jamais de transmission des modifications faites dans le contenu des variables exportées ou non par un fils vers le père. La seule façon pour faire passer une information vers le père est de positionner le code de retour. (256 codes seulement). Ce point est lié à la sécurité du système. Grâce à ce choix, tous les problèmes pouvant survenir dans un fils n'altèrent pas le fonctionnement du père.

Nota

Le Bourne shell impose de refaire la commande `export` à chaque modification de la variable, sinon la modification n'est pas vue dans les fils. Sous le shell bash, une seule commande export est suffisante. Les modifications sont propagées normalement aux fils.

2.3.12 Mise en place d'un environnement

La mise en place de l'environnement des variables prédéfinies et utilisateur est faite par le shell de connexion.

Pour cela, shell donne tout d'abord des valeurs par défaut aux variables prédéfinies les plus importantes comme `PATH, PS1` et `PS2`. Les variables comme `HOME` sont normalement fixées par le programme de connexion `login`. Ensuite l'environnement obtenu dépend des fichiers de configuration qui existent sur le système. Le shell exécute le fichier `/etc/profile` qui contient les affectations et commandes par défaut mises en place par l'administrateur du système.

Juste avant la lecture de la première commande shell exécute le fichier `.profile` placé dans le répertoire de l'utilisateur. Ce fichier contient les affectations et commandes souhaitées et mises en place par l'utilisateur. Ce fichier peut éventuellement être absent. C'est dans le fichier `.profile` que l'on placera par exemple le "prompt" choisi. Il faut également penser à exporter dans le fichier `.profile` les variables que l'on souhaite pouvoir utiliser dans un sous-shell ou une procédure. Par exemple on placera dans le fichier `.profile` la commande d'exportation des variables shell suivantes :

```
export HOME PATH PS1 PS2
```

Sous le shell `bash` on peut définir une variable `ENV` contenant le nom d'un fichier à exécuter à la connexion en plus du `.profile`. Par exemple, on peut lire un fichier `.bashrc` dans le répertoire de connexion en définissant `ENV=~/.bashrc`

Il existe une commande interne `"."` qui permet de faire prendre en compte les modifications du fichier `.profile` par le shell courant sans se déconnecter.

Syntaxe

```
. fichier
```

Description

Cette commande interne (point) lit et exécute le fichier directement sans générer un fils. Le shell courant lit donc les variables placées dans le `.profile` par exemple.

Exemple

```
. .profile
```

2.3.13 Visualisation de l'environnement

Les variables déclarées export sont visualisées par la commande `env` ou `typeset -x` en `bash`. Les variables non exportées ne sont pas visualisées par cette commande. Pour visualiser la liste complète des variables on utilisera la commande `set` ou `typeset` en `bash`.

On rappelle ici que la commande `readonly` seule donne la liste des variables protégées en écriture, tandis que la commande `export` sans argument liste les variables exportées.

2.4 Gestion des commandes

2.4.1 Enchaînement des commandes

Le shell accepte la combinaison des commandes de diverses manières. Avant d'expliciter ces combinaisons il convient de mieux définir la notion de commande. Une **commande simple** est une suite de mots séparés par des blancs. Le premier mot est le nom de la commande. Les redirections sont analysées par le shell et ne font pas partie de la commande.

Une commande est une commande simple ou une commande de contrôle comme les tests ou boucles. Un **tube** (*pipeline*) est une commande ou une séquence de commandes séparées par des barres verticales. Dans un tube, la sortie standard de chaque commande est raccordée à l'entrée de la suivante, sauf pour la dernière. Chaque commande du tube est lancée séparément par le shell, avec synchronisation des commandes pour que les informations passées par le tube ne soient pas perdues. Le shell attend la fin de la dernière commande. Le statut de sortie est celui de la dernière commande. (La première commande est à gauche du tube et la dernière à droite.)

Le regroupement des commandes constitue ce que l'on nomme une liste de commandes. La liste est un tube ou une séquence de tubes séparés par des caractères spéciaux qui spécifient la façon dont sont enchaînés les tubes.

Exécution séquentielle. Lorsque les tubes sont séparés par des `;`, l'exécution est séquentielle. Le shell attend la fin du tube précédent avant d'exécuter le suivant. C'est par exemple le cas lorsque l'on place plusieurs commandes sur la même ligne, séparées par des points-virgules.

Exécution asynchrone et en arrière plan. Nous allons voir le lancement et le contrôle des commandes en arrière plan.

Lancement. Dans le cas où les tubes sont séparés par des caractères `&` le shell lance l'exécution du premier, puis des suivants sans se préoccuper du synchronisme et donc en particulier sans attendre la fin du premier tube.

Dans le cas particulier où la ligne se termine par un `&` le shell se comporte de la même façon, il n'attend pas la fin du tube pour revenir en attente de commande, puisqu'il n'y a pas de tube ou commande après le `&`.

On a donc lancé un processus qui s'exécute en tâche de fond, tandis que le shell de connexion reste disponible pour l'utilisateur.

Contrôle des tâches. A la différence du Bourne shell, le bash utilise des commandes et des messages pour contrôler les taches de fond :

- Au lancement d'une tâche de fond on obtient le message suivant :

  `[n] PID ...`

  Le nombre `n` est un entier qui numérote les tâches de fond en cours d'exécution. (La tâche la plus ancienne porte le numéro `1`). Ce numéro est suivi de l'identification (`PID`) du ou des processus du tube lancé en tâche de fond. Par exemple :

  `[1] 524`

  Notez également que la variable `$!` contient toujours le numéro de processus de la dernière tâche de fond (fonctionne également en Bourne shell).

- Le shell suit les changements d'état des processus. Il indique avant l'impression du prompt les tâches de fond qui viennent de se terminer ou d'être suspendues, cette indication comporte le numéro d'ordre du shell, le PID et le nom de la tâche (voir ci-dessous).

- Le shell ne termine pas à la première commande de sortie. Si des tâches sont en cours d'exécution, le message `You have stopped (running) jobs` est affiché. L'utilisateur peut forcer la sortie par une seconde commande ou utiliser la commande `jobs` pour lister ces tâches.

- Une tâche lancée directement peut être suspendue. La touche `CTRL/Z` suspend la tâche en cours d'exécution au premier plan. Le processsus est en hibernation, il peut être relancé par une commande du shell `bg` ou `fg`.

2.4.2 Changement de noms des commandes (alias)

Les alias sont une fonctionnalité du shell utilisé pour les besoins suivants :

- Renommer et/ou abréger la frappe des commandes.
- Faciliter la recherche des commandes les plus usuelles en conservant le chemin d'accès complet de ces commandes. Ces alias sont dit "*tracked aliases*" dans le manuel.

Les alias sont définis à l'aide de la commande interne `alias`.

2.4.3 Interruptions des commandes

Les **interruptions** sont des évènements extérieurs aux processus. Il sont aussi connus sous Linux par le terme **signaux**. Ces interruptions asynchrones sont généralement utilisées pour les besoins suivants :

- Interrompre une commande depuis le clavier par un code connu (`CTRL/C`).
- Permettre à Linux de tuer une tâche présentant des erreurs de programmation donnant des accès mémoires interdits.
- Prévenir les tâches qu'elles doivent terminer, soit par exemple à la demande du super utilisateur, soit pour un arrêt de la machine.

L'action résultant d'une interruption dépend de la façon dont le processus va traiter le signal. Par défaut la réception d'un signal provoque la fin du processus. On peut décider dans un programme de ne pas mourir à la réception d'un signal pour des raisons propre au programme. Par exemple un éditeur ne doit pas terminer si la touche `CTRL/C` est appuyée par erreur.

Le shell, comme tout processus peut recevoir des signaux. Pour le shell, ou pour les commandes exécutées par le shell, les signaux résultent en général d'une action au clavier de l'utilisateur. Les signaux sont numérotés sous Linux à partir de 0. Le shell permet également de les nommer par un mnémonique donné en majuscules.

La liste des numéros et des mnémoniques est obtenue sous shell par la commande `kill -l`. Voici une liste des signaux plus particulièrement envoyés du clavier ou utilisés par l'utilisateur pour terminer une commande. Pour les signaux provenant du clavier, le caractère déclenchant l'envoi du signal est donné :

02	Signal `INTR`	Touche `CTRL/C`.
03	Signal `QUIT`	Touche `CTRL/\`.
09	Signal `KILL`	Ce signal *termine toujours* un processus.
15	Signal `TERM`	Demande au programme de se terminer proprement.
17	Signal `STOP`	Est utilisé pour suspendre une tâche.

Le shell qui travaille en mode interactif, c'est à dire lié à un terminal, récupère les signaux et les ignore. En revanche les processus fils de ce shell reçoivent les signaux normalement. Une commande lancée par l'utilisateur sera donc interrompue par un signal `INTR` par exemple. Si la commande a généré un sous-shell, celui-ci ne détourne pas les

signaux. Une procédure de commande sera donc interrompue par un signal.

Lorsque la commande est lancée en arrière plan (opérateur `&` en fin de ligne) celle-ci ne reçoit pas les signaux que l'on tape depuis le clavier.

2.4.4 Commaned `kill`

La commande `kill` est utilisée pour envoyer directement des signaux aux processus.

Syntaxe

```
kill [-signal] numéros ...
```

Description

La commande `kill` envoie le signal donné en option aux processus dont les numéros sont précisés en arguments. Par défaut, si le processus destinataire ne gère pas le signal, la commande `kill` aura donc pour effet de supprimer le processus. `signal` est soit le numéro de signal, soit le mnémonique connu du shell (voir ci-dessus).

Par défaut le signal vaut `15`, soit le mnémonique `TERM`. Ce signal devrait être toujours utilisé avant le signal `9` (`KILL`) pour donner une chance au processus de terminer correctement. Le signal `-9` est le plus sévère. Un processus est toujours supprimé immédiatement par ce signal.

Exemples

```
kill 2046 3089
```

Envoie le signal 15 aux processus 2046 et 3089.

```
kill -KILL 2046 3089
```

Envoie le signal `9` aux même processus s'il ne veulent pas terminer avec le signal `15`.

2.4.5 Ordre d'analyse d'une commande

Le fait que le shell soit un interpréteur de commande induit parfois des effets de bord lorsque l'on imbrique les mécanismes de substitution. En effet, l'évaluation d'une commande est faite dans un ordre précis. On trouvera ici quelques indications permettant de gérer ces difficultés. L'analyse d'une liste a lieu dans l'ordre suivant :

- Le shell lit l'entrée standard jusqu'à trouver le délimiteur de fin de ligne ou un séparateur de commande (`;` `&` `&&` `||`). Les commandes restantes (tubes) seront analysées en fonction de la préséance de ces séparateurs.
- Les tubes sont analysés, on obtient une liste de commandes simples.
- Les alias sont substitués.
- Les substitutions "tilde" sont effectuées.
- Les variables sont remplacées par leurs valeurs.
- Les substitutions de commande sont effectuées dans les chaînes de caractères.

– Le shell analyse les redirections de la commande, les traite puis les élimine de la ligne de commande.
– Le shell traite la génération des noms de fichiers à partir des caractères spéciaux. On prendra garde à la limitation de la longueur de la commande (de l'ordre de 5120 caractères). En effet un modèle de fichiers comme `*` peut conduire à une liste de fichiers très longue en fonction du contenu du répertoire courant.

La commande et les arguments séparés par les blancs sont reconnus puis la commande est exécutée par le shell.

On rappelle que la recherche de la commande est faite dans l'ordre des catégories suivantes : Commandes internes → Fonctions du shell → Commandes externes.

Autrement dit, le shell recherche si la commande est interne ; dans ce cas il l'exécute directement. Sinon il recherche si la commande correspond à un nom de fonction connu, si cela est le cas la fonction est exécutée par le shell. Enfin si ces deux recherches ont échoué, le shell tente d'exécuter la commande placée sur disque dur.

3 Les procédures de commande

3.1 Généralités

Il existe deux façons différentes de demander l'exécution d'une procédure shell. La procédure normale est de taper le nom de cette procédure. Dans ce cas il faut que le fichier soit un exécutable. Il faut donc changer le mode du fichier après sa création par l'éditeur (par exemple par `chmod u+x fichier`). La seconde procédure de lancement consiste à taper la commande `bash` suivie du nom de la procédure et de ses arguments. Dans ce cas on demande explicitement l'exécution d'un shell comme n'importe quelle autre commande, et on lui transmet des arguments, à savoir le nom du fichier de commande à lire et les arguments de ce fichier. Le fichier n'a donc pas besoin d'être exécutable dans ce cas.

Nota

Il est possible de choisir le type de shell exécutant une commande en plaçant la chaîne `#!` suivi du chemin d'accès au shell dès le début du fichier. Par exemple : `#!/usr/bin/bash`

3.2 Variables prédéfinies

Les variables prédéfinies sont un troisième type de variables que l'on utilise plus particulièrement dans les procédures de commandes. Elles informent l'utilisateur sur l'état du shell ou des commandes qui sont ou ont été exécutées. Ces variables sont exclusivement définies et manipulées par le shell. L'utilisateur peut seulement les lire, elles seront donc toujours précédées du `$`. Les principales variables prédéfinies sont les suivantes :

`#` Cette variable contient le nombre d'arguments passés au shell en dehors du nom de la procédure elle-même. `$#` contient donc le rang du dernier argument de la ligne de commande. Cette valeur n'est utilisable qu'à l'intérieur d'une procédure shell pour connaître le nombre de paramètres de l'appel.

`?` Cette variable contient le statut de sortie de la dernière commande exécutée (appelé aussi code de retour, ou code de sortie ou value). Le contenu est une chaîne de caractères décimaux positionné par la commande elle-même. Ce code peut varier entre `0` et `255` (codage sur un octet). Pour le shell, un code nul est 'vrai', il n'y a pas eu d'erreur. Les autres valeurs sont fausses, correspondent à une erreur. On se reportera à chaque commande pour connaître la signification de ces codes de retour.

`$` Cette variable contient le numéro du processus en cours d'exécution. En l'occurrence le numéro unique d'identification du shell qui interprète les commandes ou exécute une procédure. Ce numéro est unique car le numéro d'un processus mort n'est pas réutilisé. Sous le shell de connexion la commande `echo $$` donnera son numéro d'identification. En général on se sert de cette variable pour composer un nom de fichier unique et temporaire dont on a besoin dans une procédure. (placer ce fichier dans `/tmp` et penser à le retirer après usage).

`!` Variable contenant le numéro du dernier processus lancé en tâche de fond. Le contenu n'est changé que lorsque l'on lance un nouveau processus, cette variable n'indique pas l'état d'avancement des processus. (on utilisera la commande `ps` pour visualiser les taches de fond en cours d'exécution).

`-` Cette variable contient la liste des options (flags d'exécutions, section suivante) du shell courant.

3.3 Passage des paramètres (paramètres de position)

Les paramètres d'une procédure shell sont les arguments attendus par cette procédure. Le shell lit ces arguments et transmet leur contenu à la procédure au moyen des paramètres de position. Les paramètres de position sont des variables que l'on peut lire dans la procédure appelée. Le passage de paramètres à une procédure est fait grâce à ces variables : le shell numérote chaque champ de la ligne de commande de 0 à 9. A chaque champ est associé un paramètre de position appelé `$0` à `$9`.

Le premier champ est le nom de la commande, ce paramètre `$0` n'est pas modifiable par l'utilisateur. Son intérêt est de connaître à l'intérieur de la procédure le nom qui a été invoqué. Par exemple on peut choisir d'écrire une procédure `p1` et de créer un fichier lié `p2` sur `p1`. Dans la procédure `p1` on teste `$0` pour savoir quelle action entreprendre en fonction de l'appel par `p1` ou `p2`. On dispose également de la variable `$*` qui contient la liste complète des paramètres de position, sauf le paramètre `$0`.

Nota

Le bash dispose d'un nombre non limité de paramètres de position. Pour les paramètres supérieurs à 9 il faut simplement utiliser la notation de substitution explicite `${nombre}`. Par exemple `${10}` est le 10ème argument. Il existe une subtilité en Bash pour exprimer la liste de tous les argument d'appel : on dispose de `$@` en plus de `$*` :

– La notation `$*` est remplacée par `"$1 $2 $3 $4 ..."`, soit 1 argument.

– La notation `$@` est remplacée par `"$1" "$2" "$3"` ..., soit n arguments.

3.4 Les options du shell (flags d'exécution)

Les options du shell ont la même fonction que pour toute autre commande, c'est à dire modifier le comportement de la commande, en l'occurrence le shell. Les options sont connues sous le nom de flags d'exécution. Les options sont généralement manipulées par la commande `set` qui permet de positionner interactivement les options du shell courant. On peut aussi utiliser la commande `set` dans les procédures de commandes. Enfin, il est également possible de placer ces options derrière la commande bash au lancement, par exemple :

```
bash -xv procedure
```

Cette dernière méthode ne transmet pas les fonctions et les alias exportés, elle n'est donc pas toujours possible.

3.4.1 La commande `bash`

Syntaxe

```
bash [-options] [-o options] ... [-c chaîne] [arguments ...]
```

Description

Le shell est lancé directement. Sans argument, le shell est interactif et utilise les entrées/sorties standard à l'écran. Les arguments sont le nom et les arguments d'un fichier de commande à exécuter. A noter que le chemin d'accès `PATH` est utilisé pour retrouver ce fichier.

Options

Les options de lancement sont celles de la commande `set`, sauf pour la suivante : `-c chaine`, qui lit les commandes à exécuter dans la chaine donnée en argument (attention à placer des cotes).

3.4.2 La commande `set`

Syntaxe

```
set [options] [-o options] ... [arguments ...]
```

Description

La commande manipule les options du shell. Une option est mise en place par le moins (-) placé devant et retirée en la faisant précéder d'un plus (+). On rappelle que la commande a déjà été présentée pour réaliser les actions suivantes :

- Sans argument, la commande liste les variables du shell
- Sans options ou avec l'option -, la commande vide les paramètres de positions, puis réaffecte à partir des arguments précisés.

Principales Options

Certaines options peuvent être positionnées par une lettre ou par un ou plusieurs mnémoniques placés derrière l'option

`-o` Dans ce cas les deux syntaxes sont données séparées par une virgule.

`-a, -o allexport` Les variables définies à partir de la commande `set` sont exportées automatiquement.

`-e, -o errexit` Arrêt du shell si une commande donne un statut de sortie faux, c'est à dire non nul. Cette option est utile pour interrompre la lecture et l'exécution d'une procédure si une erreur se produit dans celle-ci. Après la phase de mise au point, il est préférable de récupérer et traiter les erreurs dans la procédure même.

`-f, -o noglob` Empêche l'expansion automatique des noms de fichiers.

`-h, -o trackall` Chaque commande est automatiquement un alias "tracked" (voir la section alias).

`-k, -o keyword` Ce drapeau (flag) permet de considérer tous les arguments de la forme `variable=valeur` comme des paramètres mots-clés. Cette option est prévue pour supporter les anciens scripts, elle est déconseillée.

`-m, -o monitor` Cette option par défaut permet le contrôle des tâches de fond. On peut donc revenir à la situation du Bourne shell pour les tâches de fond en positionnant l'option `+m`.

`-n, -o noexec` Le shell lit les commandes mais ne les exécute pas. Utile pour la mise au point de la syntaxe des commandes dans une procédure par exemple.

`-t` Le shell lit et exécute une seule commande puis termine directement. Cette option est typiquement utilisée pour exécuter une commande shell depuis un programme écrit en langage C.

`-u, -o nounset` Ce flag permet de générer une erreur lorsque des variables indéfinies sont utilisées en substitution. Cela permet de vérifier globalement que les différentes variables ont un contenu défini.

`-v, -o verbose` Demande l'impression des lignes de commandes qui sont lues. On peut suivre pas à pas l'exécution d'une procédure de cette façon. Ceci facilite le repérage des erreurs diagnostiquées par le shell.

`-x, -o xtrace` Demande l'affichage des commandes qui sont exécutées et de leurs arguments. Seules les commandes exécutées sont affichées contrairement à l'option `-v` ou toute commande lue est visualisée. Les commandes de contrôle du déroulement de la procédure (boucle, test) ne sont pas tracées. Chaque commande est précédée d'un "prompt" contenu dans la variable `PS4`, par défaut le prompt est `+ `.

`-o bgnice` Option par défaut, lance les tâches de fond à une priorité plus faible.

`-o ignoreeof` Le shell ne termine plus sur un `CTRL/D`, la commande `exit` doit être utilisée. Permet par exemple d'exécuter un fichier de commande en sortie en faisant un alias sur exit (on peut aussi utiliser la commande `trap`).

`-o noclobber` Évite l'écrasement des fichiers par une redirection. On peut écraser dans ce cas avec le signe de redirection `>`.

`-o vi` Rappel des commandes en mode `vi`.

Exemple

```
set -xv
echo Cette partie de programme est tracée
set +xv
echo Plus de trace
```

3.5 Commandes de test

Pour utiliser pleinement les structures de contrôle présentées ci-dessous, l'utilisateur dispose de plusieurs commandes. La première commande test est une commande disponible en Bourne shell. Bien que la syntaxe de cette commande soit délicate, Il est utile de la connaître en raison de l'utilisation qui en est faite dans les programmes existants. L'usage des commandes du `bash` simplifie l'écriture des commandes et donne des résultats plus rapides (comandes internes).

3.5.1 Syntaxe standard du Bourne shell : la commande `test`

Syntaxe

```
test expression ou [ expression ]
```

Dans la seconde forme l'expression est directement placée entre crochets (attention il ne s'agit pas d'une expression facultative, et les espaces encadrant sont impératifs).

Description

La commande évalue l'expression fournie comme argument, et renvoie un statut de `0` si celle-ci est vraie ou différent de `0` dans le cas contraire. On prendra garde aux erreurs de syntaxe conduisant à un manque d'arguments pour la commande, car dans ce cas on obtient également un statut non nul. Le test n'est donc plus correct. On peut par exemple entourer les variables utilisées en argument par des guillemets doubles, de cette façon on manipule un argument vide sans produire d'erreur de syntaxe, même si la variable n'est pas définie.

Une expression est un ensemble de primitives ayant une valeur logique vraie ou fausse. Les primitives sont groupées ensemble à l'aide du "et" logique, du "ou" logique, de la négation et des parenthèses pour obtenir l'expression logique finale. Ces opérateurs sont les suivants :

`!` Négation d'une expression.

`-a` Et logique.

`-o` Ou logique (le "et logique" à une préséance plus grande).

`\( expr\)` Parenthèses pour regrouper les expressions (il faut leur donner leur sens littéral par l'anti-slash, sinon le shell les interprète. L'évaluation a lieu de gauche à droite.

Les primitives suivantes sont utilisées avec un nom de fichier séparé de l'option par un blanc :

`-r` Vrai si le fichier existe et est accessible en lecture pour l'utilisateur.

`-w` Vrai si le fichier existe et est accessible en écriture pour l'utilisateur.

`-x` Vrai si le fichier existe et est accessible en exécution pour l'utilisateur.

`-f` Vrai si le fichier existe et est un fichier normal.

`-d` Vrai si le fichier existe et est un répertoire.

`-c` Vrai si le fichier existe et est du type spécial caractère.

`-b` Vrai si le fichier existe et est du type spécial bloc.

`-s` Vrai si le fichier existe et a une taille non nulle.

Les primitives suivantes sont utilisées pour tester les chaînes de caractères (on peut utiliser des variables comme chaîne de caractères) :

`-z s1` Vrai si la chaîne s1 est de longueur nulle.

`-n s1` Vrai si la chaîne s1 contient au moins un caractère.

`s1 = s2` Vrai si les deux chaînes sont égales (**attention aux blancs encadrant le signe égal**).

`s1 != s2` vrai si les deux chaînes sont différentes.

On prendra garde au fait qu'une variable non initialisée ou contenant des blancs sera remplacée par une chaine vide ou par des blancs. Il en résultera une erreur pour la commande `test` puisque l'argument correspondant à cette variable ne sera pas trouvé. On peut placer la variable entre guillemets pour forcer la reconnaissance de cet argument par le shell.

Enfin il existe des primitives qui analysent les chaînes de caractères comme des entiers. Le shell manipule toujours les variables comme des chaînes de caractères, simplement pour certaines commandes la chaîne de caractères prend une signification de nombre entier. Les tests sur les entiers ont la syntaxe suivante :

```
n1 operateur n2
```

Les opérateurs possibles sont les suivants :

`-ne` différent,

`-gt` supérieur,

`-ge` supérieur ou égal,

`-lt` inférieur,

`-le` inférieur ou égal,

`-eq` égal.

Exemple

```
if [ -f $1 -a -r $1 ]
    then cat $1
fi
```

Teste si le fichier est un fichier ordinaire et peut être lu.

Les commandes `true` et `false`

La commande `true` retourne un code de `0`, elle est toujours vraie. La commande `false` est toujours fausse.

Exemple

```
while true
    do echo 'Pour interrompre tapez DEL'
done
```

Nota

Sous `bash`, `true` et `false` sont des alias.

Spécificités du bash

En plus des primitives présentées dans le paragraphe précédent, le shell `bash` dispose d'autres primitives qui complètent la commande `test`. On consultera la liste complète dans le manuel `man bash`. On trouvera ci-dessous un résumé des primitives les plus utiles introduites par le `bash` :

`-L fichier` Vrai si le fichier est un lien symbolique.

`f1 -nt f2` Vrai si le fichier `f1` existe et est plus récent que `f2`.

`f1 -ot f2` Vrai si le fichier `f1` existe et est plus ancien que `f2`.

`f1 -ef f2` Vrai si les fichier `f1` et `f2` sont des liens physiques.

Deux nouvelles primitives suivantes ont été introduites pour tester les chaînes de caractères :

`s1 < s2` Vrai si la chaîne `s1` est placé avant `s2` dans l'ordre de la table ASCII.

`s1 > s2` Vrai si la chaîne `s1` est placé après `s2` dans l'ordre de la table ASCII.

En outre, la commande `let` évalue l'expression fournie comme argument, et renvoie un statut de `0` si celle-ci est vraie ou différent de `0` dans le cas contraire. Elle permet donc d'utiliser les opérateurs logiques du langage C à savoir :

`!` négation d'une expression.

`&&` et logique.

`||` ou logique (le "et logique" à une préséance plus grande).

Exemple

```
[ $nom1 > $nom2 ]
```

Teste si le `nom1` est avant `nom2` par ordre alphabétique

4 Structures de contrôle

Les structures de contrôle permettent, comme pour tout langage de programmation, de modifier l'enchaînement des instructions ou des commandes en fonction des besoins. On dispose sous le shell des quatre types d'instructions autorisant la programmation structurée suivants :

- boucle `for`
- choix multiples `case`
- test `if`
- boucles "tant que" `while` et `until`

On peut également citer ici, bien que ce ne soit pas à proprement parler une structure de contrôle, le caractère `#` qui place une ligne en commentaire dans un fichier de commande. Sous `bash` il existe également une boucle `select` spécialement construite pour la saisie des choix d'un menu. Le shell reconnaît les mots spéciaux définissant les structures de contrôle seulement dans le cas où ils sont les premiers sur la ligne. Il est donc possible d'indenter les instructions, mais on ne peut pas par exemple placer une structure de contrôle complète sur une seule ligne. Toutefois l'utilisation du `;` permet dans beaucoup de cas de contourner la difficulté, en effet le `;` est équivalent à un "return".

4.1 La structure `if`

Syntaxe

```
if liste1
    then liste2
   [else liste3]
fi
```

Description

Pour le shell, le test d'une condition est toujours fait sur le code de retour d'une commande. On ne peut pas tester directement une variable. Le shell exécute donc les commandes placées dans `liste1`, puis teste le code de retour de cette liste de commandes. La `liste1` est toujours exécutée. On utilise généralement une des commandes de test déjà présentées qui ne produisent pas d'autre action que de retourner un code pour le test dans la structure de contrôle. Si la `liste1` donne un code de retour de `0`, c'est à dire un résultat vrai, la condition `then` est choisie et la liste de commandes `liste2` est exécutée. Dans le cas contraire, la `liste1` est fausse et la condition `else` est choisie, `liste3` est exécutée. La condition `else` est facultative ; si le `else` n'est pas présent et si la condition est fausse on sort de la structure en ayant exécuté `liste1` seulement.

Exemples

- Exemple avec `chmod`

```
echo "Donner le nom de fichier qui aura tous les droits"
read nomfic
if chmod 777 $nomfic
    then echo "le fichier $nomfic a maintenant tous les droits"
    else echo "fichier introuvable ou ne vous appartenant pas"
fi
```

– Exemple avec la commande de test du bash

```
ln -s t1 t2
if [[ -L t2 ]] ; then
    echo lien symbolique
fi
rm t2
ln t1 t2
if [ t1 -ef t2 ] ; then
echo lien physiques
fi
```

– Exemple avec la commande de test d'expression entières `let`

```
if let " 0 "; then
    # 0 est faux en C, le code de retour est 1, on ne passe pas dans le then :
else
    echo ? l'expression 0 est fausse ?
fi
typeset -i i=10
if let ? i == 10 ? ; then
    echo La variable i vaut 10
fi
```

Il est possible bien entendu d'imbriquer des structures if entre elles avec une structure de la forme suivante :

```
if
    then .....
    else
        if.....
        .......
    fi
fi
```

Il existe une variante de la structure if pour réaliser facilement des tests imbriqués :
Syntaxe :

```
if liste1
    then liste2
    elif liste3
        then liste4
       [else liste5]
fi
```

Description

Lorsque la `liste1` est fausse on exécute la condition `elif` de la même façon que s'il s'agissait d'un test `if` simple. Il est possible également de continuer la structure en plaçant plusieurs `elif` en série pour réaliser une cascade de tests. Pour cela on remplace le dernier `else` par une structure `elif, then, else` de la façon suivante :

```
if
    then .....
    elif
        then ....
        elif .....
            then .......
            else .....
fi
```

Nota

La commande qui suit le `then` est obligatoire, dans cas où la seule action à faire est dans le `else`, on peut utiliser la commande interne `:` qui est la commande nulle (ne fait rien). Elle renvoie simplement un code de `0`.

Exemple

```
if let " i < 0 " ; then
    echo i est négatif
    elif let " i == 0 " ; then
        echo i est nul
    else
        echo i est positif
fi
```

4.2 La structure case

Syntaxe

```
case mot in
    chaine1 [ | chaine2 ... ] ) liste1;;
    chaine3 [ | chaine4 ... ] ) liste2;;
    ......
esac
```

Description

La chaîne de caractère `mot` (qui peut être le contenu d'une variable) est comparée aux chaînes de caractères placées dans la liste des cas en partant du haut. Lorsqu'une correspondance est trouvée shell exécute la liste de commande placée à la suite de cette valeur, puis termine la structure `case`. Il y a donc une seule liste de commande exécutée pour chaque utilisation de l'instruction `case`. On peut réaliser un "ou" entre plusieurs

chaînes différentes pour lesquelles on souhaite exécuter la même liste de commande. Pour cela il faut séparer les chaînes par une seule barre verticale.

Les chaînes peuvent contenir des caractères spéciaux pour réaliser un masque comme pour la génération de modèles de fichiers. La syntaxe est la même que dans ce dernier cas. Par exemple l'astérisque remplace n'importe quelle chaîne. On placera par exemple un `*` en dernier cas pour réaliser une action par défaut puisque, quelle que soit la chaîne nom, elle correspondra toujours à `*`. On prendra garde à l'ordre d'évaluation des cas ; si par exemple l'astérisque est placée en tête des cas la première liste sera toujours exécutée et les cas suivants ne seront jamais examinés.

Les chaînes peuvent aussi être des variables ou contenir des variables, avec la syntaxe habituelle. Il est prudent de tester les résultats obtenus, car pour des manipulations complexes l'ordre d'évaluation des commandes par le shell a une grande importance. La syntaxe habituelle s'applique pour les chaînes, en particulier si la chaîne contient des blancs il faut la placer entre guillemets. Le statut de sortie du case est celui de la dernière liste exécutée et `0` si aucune commande ne l'a été.

Exemple

```
# Cette procédure donne le prénom du mathématicien dont le nom est passé en argument
case $1 in
    Euler ) prenom=Leonard;;
    Galois ) prenom=Evariste;;
    "de la Vallee Poussin" ) prenom=Charles;;
    Hilbert ) prenom=David;;
    Serre ) prenom="Jean Pierre";;
    * ) prenom=inconnu;;
esac
echo "Le prénom du mathematicien $1 est $prenom"
```

4.3 Les structures `while` et `until`

Syntaxe

```
while liste1
do
    liste2
done
```

Description

Comme pour l'instruction `if`, le shell teste le code de retour de la `liste1`. Tant que la `liste1` est vraie (code de retour `0`) la `liste2` est exécutée. Si `liste1` est fausse on saute à la commande suivant le mot réservé `done`. La boucle se poursuit indéfiniment si l'on ne prend pas garde à ce que `liste1` devienne fausse par les actions réalisées dans `liste1` ou `liste2`. Le code de retour du `while` est `0` (vrai) lorsque l'on sort normalement par le test de la `liste1`. Sinon le code de retour est celui de la dernière commande exécutée.

Syntaxe

```
until liste1
do
    liste2
done
```

Description

Le principe est le même que pour le `while`, simplement le test est inversé, la boucle se poursuit jusqu'à ce que `liste1` devienne vraie (ou tant que `liste1` est fausse). Pour l'instruction `while` comme pour `until`, le code de retour est celui de la dernière commande exécutée.

Exemple

La procédure suivante liste les répertoires en amont du répertoire courant (on doit avoir un fichier Linux dans la racine) :

```
until ls Linux 2>/dev/null
do
    pwd
    cd ..
done
echo "Vous êtes arrivé dans le repertoire root"
Cet exemple boucle 100 fois:
typeset -i i=0
while let " i < 100 "
do
    echo $i
    i=i+1
done
```

4.4 La structure `for`

Syntaxe

```
for variable in mots
do
    liste1
done
```

Description

La boucle est exécutée autant de fois qu'il y a de mots dans la liste `mots`. À chaque passage, la variable de boucle `variable` est assignée à la chaîne de caractères du mot correspondant au rang de ce passage. Ensuite la liste de commande `liste1` est exécutée. Les séparateurs de mots sont des blancs ou des tabulations. Lorsque l'on omet ce qui suit le nom de variable, l'instruction est équivalente à :

`for variable in $*`

On va donc boucler autant de fois qu'il y a d'arguments dans la commande, avec un contenu de `variable` qui vaudra successivement `$1`, `$2`, etc.

Exemple

Voici une procédure qui concatène tous les fichiers donnés en arguments :

```
for nomfic
do
    cat $nomfic >>grosfic
done
```

La procédure suivante donne le type de tous les fichiers du répertoire courant (on pourrait aussi faire `file *)` :

```
for fichier in * ; do
    echo "le fichier $fichier est du type suivant:" ; file $fichier
done
```

4.5 Instructions de débranchement, `break` et `continue`

Ces instructions modifient l'exécution des boucles `while`, `until` et `for`. Elle doivent être placées entre `do` et `done`. L'instruction `break [n]` provoque la fin de l'exécution d'une boucle, c'est à dire la sortie après le `done`. Dans le cas de boucles imbriquées, on peut préciser un nombre `n` qui permet de sortir des `n` boucles les plus intérieures. Par défaut on sort de la boucle courante.

L'instruction `continue [n]` saute la liste de commande située entre elle et la fin de boucle `done`. La boucle se poursuit à l'itération suivante. Si `n` est précisé on poursuit l'itération suivante de la boucle de niveau `n`. Ces instructions sont généralement associées à un test intérieur à la boucle, faute de quoi certaines parties des boucles ne sont jamais exécutées.

4.6 Instruction `select`

Syntaxe

```
select variable [ in mot1 mot2... ]
do
    liste1
done
```

Description

La commande select imprime sur la sortie d'erreur la liste des mots précédés par un numéro d'ordre. Si les mots sont omis, les paramètres de position sont utilisés. La commande boucle sur la séquence suivante :

– Le "prompt" contenu dans la variable shell `PS3` est affiché (par défaut `#?`), puis la réponse de l'utilisateur (le numéro d'ordre) est lue.

– Si la réponse est vide, le menu est réaffiché. Dans le cas contraire la variable est initialisé par le contenu du mot choisi ou est vidé si le choix est incorrect, puis la liste de commande est exécutée.

Nota

La réponse de l'utilisateur est disponible dans la variable shell `$REPLY`. Le corps de la boucle est toujours exécuté et il contient généralement une instruction `break` permettant de sortir du menu.

Exemple

```
c1="cas1"
c2="cas2"
c3="Sortie du menu"
select choix in "$c1" "$c2" "$c3"
do
    case $REPLY in
        1 )
        2 ) echo Vous avez choisi $c1;;
            echo Vous avez choisi $c2;;
        3 ) echo Fin!
            break;;
        * ) echo Taper un nombre entre 1 et 3 ;;
    esac
done
```

4.7 Sortie

La commande exit permet de sortir élégamment d'une procédure shell.

Syntaxe

```
exit [n]
```

Description

Lors du retour au shell père, le code de retour d'une procédure sera par défaut celui de la dernière commande exécutée dans la procédure. La commande `exit` permet de sortir d'une procédure en maîtrisant le code de retour qui peut donc différer de celui de la dernière commande exécutée. La commande `exit` peut être placée par exemple dans une boucle, associée à un test, elle permet de quitter conditionnellement la procédure sans passer par la fin de fichier.

On rappelle que le statut de sortie est la seule manière de transmettre une information vers le processus père, `exit` pourra donc éventuellement être utilisé pour coder une information vers le père. On retrouvera l'information dans le père grâce à la variable `$?`. Les codes de retour possibles vont de `0` à `255`, le codage est fait sur un octet.

Partie C

Administration système

IV – Rudiments d'administration système

1 Gestion des fichiers

1.1 Structure d'un système de fichiers

L'organisation d'un système de fichiers Linux est fondée sur une division de l'espace disque en blocs de longueur fixée à l'initialisation du disque (en général de 1024 à 4096 octets chacun). Ces blocs sont numérotés à partir de 0. Les blocs contiennent les données des fichiers et sont alloués dynamiquement en fonction des besoins.

1.2 Notion d'i-nœud (ou inode)

Linux identifie chaque fichier existant dans le système de fichiers par un numéro unique qu'on appelle numéro d'i-nœud ou "inode number". Pour chaque fichier, Linux conserve un certain nombre d'informations lui permettant d'identifier le fichier et d'accéder aux blocs contenant les données. Ces informations sont stockées dans une table nommée table des i-nœuds. L'i-nœud représente le numéro de l'entrée dans la table. L'i-noeud (index node) est une structure de données formant une référence interne d'un fichier. Elle contient principalement :

- Le type de l'entrée
- Les droits d'accès et les identificateurs des propriétaire et groupe (UID, GID)
- La taille de l'entrée
- Le nombre de liens de l'entrée
- Les dates de création et de dernière modification de l'entrée

- Les adresses des blocs utilisés (pour les fichiers disques)
- Les ressources associées (pour les fichiers spéciaux)

1.3 Droits d'accès et permissions d'un fichier

1.4 Recherche des données à partir du nom de fichier

Schématiquement, lorsqu'un utilisateur réclame l'accès à un fichier, le système Linux effectue les opérations suivantes :

- A partir du nom fourni par l'utilisateur, Linux recherche dans le répertoire ad-hoc le numéro d'inode correspondant au fichier.
- En cas de succès, Linux recherche l'inode correspondant dans la table des inodes, qui se trouve à un endroit connu dans le système de fichiers
- Après avoir vérifié que la requête de l'utilisateur est conforme aux droits d'accès du fichier, Linux isole les adresses des blocs, ce qui permet d'accéder aux données.

1.5 Installation de logiciels

Les logiciels, sous Linux, sont généralement organisés en paquets, d'extension `.deb` sur les distributions de type debian et `.rpm` sur celles de type red hat.

Pour récupérer un paquet sur internet, on utilise la commande `apt-get`

Commandes d'informations système

`uname`	Identification du système.
	`-a` : toutes les informations.
`dmesg`	Messages du noyau (et ceux du boot).
`uptime`	Durée et charge du système.
`free`	Occupation de la mémoire.
`vmstat`	Détails sur l'utilisation de la mémoire.
`ldconfig`	Valider les bibliothèques dynamiques.
`init`	Changement de niveau de fonctionnement :
	`0` : arrêt.
	`1` : mono-utilisateur,
	`3` : multi-utilisateurs mode texte,
	`5` : multi-utilisateurs mode graphique,
	`6` : redémarrer.

1.6 Partitions et systèmes de fichiers

Pour créer un système de fichiers, il faut effectuer, dans l'ordre, les opérations suivantes :

(i) Créer une table des partitions avec création d'une zone dite d'échange ou de "swap".

Cette zone sert de mémoire étendue lorsqu'il n'y a plus de place en mémoire physique pour les processus en attente.

(*ii*) Création d'un système de fichiers. Sont créés la table des i-nœud la liste et la table des blocs disque libres.

L'outil `fdisk` permet l'édition de la table des partitions. La commande `sudo fdisk -l` fournit une sortie de la forme :

Exemple de code IV.1

```
255 têtes, 63 secteurs/piste, 30401 cylindres, total 488397168 secteurs
Unités = secteurs de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
Identifiant de disque : 0x259317c3

Périphérique Amorçage  Début         Fin      Blocs    Id. Système
/dev/sdb1               2048     3999743     1998848   82  partition d'échange Linux
/dev/sdb2            4001790   488396799   242197505    5  Étendue
/dev/sdb5            4001792     7999487     1998848   83  Linux
/dev/sdb6            8001536    95889407    43943936   83  Linux
/dev/sdb7           95891456   183779327    43943936   83  Linux
/dev/sdb8          183781376   488396799   152307712   83  Linux

Disque /dev/sdc : 1014 Mo, 1014497280 octets
17 têtes, 32 secteurs/piste, 3642 cylindres, total 1981440 secteurs
Unités = secteurs de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
Identifiant de disque : 0xc3072e18

Périphérique Amorçage  Début         Fin      Blocs    Id. Système
/dev/sdc1      *          32     1981439      990704    6  FAT16
```

Un outil de haut niveau pour la gestion des partitions est `parttionMagic` ou `gparted`. *Il est*, lorsque cela est possible, *recommandé d'utiliser un outil haut niveau qui permet de minimiser les erreurs involontaires.*

Le fichier `fstab` (file systems table) est la table des différents systèmes de fichiers sur un ordinateur sous Linux : il contient une liste des disques utilisés au démarrage et des partitions de ces disques. Pour chaque partition, il indique comment elle sera utilisée et intégrée à l'arborescence du système de fichiers global (c'est-à-dire le point de montage). Le chemin complet de ce fichier est généralement `/etc/fstab`.

L'aspect d'un fichier `fstab` sera typiquement le suivant :

Exemple de code IV.2

```
# /etc/fstab: static file system information.
#
# Use 'blkid -o value -s UUID' to print the universally unique identifier
# for a device; this may be used with UUID= as a more robust way to name
# devices that works even if disks are added and removed. See fstab(5).
#
# <file system> <mount point>   <type>  <options>          <dump>  <pass>
proc            /proc           proc    nodev,noexec,nosuid  0       0
/dev/sdb6       /               ext4    errors=remount-ro    0       1
/dev/sdb5       /boot           ext4    defaults             0       2
/dev/sda5       /home           ext3    defaults             0       2
/dev/sdb7       /usr/local      ext4    defaults             0       2
/dev/sdb1       none            swap    sw                   0       0
# /espace was on /dev/sdb8 during installation
UUID=ca68f751-a846-4836-a1e6-618c9dd6f3cc /espace         ext4    defaults        0
```

Commandes diverses

`mkswap` Création d'une zone de swap : `mkswap /dev/hda2`

`swapon` Activation d'une zone de swap : `swapon /dev/hda2`
`-a` active toutes les zones de swap de `/etc/fstab`.

`swapoff` Désactivation d'une zone de swap : `swapoff /dev/hda2`

`mkfs` Création d'un système de fichiers : `mkfs.ext3 /dev/hda3`
`mkfs.vfat /dev/hda5`

`fsck` Vérification d'un système de fichiers :
`fsck.ext2 -p /dev/hda3` réparation automatique d'un système `ext2/ext3`,
`fsck.vfat /dev/hda4` vérification d'une partition Windows.

`mount` Insertion (montage) de partition dans le système :
`mount -t vfat /dev/hda4 /mnt/dos/` monter une partition Windows,
`mount -a` monter toutes les partitions de `/etc/fstab`,
`mount 192.1.1.254:/home /home/users/` Montage d'un répertoire distant par NFS.
Options avec `-o` ou dans `/etc/fstab` :
default : `rw,suid,dev,exec,auto,nouser,async`,
`remount` : changer les attributs d'un système monté,
`rw` : lecture-écriture,
`ro` : lecture seule,
`noauto` : ne pas monter automatiquement avec -a,
`nodev` : interdire les fichiers spéciaux,

	`noexec` : pas de fichiers exécutables, `nosuid` ; ignorer les bits Set-UID/GID, `sync` : écritures synchrones, `user` : peut être monté par un utilisateur.
`umount`	Démontage d'un système de fichiers : `-a` : démonte tous les systèmes dans /etc/mtab. `umount /dev/hda4`
`df`	Taux d'occupation des systèmes de fichiers montés.

Partie D

Appels systèmes

V – Processus et Threads Posix

1 Processus

1.1 Introduction

Un programme est une suite d'instructions agissant sur des données. L'ensemble est contenu dans un fichier ordinaire ayant le droit d'exécution. Un tel fichier est obtenu généralement par compilation d'un fichier source contenant la description du code dans un langage donné.

La compilation fournit un module objet qui est ensuite transformé en un module exécutable par l'éditeur de liens. Cependant pour exécuter un tel module exécutable, le noyau doit créer un nouveau processus.

> *Un **processus** est l'exécution d'une **image**. Une image la réunion de 3 zones mémoire : le segment d'instructions (Instruction segment), le segment de données utilisateur (User data segment) et le segment de données système (System data segment).*

Le segment de données système est la zone mémoire où le noyau stocke les informations telles que les descripteurs des fichiers ouverts, les variables d'environnement, le temps CPU utilisé, etc. Ce segment est exclusivement manipulé par le noyau, et ne peut être accédé qu'au moyen d'appels système spécifiques.

A chaque fois qu'un processus est créé, Linux l'identifie par un nombre entier positif qui est garanti à chaque instant comme étant unique. Ce numéro est appelé PID (process identification) et le noyau l'utilise pour contrôler le processus correspondant. Il existe un nombre maximum de processus pouvant exister simultanément dans le système. Ce nombre est fixé à la génération du système. Linux met à la disposition de l'utilisateur une variété d'appels système permettant la gestion des processus.

Le seul mode de création d'un nouveau processus sous Linux est la duplication d'un processus existant. Le processus à l'origine de la duplication est appelé le processus père, et le nouveau processus est appelé le processus fils. Le fils hérite de la plupart des attributs du père. Excepté le premier processus (le gestionnaire de la mémoire) qui est implanté en mémoire directement lors du bootstrap, tous les processus sous Linux sont obtenu par duplication, et de ce fait ont tous un ancêtre commun le processus `init` (dont le PID est 1). Les appels système `fork()`, `exec()`, `wait()`, `exit()` donnent un contrôle direct de la création, de l'exécution et de l'arrêt de nouveaux processus, autorisant un pilotage très rigoureux d'un ou plusieurs processus simultanément.

1.2 Droits des processus

Tout utilisateur autorisé à se connecter sur un système Linux a un nom de login auquel est associé un nombre entier positif appelé UID (user-ID). Quand l'utilisateur se connecte, le processus login affecte au processus qui est créé (généralement le shell de connexion) la valeur UID comme identification de propriétaire. Par la suite, les processus descendants héritent de cet UID. Chaque utilisateur appartient à un groupe qui est repéré également par un nombre entier positif noté GID. Une action similaire à celle effectuée pour l'UID est exécutée pour le GID par le processus login. Ainsi, tous les descendants du shell héritent du même GID.

Ces deux identifications sont l'UID réel et le GID réel. Deux autres identificateurs sont associés à un processus : l'UID effectif, et le GID effectif. Ils sont normalement identiques aux identificateurs réels correspondants. Ils peuvent différer si l'utilisateur exécute un fichier muni du bit set-user ID ou du set-group ID.

Par exemple, en listant les informations au format long du fichier des mots de passe `/etc/passwd` dont le propriétaire est `root`, on obtient :

```
$ ls -l /etc/passwd
-rw-r--r-- 1 root root 2028 févr. 27  2015 /etc/passwd
```

Pour pouvoir modifier son propre mot de passe, on doit avoir le droit en écriture sur ce fichier, ce qui n'est le cas que pour le propriétaire, `root`. En listant les informations au format long de la commande de modification de mot de passe `/bin/passwd`

```
$ ls -l /usr/bin/passwd
-rwsr-xr-x 1 root root 47032 juil. 15 21:29 /usr/bin/passwd
```

On note que le droit en exécution du propriétaire de la commande `/usr/bin/passwd` est `s` et non `x`, ce qui signifie que le bit de modification d'identité (set-user ID bit) est positionné. De cette manière, le propriétaire du processus ayant lancé `/usr/bin/passwd` devient `root` *le temps d'exécution de cette commande*. La modification du fichier `/etc/passwd` est alors possible.

Deux appels système, `setuid()` et `setgid()` permettent également de modifier les UID et GID effectifs. Pour déterminer les droits d'accès à une ressource, comme un fichier par exemple, c'est toujours l'UID effectif qui est utilisé par le noyau. Celui-ci exécute l'algorithme suivant pour déterminer si le droit demandé est accordé :

(*i*) Si l'UID effectif est 0 (c'est l'UID du superuser) la permission est accordée immédiatement.

(*ii*) Si l'UID effectif du processus et UID du propriétaire de la ressource à accéder sont identiques, alors les droits du propriétaire de la ressource sont utilisés pour accorder ou non la permission.

(*iii*) Si le GID effectif du processus et le GID du propriétaire de la ressource à accéder sont identiques, alors les droits de groupe de la ressource sont utilisés pour accorder ou non la permission.

(*iv*) Si ni l'UID effectif ni le GID effectif du processus ne sont identiques à ceux du propriétaire de la ressource à accéder, alors les droits du public de la ressource sont utilisés pour accorder ou non la permission.

1.3 Gestion de processus

Au niveau du noyau, la gestion des processus est assurée à l'aide notamment d'une table appelée table des processus. Cette table est résidente en mémoire et de taille fixe, mais configurable au boot. Chaque entrée de la table contient les informations principales suivantes :

- les identificateurs du processus (PID, parent PID, group PID, UID)
- les informations de scheduling (priorité, usage CPU, etc.)
- un pointeur sur la description du fichier exécutable
- un pointeur sur la table des textes utilisée pour adresser le code.
- un pointeur sur la table des régions du processus.
- la taille du code résidant en mémoire.
- l'adresse de la u.area.

Dans le système Linux la mémoire est paginée, avec allocation possible à la page dynamiquement. Le noyau est résident en permanence en mémoire. A un instant donné, plusieurs processus peuvent partager la mémoire. Lorsque le nombre de processus devient trop grand pour la mémoire disponible, le système transfert les processus en excès sur une mémoire

secondaire (swapping). La gestion des transferts du disque vers la mémoire est fonction notamment de la durée de résidence disque et de la taille du processus. La gestion des transferts de la mémoire vers le disque prend en compte :

- les processus en attente d'événements lents (E/S)
- la durée de résidence en mémoire
- la taille mémoire du processus

Il existe une durée minimale de présence en mémoire pour éviter au système de multiplier les transferts disque-mémoire et vice versa lorsque la charge augmente (thrashing).

1.3.1 Duplication de processus

Comme nous l'avons vu précédemment, il n'y a qu'une seule méthode pour créer un nouveau processus sous Linux qui consiste à dupliquer un processus existant. Cette opération est réalisée par l'appel système `fork()`, de syntaxe :

```
int fork(void);
```

Description : Le processus créé, appelé fils, est la copie du processus original qui devient alors processus père. Les deux processus reçoivent le retour de l'appel `fork()`, mais la valeur retournée est évidemment différente afin de pouvoir les distinguer au niveau des actions à exécuter :

- Le processus fils reçoit un retour égal à 0,
- tandis que le processus père reçoit un retour de fork() égal au PID du fils créé.

Comme souvent pour les appels système un retour égal à -1 indique une erreur qui ne peut provenir que du système puisque l'utilisateur ne passe aucun paramètre. L'unique motif d'un l'échec de `fork()` est une insuffisance de ressource, soit de la mémoire de swap, soit de la taille de la table des processus, soit du nombre de processus. Les caractéristiques du processus fils sont identiques pour la plupart à celles du processus père. En particulier, les deux processus partagent le même code d'instructions, les mêmes fichiers ouverts, de sorte que si le fils exécute un `lseek()` sur l'un des fichiers par exemple, le prochain `read()` ou `write()` du père sur ce fichier en sera affecté. Cependant, le processus fils se distingue sur les points suivants :

- Son identification par le système est différente : il a un nouveau PID et il a comme parent-PID, le PID du père.
- Le processus fils a sa propre copie des descripteurs de fichiers ouverts par le père. Ces descripteurs pointent les mêmes fichiers que ceux du père, mais ils sont différents. Ainsi, si le fils fait un appel à `close()` sur l'un des fichiers, le processus père n'en sera pas affecté.
- Le processus fils démarre son exécution autonome juste après l'instruction `fork()`. Par suite, le temps d'exécution cumulée du processus est ramenée à 0.

Exemple de code V.1

```
 main(void)
{
  switch(fork()) {
    case -1:
        printf("Impossible de créér un nouveau processus \n");
        exit(1);
        break;
    case 0:
        printf("Je suis le fils\n");
        exit(0);
        break;
    default:
        printf("Je suis le père \n");
  }
}
```

Remarques V.1. (*i*) Le fils et le père continuent de partager les mêmes fichiers. Il s'ensuit que s'ils doivent s'exécuter simultanément, un protocole doit fixer l'utilisation simultanée des fichiers partagés. Ainsi quand un processus est lancé en tâche de fond (background), shell utilise `fork()` pour créer le nouveau processus qui exécute la commande, mais redirige l'entrée standard de ce processus sur `/dev/null` pour pouvoir conserver l'usage normal du terminal.

(*ii*) Certains effets inattendus liés à l'opération de duplication peuvent se produire. Par exemple, lorsque qu'une sortie est effectuée juste avant un `fork()`, l'écriture est faite normalement dans un tampon. Or ce tampon est dupliqué par le `fork()` entraînant ainsi une double écriture de ce que restait dans le tampon avant le `fork()`. Il est donc prudent de toujours vider les tampons (par `fsync()` par exemple) avant la création d'un nouveau processus.

1.4 Arrêt d'un processus

Un processus sous Linux s'achève en exécutant l'appel système `exit()`, de syntaxe :

```
void exit(int status);
```

Description : C'est le seul appel système qui n'a jamais de retour au processus appelant. Le processus qui exécute `exit()` entre dans l'état zombie qui se traduit par l'abandon de toutes les ressources, mais le noyau maintient l'entrée correspondante dans la table des processus. Le processus reste dans l'état zombie jusqu'à ce que le processus père exécute un `wait()`. Il disparaît alors. Il n'y a pas d'état zombie si le processus père intercepte le signal `SIGCHLD`.

L'appel `exit()` transmet au processus père l'octet de poids faible de status. La valeur transmise est récupérée par le processus père grâce à l'appel système `wait()`. Les 256

valeurs que peut transmettre un processus fils à son père sont comprises entre 0 et 255. Par convention, la valeur 0 indique une fin normale et une valeur non nulle (1 en général) renseigne le processus parent sur la nature de l'erreur survenue. Les 256 valeurs possibles sont utilisables. Si `exit()` est invoqué sans argument, la valeur transmise au processus père est aléatoire.

L'appel à `exit()` ferme tous les fichiers ouverts par le processus. D'autre part, si les fils du processus exécutant un `exit()` sont encore vivants, leur exécution n'est pas affectée par la disparition du processus père, mais leur numéro de parent PID est changé en 1. Dans ces conditions les processus fils peuvent à la fin de leur exécution rester un temps variable dans l'état zombie jusqu'à ce que le processus init exécute un `wait()` (ce qui est fait périodiquement).

Exemple de code V.2

```
 main(void)
{
    int fd1,fd2;
    if ((fd1 = open("entree", O_RDONLY)) < 0)
        exit(1);
    if ((fd2 = open("sortie", O_WRONLY)) < 0)
        exit(2);
    exit(0);
}
```

A la fin de cette séquence, la valeur retournée par `exit()` sera :

0 fin normale

1 impossible d'ouvrir le fichier d'entrée

2 impossible d'ouvrir le fichier de sortie

1.5 Attente d'un processus

Le processus père peut synchroniser son exécution avec la terminaison d'un processus fils grâce à l'appel système `wait()`, de syntaxe :

```
int wait(int *status);
```

L'appel `wait()` suspend l'exécution du processus père. La fin d'un processus fils provoque le réveil du processus père. `wait()` retourne le PID du fils ou -1 en cas d'erreur. Trois causes peuvent entraîner l'arrêt d'un processus. Le processus peut faire un appel à `exit()`. Il peut également recevoir un signal fatal. Enfin il peut être arrêté par une panne matérielle ou logicielle de l'ordinateur. `wait()` indique lequel des 2 premiers cas est survenu.

En fait, `status` est une valeur combinée de deux octets en un entier de 16 bits. Les 8 bits de poids faible sont fournis par le système et contiennent 0 si l'exécution s'est bien terminée. Si le noyau a stoppé le processus fils par suite d'une erreur, les 7 bits de poids faible contiennent le numéro du signal d'arrêt et le 8ème bit à 1 indique dans ce cas qu'un

fichier core a été généré. Les 8 bits de poids forts correspondent à l'octet fourni par l'appel système `exit()` du processus fils (cas de sortie normale) ou 0 en cas d'interruption. Il n'y a aucun moyen de spécifier lequel des fils est attendu. En fait `wait()` attend tous les processus fils et retourne au premier des fils qui se termine. Un processus fils qui se termine reste à l'état zombie tant que le processus père n'a pas exécuté un `wait()`. Il faut donc en principe autant de `wait()` que d'appels à `fork()`. Seul le père peut attendre la fin d'exécution de ses fils et il faut bien se rappeler que status est un pointeur sur un entier.

Exemple de code V.3

```
 main(void)
{
    int stat,pid;

    if(fork()==0) {
        printf("je suis le fils\n");
        exit(0);
    }
    else {
        pid=wait(&stat);
        printf("je suis le père\n");
        printf("Le PID du fils terminé est %d\n",pid);
        if (stat & 0377)    /* teste le code système */
            printf("Fin anormale par le signal %d\n",stat & 0177);
        else
            printf("Fin normale, code de retour : %d\n",stat >> 8);
    }
}
```

Dans cette séquence, le processus père teste si le fils s'est terminé normalement et affiche dans ce cas le code de retour (0). Dans le cas contraire, il affiche le numéro de signal que le noyau a envoyé au fils pour l'arrêter.

1.6 Recouvrement d'un processus

1.6.1 Appel système `execl()`

Pour comprendre le fonctionnement des appels systèmes `fork()` déjà vu, et `exec()` que nous allons étudier maintenant, il est essentiel de bien différencier la notion de processus de celle de programme. Un processus est l'exécution d'un ensemble fait d'une zone d'instructions, d'une zone de données utilisateur, et d'une zone de données système. Un programme est un fichier contenant les instructions et les données utilisées pour initialiser la zone d'instructions et la zone de données utilisateur d'un processus.

L'appel système `exec()` a pour objet de ré-initialiser un processus à partir d'un programme désigné. Il ne crée pas de nouveau processus. Au contraire, l'appel `fork()` créé un nouveau processus en dupliquant les zones instructions, données utilisateur et données système d'un processus déjà existant, mais le nouveau processus n'est pas initialisé à partir d'un programme. Les deux appels systèmes sont donc complémentaires, et par suite, l'utilisation de l'un sans l'autre n'a que des applications assez limitées.

Il y a 6 appels systèmes `exec()` qui fonctionnent de manière similaire à quelques nuances près. Nous présentons ici seulement `execl()` en détail et nous signalerons dans le paragraphe suivant les variantes des autres appels. Syntaxe de l'appel :

```
int execl(char *path, char *arg0,..., char *argn, NULL);
          /* Recouvrement de processus                */
          /* path = chemin d'accès au fichier         */
          /* arg0 = premier argument (nom du fichier) */
          /* argn = dernier argument                  */
```

L'appel système `execl()` recouvre la zone d'instructions et la zone des données utilisateur par celles du programme contenu dans le fichier spécifié. Alors, le processus exécute le nouveau programme depuis le début (c'est à dire que sa fonction `main()` est appelée). En cas de succès, il n'y a pas de retour au programme appelant car le programme appelé est chargé en lieu et place du programme appelant.

Le chemin d'accès désigné par path doit faire référence à un fichier exécutable. D'autre part, les autres arguments de `execl()` sont rassemblés dans un tableau de pointeurs sur caractères, le dernier argument devant être obligatoirement le pointeur `NULL`. Ces arguments sont accessibles par la récupération traditionnelle des arguments de passage `main(argc, argv)`. C'est pourquoi, pour rester cohérent avec cette méthodologie, le paramètre `arg0` doit être par convention, l'adresse de la chaîne contenant le nom du fichier (et non le chemin d'accès complet).

Le programme appelé reprend le contexte d'exécution du processus qui a subi le recouvrement. Cela se traduit notamment par le fait que presque tous les attributs du processus demeurent inchangés, en particulier le PID, le parent PID, le process group ID, l'UID et le GID réels (identifications d'utilisateur et de groupe réelles), ou encore la priorité et le temps d'exécution cumulée restent identiques.

Les fichiers ouverts avant `execl()` restent accessibles après sous réserve de transmettre les descripteurs en arguments. Ces descripteurs peuvent être, par exemple, préalablement à l'appel à `execl()`, transformés en chaînes de caractères, puis retransformés en nombres entiers après. Ces opérations se font aisément avec les routines `sprintf()` et `atoi()`.

Exemple de code V.4

```
 main(void)
{
    int fd1, fd2, pid, status;
    char sdf2[10], *path = "/home/stage1/b.out";
```

```
    if ((fd1 = open("toto",O_RDONLY)) < 0)
            exit(1);
    if ((fd2 = open("tata",O_WRONLY)) < 0)
            exit(2);
    switch(fork()) {
    case -1:
        printf("Impossible de créer un nouveau processus\n");
        exit(1);
    case 0:    /* processus fils */
        fcntl(fd1,F_GETFD,0);    /* fichier fd1 fermé à l'execl() */
        sprintf(sfd2,"%d",fd2);  /* conversion de fd2 en chaine */
        execl(path,"b.out",sfd2,NULL);
        printf("Echec execl()\n");
        exit(2);
    default:    /* processus père */
        pid = wait(&status);
    }
}
```

Dans cet exemple, les fichiers `"toto"` et `"tata"` sont ouverts. Puis le processus fils qui est créé, demande au noyau par un appel à `fcntl()` de fermer le fichier dont le descripteur est `fd1` lors du prochain `execl()`. Il faut noter que si l'appel à `execl()` échoue, le fichier `fd1` restera ouvert et disponible pour le processus fils. Le fichier dont le descripteur est `fd2` restera normalement ouvert et disponible dans le programme `b.out`.

Remarques V.2. (*i*) `path` doit être l'adresse du nom complet du fichier correspondant au programme à exécuter. Si on se trouve dans le répertoire `/home/stage1` et que l'on désire exécuter le programme `a.out`, il faut spécifier `/home/stage1/a.out` et non simplement `a.out`.

(*ii*) Il faut se rappeler que le premier des arguments passés à un programme est le nom de la commande elle-même. Donc `arg0` doit exister dans tous les cas.

Par exemple, pour provoquer le recouvrement du processus appelant par le programme contenu dans `/bin/date` il faut passer date en premier argument de `execl()` :

```
execl("/bin/date","date",0);
```

1.6.2 Autres formes d'`exec()`

Les cinq autres appels système `exec()` fournissent selon les cas, les trois possibilités suivantes :

- Passer les arguments dans un tableau au lieu de les lister. Cette propriété est utile, lorsque le nombre d'argument est inconnu au moment de la compilation.
- Utiliser la variable d'environnement `PATH` pour chercher le chemin d'accès du fichier exécutable spécifié.

– Passer un pointeur explicite d'environnement, au lieu d'utiliser le pointeur standard environ.

Les diverses variantes d'`exec()` sont présentées ci-dessous :

Appel système	Passage d'arguments	Passage d'environnement	Utilisation de `PATH`
`execl()`	liste	automatique	non
`execv()`	tableau	automatique	non
`execle()`	liste	manuel	non
`execve()`	tableau	manuel	non
`execlp()`	liste	automatique	oui
`execvp()`	tableau	automatique	oui

Syntaxe des appels :

```
int execv(char *path,char *argv[]);
        /* path = chemin d'accès au fichier                */
        /* argv = tableau de pointeurs sur les arguments   */
int execle(char *path,char *arg0,..,char *argn,NULL,char *envp[]);
        /* path = chemin d'accès au fichier                */
        /* arg0 = premier argument (nom du fichier)        */
        /* argn = dernier argument                         */
        /* envp = tableau de pointeurs sur l'environnement */
int execve(char *path,char *argv[],char *envp[]);
        /* path = chemin d'accès au fichier                */
        /* argv = tableau de pointeurs sur les arguments   */
        /* envp = tableau de pointeurs sur l'environnement */
int execlp(char *path,char *arg0,..,char *argn,NULL);
        /* path = nom du fichier                           */
        /* arg0 = premier argument (nom du fichier)        */
        /* argn = dernier argument                         */
int execvp(char *path,char *argv[]);
        /* path = chemin d'accès au fichier                */
        /* argv = tableau de pointeurs sur les arguments   */
int execvpe(char *path,char *argv[],char *envp[]);
        /* path = chemin d'accès au fichier                */
        /* argv = tableau de pointeurs sur les arguments   */
        /* envp = tableau de pointeurs sur l'environnement */
```

Les appels `execv()` et `execve()` sont spécialement recommandés quand on ne connait pas a priori le nombre d'arguments. Le nombre d'arguments est déterminé par la position du pointeur `NULL`. Si on suppose qu'un programme récupère un certain nombre de paramètres contenus dans `argv`, `execv()` peut être utilisé sous la forme suivante :

Exemple de code V.5

```
main(int argc, char *argv[])
{
    char *cmd[];
    int status, pid;

    if ((pid = fork()) == 0) {
        cmd = (char *[])calloc(argc, sizeof(char *));
        cmd[0] = calloc(strlen("cat")+1, sizeof(char));
        strcpy(cmd[0], "cat");
        for (i = 1; i < argc; i++)
            cmd[i] = argv[i];
        cmd[argc] = (char *)NULL;
        execv("/bin/cat",cmd);
    }
    else
        pid = wait(&status);
        exit(0);
}
```

L'appel suivant :

```
execl("ls", "ls", NULL);
```

échouera car le premier argument de `execl()` doit être le chemin complet de la commande ; il faudra taper

```
execl("/bin/ls", "ls", NULL);
```

Si l'on veut éviter d'entrer le chemin complet de la commande, on peut utiliser une forme dite avec utilisation de `PATH` dans le tableau ci-dessus. Rappelons que la variable d'environnement `PATH` contient une liste de répertoires séparés par des `:`. Cette liste est utilisée par l'interpréteur de commandes lorsque l'utilisateur entre une commande (par exemple `ls`) : si ce dernier trouve la commande dans l'un de ces répertoires, il l'exécute ; dans le cas contraire, il renvoie un message d'erreur. Demanière similaire, `execlp()` utilise la variable `PATH` pour inspecter les divers répertoires qu'elle référence afin de voir s'il trouve la commande donnée en premier argument ; s'il la trouve, il l'exécute ; dans le cas contraire, il échoue en renvoie un message d'erreur. Ainsi l'appel

```
execlp("ls", "ls", NULL);
```

est il correct. Dans les appels `execlp()` et `execvp()`, si la recherche du fichier à l'aide de la variable `PATH` n'aboutit à rien, l'`exec()` échoue. Si le nom du fichier fourni contient le caractère `'/'`, alors le nom du fichier est supposé avoir été donné sous sa forme complète et `PATH` n'est pas utilisée. Il faut noter enfin qu'aucune des formes de l'`exec()` n'autorise l'utilisation des symboles spéciaux shell tels que `>`, `<`, `*`, ou `?`. Par exemple, l'appel suivant ne provoquera pas l'effet escompté :

```
execl("/bin/cat","toto",">fich",0);
```

Cet inconvénient peut être contourné en invoquant l'exécution d'un shell.

1.6.3 Exécution d'un programme via le shell

Les différents appels système à `exec()` peuvent initialiser l'exécution d'un shell auquel une commande à exécuter est transmise. De cette façon, il est possible de récupérer toutes les fonctions propres au shell. La forme la plus adaptée à cet effet est l'appel `execl()` ou l'appel `execlp()` :

```
execl("/bin/sh","sh","-c",commande,NULL);
```

Dans cette syntaxe, commande est un pointeur sur une chaine de caractères exactement semblable à celle que l'on aurait directement tapée au terminal sous le shell. Le shell doit être invoqué avec l'option `-c`, sinon la commande serait interprétée comme le nom d'un fichier exécutable.

Exemple de code V.6

```
main(int argc, char *argv[])
{
    int status, pid;
    if ((fork() == 0) {
        execl("/bin/sh","sh","-c","cat *.c >fich",NULL);
    }
    else
        pid = wait(&status);
    exit(0);
}
```

L'exécution de cette séquence se traduira par la copie dans le fichier `"fich"` du contenu de tous les fichiers dont le nom se termine par `".c"`.

Notons au passage, qu'il existe sous Linux la fonction `system()` (qui n'est pas un appel système en dépit de son nom) permettant de générer un nouveau processus et de lancer l'exécution d'un programme donné à l'aide d'un shell. Cette fonction enchaîne un appel système `fork()` suivi d'un `execl()` semblables à ceux utilisés dans l'exemple précédent. Syntaxe :

```
#include <stdio.h>
int system(char *cmde);
```

La fonction `system()` lance un shell en lui fournissant le paramètre `cmde`. Le processus appelant est suspendu durant l'exécution de l'instruction `system()`. Il reprend après la fin d'exécution de la commande `cmde`. La fonction `system()` retourne le code de retour de la commande exécutée. Exemple :

```
system("cat *.c >fich");
```

Cette instruction provoquera le même résultat que celui obtenu dans le précédent exemple.

1.7 Gestion de l'environnement

1.7.1 lecture de l'environnement

Plusieurs appels système permettent à un processus d'obtenir des informations sur son environnement. Ils sont simples et d'une utilisation généralement évidente. Les plus utiles sont les suivants :

```
int getpid(void);
  /* Retourne le numéro de PID du processus appelant              */
int getppid(void);
  /* Retourne le numéro de processus du père.(PPID)               */
int getuid(void);
  /* Retourne l'identification du propriétaire du processus (UID) */
  /* L'UID est associé au nom de login dans le fich /etc/passwd   */
int geteuid(void);
  /* Retourne l'UID effectif                                      */
int getgid(void);
  /* Retourne l'identification du groupe du propriétaire (GID)    */
  /* Le GID est associé au nom de login dans le fich /etc/group   */
int getegid(void);
  /* Retourne le GID effectif                                     */
```

Tout processus peut accéder directement à son environnement grâce à la variable standard `environ`. Par convention, l'environnement est formé de couples variable=valeur terminés par le caractère `'\0'` (après la valeur). Les variables d'environnement sont normalement mises en place par le shell. L'accès à cet environnement se fait en utilisant le pointeur environ.

Exemple de code V.7

```
main(int argc,char *argv[])
{
    extern char **environ;
     /* environ est un pointeur sur tableau de caractères */
    int i=0;
    while (environ[i]) {
        printf("%s\n",environ[i]);
        i++;
    }
    exit(0);
}
```

Cet exemple affiche à l'écran l'ensemble des variables d'environnement. Il existe une fonction standard (ce n'est pas un appel système) permettant d'accéder à la valeur d'une variable donnée de l'environnement. Cette fonction est définite comme suit :

```
char *getenv(char *nom);
  /* Retourne l'adresse de la valeur associée à nom */
```

`getenv()` cherche dans la liste des variables d'environnement associées au processus, la chaine de caractères passée en argument. `getenv()` renvoie l'adresse de la valeur de la variable si nom appartient à l'environnement. Dans le cas contraire la fonction renvoie `NULL`.

1.7.2 Modification de l'environnement

Il est possible d'apporter certaines modifications à l'environnement d'un processus. Parmi les appels système disponibles à cet effet, on trouve notamment :

```
int setuid(int uid);
  /* Retourne 0 ou -1 en cas d'erreur */
int setgid(int gid);
  /* Retourne 0 ou -1 en cas d'erreur */
```

Si le super-user est propriétaire du processus appelant, `setuid()` fixe l'UID effectif et réél du processus à la valeur `uid`, et `setgid()` fixe le GID effectif et réél du processus à la valeur `gid`. Le super-user a donc la possibilité de devenir n'importe quel utilisateur.

Lorsque ces appels système sont invoqués par des utilisateurs ordinaires, ils ne peuvent être utilisés que pour ramener respectivement l'UID effectif à l'UID réél de l'utilisateur, et le GID effectif au GID réél. Dans ce cas, les arguments `uid` et `gid` doivent être évidemment adéquats.

```
int seteuid(int uid);
  /* Retourne 0 ou -1 en cas d'erreur */
int setegid(int gid);
  /* Retourne 0 ou -1 en cas d'erreur */
```

L'appel `seteuid()` fixe l'UID effectif à la valeur du paramètre `uid` si celle-ci est égale à l'UID réel du processus, ou si l'UID effectif actuel est celui de `root`. `setegid()` fixe le GID effectif du processus à `gid` si l'une des conditions suivantes est remplie :

- le GID effectif demandé est égal à la valeur courante du GID réel.
- l'UID effectif du processus est celui de root.

2 Threads Posix

2.1 Notion de thread

Les threads ("fils d'éxecution") ne retiennent que la notion d'unité d'exécution par rapport aux processus. En particulier, ils partagent le même espace d'adressage (même segment de données utilisateur). L'aspect unité d'exécution comporte en particulier la pile, les registres du processeur et un compteur d'instruction. La gestion et la commutation de contexte sont plus légères et plus aisées que pour un processus, par contre l'accès aux données est concurrentiel.

L'utilisation des threads posix, également dénomées pthreads se fait usuellement au moyen de la bibliothèque LinuxThreads. Pour utiliser cette dernière, il faut inclure l'entête `<pthread.h>` dans les fichiers source, ajouter la définition de constante `-D_REENTRANT`

sur la ligne de commande du compilateur, et ajouter l'option `-lpthread` sur la ligne de commande de l'éditeur de liens.

Les threads de la bibliothèque LinuxThreads sont dites implantées dans l'espace noyau, ce qui signifie que chaque thread est représentée par un processus indépendant, partageant son espace d'adressage avec les autres threads de la même application. En implémentation dite utilisateur, l'application n'est constituée que d'un seul processus, et la répartition en différentes threads est assurée par une bibliothèque indépendante du noyau. L'implantation dans l'espace noyau possède les caractéristiques suivantes :

- La création d'un thread nécessite un appel-système.
- La commutation entre deux threads est opérée par le noyau avec changement de contexte.
- Du point de vue de l'ordonnancement, chaque thread dispose des mêmes ressources CPU que les autres processus du système.
- Chaque thread peut s'exécuter avec une priorité indépendante des autres, éventuellement en ordonnancement dit temps-réel.
- Le noyau peut accessoirement répartir les threads sur différents processeurs pour profiter du parallélisme d'une machine SMP.

2.2 Création, destruction, attente

Création

Pour créer une thread on utilisera l'appel `pthread_create()` Le prototype de la fonction `pthread_create()` est le suivant :

```
int pthread_create(pthread_t *thread, pthread_attr_t *attributs,
                   void * (*fonction) (void *argument), void *argument);
```

Et les arguments ont la signification suivante :

- Le 1^{ier} argument est l'identifiant de la nouvelle thread.
- Le 2^{e} argument est un pointeur sur une structure d'attributs (voir sous section suivante) ; si cet argument est mis à la valeur `NULL`, les attributs par défaut sont adoptés.
- Le 3^{e} argument est un pointeur sur la fonction (routine) exécutée lors du lancement de la thread, dite fonction principale de la thread.
- Le 4^{e} argument est un pointeur sur l'argument transmis à la fonction (l'argument peut être un type numérique ou une structure).

Cet appel renvoit zéro si elle réussit et une valeur non nulle sinon, correspondant à l'erreur survenue.

Destruction – Recupération de code de retour

Lorsque la fonction principale d'une thread se termine, la thread correspondante est éliminée. Il est également possible de terminer une thread par l'appel `pthread_exit()`, de syntaxe :

```
void pthread_exit(void *retour);
```

`pthread_exit()` met fin à la thread appelante et renvoie le pointeur `retour` de type `void *` passé en argument. Ce code de retour peut être récupéré par la fonction `pthread_join()`, de syntaxe :

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

qui suspend l'exécution de la thread appelante jusqu'à la terminaison de la thread indiquée en argument. Elle remplit alors le pointeur passé en second argument avec la valeur de retour de la thread terminée.

Dans certaines conditions, une thread peut être tuée par une autre thread, un peu à la manière d'un processus qui peut être tué par un signal. Dans ce cas, la thread terminée n'a pu renvoyer de valeur, aussi la variable `* retour` prend-elle une valeur particulière : `PTHREAD_CANCEL`.

Voici un exemple simple de création de threads :

Exemple de code V.8

```
#include <pthread.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
#include <unistd.h>
#define NB THREADS 5
void *fn_thread (void *numero);
static int compteur = 0;

int main(void)
{
  pthread_t thread[NB_THREADS];
  int i;
  int ret;

  for (i = 0; i < NB THREADS; i ++)
    if ((ret = pthread_create(&thread [i], NULL, fn_thread,
                                                (void *)&i)) != 0) {
          fprintf(stderr, "%s", strerror (ret));
          exit (1);
    }
  while (compteur < 40) {
      fprintf(stdout, "main : compteur = %d \n", compteur);
```

```
        sleep (1);
    }
    for (i = 0; i < NB THREADS; i++)
        pthread_join(thread[i], NULL);
        return (0);
    }

void *fn_thread(void * num)
{
    int numero = *((int *)num);
    while (compteur < 40) {
        usleep (numero * 100000);
        compteur ++;
        fprintf(stdout, "Thread %d : cptr = %d \n", numero, compteur);
    }
    pthread_exit(NULL);
}
```

Le partage d'une variable globale entre les différentes threads n'est pas une bonne chose en général. Il n'est utilisé ici qu'à des fins de simplicité de l'exemple. Notez la manière dont est passé l'argument de type `int` ; ceci est typique d'un passage d'argument à une fonction qui attend un type générique `void *`.

Voici un autre exemple d'utilisation de `pthread_exit()` et `pthread_join()` :

Exemple de code V.9

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

void *fn_thread(void *inutile)
{
    char chaine [128];
    int i = 0;

    fprintf(stdout, "Thread : entrez un nombre :");
    while (fgets(chaine, 128, stdin) != NULL)
        if (sscanf (chaine, "%d", & i) != 1)
            fprintf(stdout, "un nombre SVP :");
        else
            break;
    pthread_exit ((void *)i);
```

```
}

int main (void)
{
  int i;
  int ret;
  void *retour;
  pthread_t thread;

  if ((ret = pthread_create(&thread, NULL, fn_thread, NULL)) != 0) {
      fprintf(stderr, "%s\n", strerror (ret));
      exit(1);
  }
  pthread_join(thread , &retour);
  if (retour != PTHREAD_CANCELED) {
      i = (int)retour;
      fprintf(stdout, "main : valeur lue = %d\n", i);
  }
  return(0);
}
```

L'exécution donne l'affichage suivant :

```
# ./exemple_join
Thread : entrez un nombre :6569
main : valeur lue = 6569
```

La fonction `pthread_detach()` permet de faire disparaître complètement une thread lors de sa terminaison :

```
int pthread_detach(pthread_t thread_id);
```

où `thread_id` est l'identification de la thread à arrêter.

La fonction `pthread_self()` renvoie l'identificateur de la thread appelante :

```
pthread_t pthread_self(void);
```

Pour comparer l'identificateur de deux threads, on doit utiliser la fonction `pthread_equal()`

```
int pthread_equal(pthread_t thread_1, pthread_t thread_2);
```

Cette fonction renvoie une valeur non nulle s'ils sont égaux.

2.3 Gestion des attributs

Avant de pouvoir gérer les attributs d'une thread, il faut appeler la fonction `int pthread_attr_init()` :

```
int pthread_attr_init (pthread_attr_t * attributs);
```

Une fois initialisés, les fonctions `pthread_attr_getXXX()` et `pthread_attr_setXXX()` permettent de consulter et de modifier les attributs via un objet de type `pthread_attr_t`.

VI – Signaux et Timers Posix

1 Généralités sur les signaux

1.1 Introduction

Le mécanisme des signaux est un mécanisme d'interruption logicielle. Il est souvent utilisé comme réponse du système à un événement jugé anormal qui survient pendant l'exécution d'un programme. La frappe de la touche d'interruption (`Ctrl-c`) ou bien la détection par le système d'une action illicite entraine la génération d'un signal. Un signal interrompt l'exécution normale d'un processus et aboutit généralement à l'arrêt du processus. Le système Unix met à la disposition de l'utilisateur un certain nombre d'appels

système permettant de contrôler l'action des signaux envoyés à un processus. L'appel système `signal()` permet de définir l'action du noyau sur le processus lors de la survenue d'un signal. Il est utilisé souvent en conjonction avec un autre type d'appels système permettant la sauvegarde et la restauration du contexte d'exécution d'un processus ; on utilise la paire `setjmp()`, `longjmp()`, ou bien l'ensemble `getcontext()`, `makecontext()`, `setcontext()` et `swapcontext()`.

Il existe un nombre déterminé de signaux (32 sous Linux 2.0, 64 depuis Linux 2.2). Chaque signal dispose d'un nom défini sous forme de constante symbolique commençant par `SIG` et d'un numéro associé. Il n'y a pas de nom pour le signal numéro 0, car cette valeur a un rôle particulier que nous verrons plus tard. Toutes les définitions concernant les signaux se trouent dans le fichier d'en-tête `<signal.h>` (ou dans d'autres fichiers qu'il inclut lui-même). La constante symbolique `NSIG` (ou `_NSIG` sur certaines distributions) définie par la bibliothèque C correspond au nombre de signaux, y compris le signal 0.

Les signaux dits classiques vont de 0 à 31 et les signaux dits temps réel vont de 32 (valeur généralement prise par `SIGRTMIN`) à 63 (valeur généralement prise par `SIGRTMAX`).

1.2 Types de signaux

Posix précise que le comportement d'un programme qui ignore les signaux d'erreur du type SIGFPE, SIGILL, SIGSEGV, SIGBUS est indéfini.

Signaux provoqués par une exception matérielle	
`SIGFPE`	*Floating point exception* : envoyé lors d'un problème avec une opération en virgule flottante mais il est aussi utilisé pour l'erreur de division entière par zéro. L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core. Ce fichier est alors utilisé par l'outil de déboguage `gdb`.
`SIGILL`	*Illegal instruction* : envoyé au processus qui tente d'exécuter une instruction illégale. Ceci ne doit jamais se produire dans un programme normal mais il se peut que le fichier exécutable soit corrompu (erreur lors du chargement en mémoire). L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGSEV`	*Segmentation violation* : envoyé lors d'un adressage mémoire invalide (emploi d'un pointeur mal initialisé). L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGBUS`	*Bus error* : envoyé lors d'une erreur d'alignement des adresses sur le bus. L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGTRAP`	*Trace trap* : envoyé après chaque instruction, il est utilisé par les programmes de mise au point (debug). L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
Signaux de terminaison ou d'interruption de processus	

`SIGHUP`	*Hangup* : envoyé à tous les processus attachés à un terminal lorsque celui-ci est déconnecté du système. Il est courant d'envoyer le signal à certains processus démons afin de leur demander de se réinitialiser. L'action par défaut est la mise à mort du processus.
`SIGINT`	*Interrupt* : le plus souvent déclenché par `[CTRL-C]`. L'action par défaut est la mise à mort du processus.
`SIGQUIT`	*Quit* : similaire à `SIGINT` mais pour `[CTRL] [\]`. L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGIOT`	*I/O trap instruction* : émis en cas de problème hardware (I/O). L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGABRT`	*Abort* : Arrêt immédiat du processus (erreur matérielle). L'action par défaut est la mise à mort du processus en créant un fichier d'image mémoire core.
`SIGKILL`	*Kill* : utilisé pour arrêter l'exécution d'un processus. L'action par défaut est non modifiable.
`SIGTERM`	*Software termination* : C'est le signal qui par défaut est envoyé par la commande `kill`. L'action par défaut est la mise à mort du processus.
	Signaux utilisateurs
`SIGUSR1`	*User defined signal 1* : Définie par le programmeur. L'action par défaut est la mise à mort du processus.
`SIGUSR2`	*User defined signal 2* : idem que `SIGUSR1`.
	Signal généré pour un tube
`SIGPIPE`	Signal d'erreur d'écriture dans un tube sans processus lecteur. L'action par défaut est la mise à mort du processus.
	Signaux liés au contrôle d'activité
`SIGCLD`	Signal envoyé au processus parent lorsque les processus fils terminent. Aucune action par défaut.
`SIGCONT`	L'action par défaut est de reprendre l'exécution du processus s'il est stoppé. Aucune action si le processus n'est pas stoppé.
`SIGSTOP`	Suspension du processus. L'action par défaut est la suspension (non modifiable).
`SIGTSTP`	Emission à partir d'un terminal du signal de suspension `[CTRL-Z]`. Ce signal a le même effet que `SIGSTOP` mais celui-ci peut être capturé ou ignoré. L'action par défaut est la suspension.
`SIGTTIN`	Est émis par le terminal en direction d'un processus en arrière-plan qui essaie de lire sur le terminal. L'action par défaut est la suspension.
`SIGTTOU`	Est émis par le terminal en direction d'un processus en arrière-plan qui essaie d'écrire sur le terminal. L'action par défaut est la suspension.
	Signal lié à la gestion des alarmes

`SIGALRM`	*Alarm clock* : généré lors de la mise en route du système d'alarme par l'appel système `alarm()`.

2 Signaux Linux

2.1 Déclaration de comportement : `signal()`

Le but de l'appel système `signal()` est d'indiquer au noyau l'action à effectuer lors de l'arrivée d'un signal. Par défaut, le noyau exécute une action standard consistant à arrêter et à supprimer le processus qui reçoit le signal, sauf pour quelques signaux (`SIGCHLD`, `SIGPWR`, `SIGCONT`) pour lesquels, l'action standard est d'ignorer le signal.
Lorsque la macro `_GNU_SOURCE` est définie

```
typedef void (*sighandler_t)(int);
```

le type `sighandler_t` (non standard) correspond à un pointeur sur fonction ne renvoyant rien (`void`) et admettant un seul argument de type `int`.

La syntaxe de l'appel `signal()` est alors :

```
#include <signal.h>
sighandler_t signal(          Redéfinit l'action du signal nosig
    int nosig,                Numéro du signal
    sighandler_t handler)     pointeur sur la fonction définissant l'action
```

Description : `nosig` est un entier ou une constante mnémonique qui définit le signal dont on veut changer le comportement (cf. sous section 1.2, p. 140) et `handler` est un pointeur sur une fonction définissant la nouvelle action ou une constante se référant à une action prédéterminée. `signo` ne peut être ni `SIGKILL`, ni `SIGSTOP`. `handler` peut être assigné à trois valeurs, `SIG_DFL` (action par défaut), `SIG_IGN` (ignorer le signal) ou l'adresse d'une fonction définie par l'utilisateur et qui représente l'action à effectuer en cas de réception du signal.

L'appel système `signal()` retourne un pointeur sur une fonction ne retournant rien (`void`). Ce pointeur pointe sur l'action précédant la réception du signal et peut ainsi être utilisé dans des actions ultérieures pour restaurer l'action du signal. En cas d'erreur, l'appel système retourne `-1`.

Les actions définies pour un processus père sont héritées par les processus fils lors d'une création de nouveau processus par `fork()`. De plus, les actions `SIG_DFL` et `SIG_IGN` sont préservées lors du recouvrement d'un processus par un `exec()`. En revanche, les actions définies par l'utilisateur sont ramenées à `SIG_DFL` lors d'un `exec()` puisque les fonctions d'interception sont effacées.

2.2 Interception d'un signal

L'utilisateur à la possibilité de définir l'action à effectuer en cas de réception d'un signal donné. Le deuxième paramètre de `signal()` est l'adresse de la fonction stipulant cette

action. Le numéro du signal reçu peut éventuellement être passé comme unique argument de cette fonction. Il est impossible de redéfinir l'action des signaux `SIGILL`, `SIGTRAP` et `SIGPWR`. Le mode d'action est le suivant :

- Le processus qui reçoit le signal est interrompu.
- Avant d'exécuter la fonction spécifiant l'action à exécuter, le système rétablit l'action par défaut. Cela signifie que si le même signal est reçu une nouvelle fois, il sera traité de façon standard.
- Lorsqu'un signal est intercepté durant un appel système, en particulier `read()`, `write()`, `open()`, `wait()`, l'appel considéré est interrompu et renvoie un code de retour égal à `-1` au processus appelant indiquant ainsi une erreur. Dans ce cas, la variable globale d'erreur `errno` est mis à la valeur `EINTR`.
- Après exécution de la fonction, le processus reprend à l'endroit où il a été interrompu, sauf bien sûr si la fonction d'interception exécute un `exit()`. Cependant, compte tenu de la remarque précédente, la fiabilité de l'exécution du processus interrompu et redémarré n'est pas assurée.

Dans l'exemple suivant, taper `Ctrl/c` au clavier n'interrompera pas le processus (action par défaut), mais affichera la chaîne `"Ctrl/c appuyé !"`.

Exemple de code VI.1

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

static void sigHandler(int nosig)
{
   printf("Ctrl/c appuyé !\n");
}

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

  if (signal(SIGINT, sigHandler) == SIG_ERR) {
      perror("erreur en appel de signal() : ");
      exit(1);
  }
  for (j = 0; ; j++) {
      printf("%d\n", j);
      sleep(3);
  }
}
```

Pour interrompre un tel programme, il faudra taper `Ctrl/z` au clavier.

2.3 Inhibition d'un signal

`SIG_IGN` indique que le signal sera ignoré. Cette action ne peut s'appliquer ni au signal `SIGKILL`, ni à `SIGSTOP`. Exemple :

```
signal (SIGINT, SIG_IGN);
```

Ceci a pour effet de rendre inopérant le signal d'interruption (touche `Ctrl/c`). Cet appel de `signal()` est typiquement utilisé pour protéger l'exécution d'un processus fils par exemple que l'on désire voir aller à son terme, tandis que l'on se réserve la possibilité d'interrompre les autres processus s'exécutant simultanément. Il faut être averti du fait que le système Linux passe les signaux issus du clavier (frapper `Ctrl/c` par exemple) à tous les processus qui ont été lancés à partir du même terminal. Cela signifie que si un processus père est lancé depuis le clavier, qu'il génère des fils, la touche `Ctrl/c` interrompra à la fois le père et les fils, sauf si certains processus ont été protégés par l'appel `signal()` avec `SIG_IGN`.

Exemple de code VI.2

```
#include <signal.h>

main(void)
{
  if (fork() == 0)
  {
      signal(SIGINT, SIG_IGN);
      /* ce fils est protégé d'une interruption par crtl/c*/
  }
  else
  {
      /* processus père */
  }
}
```

Remaque VI.1. Le signal `SIGCHLD` se comporte différemment de tous les autres signaux. Lorsqu'il est armé à la valeur `SIG_IGN` dans un processus père, les effets de `exit()` dans les processus fils et de `wait()` dans le père sont changés. Quand un processus fils exécute `exit()`, il ne passe pas à l'état zombie, la table des processus est nettoyée immédiatement. D'autre part, quand le processus père exécute `wait()`, il attend la fin d'exécution de tous les processus fils et retourne alors `-1` avec le code d'erreur `ECHILD`. Un seul `wait()` suffit donc pour attendre dans ce cas la fin de tous les fils quel que soit leur nombre.

2.4 Restauration de l'action par défaut

Il est possible de restituer action par défaut d'un signal grâce à `SIG_DFL`. L'appel `signal(SIGINT, SIG_DFL);` restaure l'action normale du signal d'interruption `Ctrl/c`. À réception du signal `SIGINT`, `SIG_DFL` entrainera la terminaison du processus. Il est

possible également de restaurer l'action d'un signal en sauvegardant l'adresse retournée par `signal()`, puis d'utiliser cette valeur dans un appel ultérieur à signal(). En voici un exemple :

Exemple de code VI.3

```
void (*ancien)();

ancien = signal(sigtype,fonc);
/* nouvelle action définie par fonc()*/
   ...........
signal(sigtype, ancien);
/* restauration de l'action par défaut */
```

À la réception d'un signal, l'action par défaut a pour conséquences de :

- Fermer les descripteurs de fichiers du processus courant, et d'un façon générale, d'exécuter toutes les actions réalisées par un `exit()`.
- Réactiver le processus père si celui-ci est en `wait()`.
- Dans le cas où le père est en cours d'exécution, le processus fils reste à l'état zombie jusqu'à ce que le père exécute un `wait()` ou se termine.
- Créer un fichier "core" dans le répertoire courant pour les signaux `SIGQUIT,SIGILL, SIGTRAP, SIGIOT, SIGEMT, SIGFPE, SIGBUS, SIGSEGV, SIGSYS`.
- Générer une entrée dans le fichier d'accounting du système.

Dans l'exemple ci-dessous, le comportement par défaut est restauré au bout de 6 appuis sur touche `Ctrl/c` :

Exemple de code VI.4

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

static void sigHandler(int nosig)
{
  static int i = 0;

  if (i < 5) {
      printf("Ctrl/c appuyé !\n");
      i++;
  } else
      signal(SIGINT, SIG_DFL);

}
```

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

  if (signal(SIGINT, sigHandler) == SIG_ERR) {
      perror("erreur en appel de signal() : ");
      exit(1);
  }
  for (j = 0; ; j++) {
      printf("%d\n", j);
      sleep(3);
  }
}
```

3 Envoi d'un signal

L'appel système `kill()` permet d'envoyer d'un processus à un autre un signal donné. Sa syntaxe est la suivante :

```
#include <signal.h>
int kill(            Envoi d'un signal
    int pid,         identification du processus récepteur
    int nosig)       numéro du signal à envoyer
```

Cette appel système renvoie `0` en cas de succès et `-1` en cas d'erreur.

Description :

Le processus émetteur du signal et le processus récepteur doivent avoir le même propriétaire, exception faite du super-user. L'appel système `kill()` envoie au processus de numéro `pid` le signal spécifié par `nosig`.

Si `pid == 0`, le signal est envoyé à tous les processus appartenant au même groupe.

Si `pid == -1`, le signal est envoyé à tous les processus appartenant au même utilisateur. Si le super-user exécute un `kill()` avec `pid == -1`, tous les processus excepté les processus `0` et `1` sont tués. Enfin, si `pid` est négatif mais différent de `-1`, le signal spécifié est envoyé à tous les processus dont le process group ID est égal à la valeur absolue de `pid`.

4 Réactivation d'un processus : `alarm()` et `pause()`

L'appel système `alarm()` permet de générer le signal `SIGALRM` après une certaine durée exprimée en secondes. Sa syntaxe est la suivante :

```
#include <unistd.h>
unsigned alarm(      Génération programmée d'envoi de SIGALRM
    unsigned sec)    Nombre de secondes avant l'envoi du signal
```

Description :
Après le temps spécifié sec exprimé en secondes, le signal SIGALRM sera envoyé par le système au processus appelant (entrainant normalement sa fin). Chaque appel à alarm() ré-initialise l'horloge déterminant l'émission de SIGALRM (i.e., dans des appels successifs, seul le dernier est pris en compte). Si sec == 0, les appels antérieurs à alarm() sont annulés.

Cet appel système est typiquement utilisé pour terminer un processus au bout d'un délai donné considéré comme maximal pour exécuter une tâche.

L'appel système pause() suspend le processus appelant jusqu'à ce qu'il reçoive un signal. Sa syntaxe est la suivante :

`#include <unistd.h>`	
`int pause(void)`	Suspend le processus jusqu'à réception d'un signal

Le processus étant suspendu, à la réception d'un signal il sera terminé ou si le signal est intercepté il sera réactivé, pause() retournant dans ce cas -1 avec le code d'erreur EINTR.

Voici un exemple d'utilisation de pause() qui attribue également le même gestionnaire à deux signaux :

Exemple de code VI.5

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

static void sigHandler(int sig)
{
  static int count = 0;

  if (sig == SIGINT) {
      count++;
      printf("SIGINT genere %d fois\n", count);
      return; // Revenir au point d'interruption
  }
  /* le signal doit etre SIGQUIT - afficher un message et terminer */
  printf("SIGQUIT intercepte - Au revoir!\n");
  exit(0);
}

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

```
    /* Etablit le meme gestionnaire pour SIGINT et SIGQUIT */
    if (signal(SIGINT, sigHandler) == SIG_ERR) {
        perror("erreur en appel de signal() : ");
        exit(1);
    }
    if (signal(SIGQUIT, sigHandler) == SIG_ERR) {
        perror("erreur en appel de signal() : ");
        exit(1);
    }  /* Boucle infinie en attente de signaux */
    for (;;)
        pause();  // Bloque jusqu'a ce qu'un signal soit intercepte
}
```

À réception du signal `SIGINT` (touches `Ctrl/c` enfoncées), le programme affiche une chaîne ainsi que le nombre d'interceptions de ce signal ; à réception de SIGQUIT (touches `Ctrl/\` enfoncées), le programme affiche un message et se termine.

5 Signaux Posix

5.1 Actions sur un ensemble de signaux

Signal Sets Plusieurs appels systèmes relatifs aux signaux ont besoin d'une représentation d'un groupe de signaux. C'est le case de `sigaction()` (l'équivalent posix de `signal()`). Un groupe de signaux est représenté par une structure nommée ensemble de signaux, correspondant au type `sigset_t`.

La fonction `sigemptyset()` initialise un ensemble de signaux vide. La fonction `sigfillset()` initialise un ensemble contenant tous les signaux.

L'une des fonctions `sigemptyset()` ou `sigaddset()` doit être utilisée pour initialiser un ensemble de signaux. Après initialisation, un signal peut être ajoutés à un ensemble en utilisant `sigaddset()` ou enlevé en utilisant `sigdelset()` :

```
#include <signal.h>
int sigaddset(        Ajout d'un signal
    sigset_t *set,    ensemble à qui ajouter
    int nosig)        numéro du signal à ajouter

int sidelset(         Retrait d'un signal
    sigset_t *set,    ensemble à qui ajouter
    int nosig)        numéro du signal à ajouter
```

Chacune renvoie `0` en cas de succès et `-1` en cas d'erreur.

La fonction `sigismember()` peut être utilisée pour tester l'appartenance à un ensemble.

```
#include <signal.h>
int sigismember(      Teste l'appartenance d'un signal
```

```
    sigset_t *set,    ensemble au sein duquel tester
    int nosig)        numéro du signal à tester
```

Cette fonction renvoie `1` si `nosig` appartient à `set` et `0` sinon.

La bliothèque `GNU C` fournit l'implantation de trois fonctions non standard qui effectuent des tâches complémentaires.

```
#define _GNU_SOURCE
#include <signal.h>
int sigandset(sigset_t *dest, sigset_t *left, sigset_t *right );
int sigorset(sigset_t  *dest, sigset_t *left, sigset_t *right );
```

Chacune renvoie `0` en cas de succès et `-1` en cas d'erreur. La fonction `sigisemptyset()` de syntaxe

```
int sigisemptyset(const sigset_t * set );
```

Renvoie `1` si `set` est vide et `0` sinon. Ces fonctions effectuent les tâches suivantes :

- `sigandset()` écrit l'intersection des ensembles `left` et `right` dans l'ensemble `dest` ;
- `sigorset()` écrit l'union des ensembles `left` et `right` dans l'ensemble `dest` ;
- `sigisemptyset()` renvoie `1` si l'ensemble ne contient aucun signal.

5.2 Masque et blocage d'un signal

Pour chaque processus, le noyau maintient un masque des signaux, un ensemble des signaux actuellement bloqués. Si un signal bloqué est envoyé à un processus, l'envoi de ce signal est retardé jusqu'à ce qu'il soit débloqué en étant enlevé du masque des signaux de ce processus (En fait le masque de signaux est un attribut de thread et chaque thread peut indépendamment examiner et modifier son masque de signaux par la fonction `pthread_sigmask()`). Un signal peut être ajouté au masque de l'une des manières suivantes :

- Lorsqu'un gestionnaire de signal est appelé, le signal l'ayant déclenché peut être automatiquement ajouté au masque (ceci est réalisé via des options de `sigaction()`).
- Lorsqu'un gestionnaire de signal est déclaré en utilisant `sigaction()` il est possible de spécifier un ensemble de signaux additionnels qui doivent être bloqués lorsque le gestionnaire est appelé.
- l'appel système `sigprocmask()` peut être utilisé à tout moment pour ajouter ou retirer explicitement des signaux du masque.

La fonction `sigprocmask()` permet de changer et/ou de récupérer le masque d'un signal. La syntaxe en est la suivante :

```
#include <signal.h>
int sigprocmask(          Changer et récupérer le masque d'un signal
    int how,              Type d'opération
    sigset_t *set,        ensemble mis en jeu
    sigset_t *oldset)     ancien masque
```

La fonction renvoie `0` en cas de succès et `-1` en cas d'erreur.

L'argument `how` spécifie les changements opérés par `sigprocmask()` au masque du signal :

`SIG_BLOCK` Les signaux spécifiés dans l'ensemble `set` sont ajoutés au masque. En d'autres termes, le masque est mis à jour avec l'union de sa valeur actuelle et de `set`.

`SIG_UNBLOCK` Les signaux de `set` sont retirés du masque. Débloquer un signal qui n'est pas bloqué ne provoque pas d'erreur.

`SIG_SETMASK` L'ensemble de signaux `set` devient le masque de signaux.

Dans chaque cas, si l'argument `oldset` est différent de `NULL`, il pointe sur l'ancien masque. Si l'on désire récupérer le masque sans le modifier, il faut mettre le deuxième argument `set` à la valeur `NULL` et l'argument `how` est ignoré.

Pour empêcher temporairement l'émission d'un signal, l'on peut utiliser une série d'appels listés ci-dessous afin de bloquer le signal, puis de remettre ce dernier dans son état initial.

Exemple de code VI.6

```
sigset_t blockSet, prevMask;

/* Initialiser un ensemble de signaux a SIGINT */
sigemptyset(&blockSet);
sigaddset(&blockSet, SIGINT);

/* Block SIGINT, save previous signal mask */
/* Bloquer SIGINT, sauvegarder le masque precedent */
if (sigprocmask(SIG_BLOCK, &blockSet, &prevMask) == -1) {
    perror("sigprocmask() ");  exit(1);
}

/* Code qui ne doit pas etre interrompu par SIGINT */

/* restaurer le masque de signal precedent, deblaquant SIGINT */
if (sigprocmask(SIG_SETMASK, &prevMask, NULL) == -1) {
    perror("sigprocmask() ");  exit(1);
}
```

S'il y a plusieurs signaux en attente et que l'on effectue un déblocage par un appel à `sigprocmask()`, au moins l'un de ces signaux sera émis avant que l'appel ne se termine. En d'autres termes si l'on débloque un signal en attente, il est délivré au processus immédiatement. Les essais de blocage de `SIGKILL` et `SIGSTOP` sont silencieuselent ignorés. Le code suivant permettra de bloquer tous les signaux excepté `SIGKILL` et `SIGSTOP` :

```
sigfillset(&blockSet);
if (sigprocmask(SIG_BLOCK, &blockSet, NULL) == -1) {
```

```
    perror("sigprocmask() ");  exit(1);
}
```

5.3 Déclaration de comportement : `sigaction()`

5.3.1 Description générale de `sigaction()`

L'appel système `sigaction()` est une alternative à `signal()` pour modifier le comportement du processus à réception d'un signal. Cet appel est à la fois plus complexe et plus flexible que `signal()` ; il permet en particulier de récupérer le comportement à réception d'un signal sans le modifier et de fixer divers attributs.

`#include <signal.h>`	
`int sigaction(`	Modifier le comportement à réception d'un signal
`int nosig,`	Numéro du signal
`const struct sigaction *act,`	Structure de modification/récupération de comportement
`const struct sigaction *oldact)`	Ancienne structure de comportement

La fonction renvoie `0` en cas de succès et `-1` en cas d'erreur.

L'argument `nosig` identifie le signal dont on veut modifier ou récupérer le comportement. Il peut être tout signal excepté `SIGKILL` ou `SIGSTOP`. L'argument `act` spécifie le nouveau comportement à réception du signal. Si l'on cherche uniquement à récupérer le comportement, on spécifie `NULL` pour cet argument. L'argument `oldact` contient le comportement actuel à réception du signal. Si l'on n'est pas intéressé par cette information, l'on spécifie `NULL` pour cet argument. La structure `sigaction` est de la forme suivante :

```
struct sigaction {
  void     (*sa_handler)(int);   /* Addresse de la routine */
  sigset_t   sa_mask;            /* signaux bloques e l'execution de sa_handler */
  int        sa_flags;           /* drapeau controlant l'appel de sa_handler */
  void     (*sa_restorer)(void); /* usage non autorise dans les applications */
};
```

L'argument `sa_handler` est un pointeur sur fonction analogue à celui passé à `signal()` ; il spécifie soit l'adresse d'une fonction, soit l'une des constantes `SIG_IGN` or `SIG_DFL`. Les champs `sa_mask` et `sa_flags` ne sont utilisés que si `sa_handler` correspond à l'adresse d'une fonction.

Le champ `sa_restorer` est utilisé de manière interne pour restaurer le contexte de la thread interrompue au retour de la routine `sa_handler`.

Le champ `sa_mask` définit un ensemble de signaux qui seront bloqués lors de l'exécution de `sa_handler`. Ils sont automaiquement ajoutés au masque du signal le temps de l'exécution de `sa_handler` et en sont retirés ensuite. Ce champ `sa_mask` permet donc de spécifier un ensemble de signaux qui ne sont pas autorisés à interrompre la routine `sa_handler`. En outre, le signal qui a généré l'appel de `sa_handler` est automaiquement ajouté au masque de signaux du processus. Il n'y aura donc pas d'appel récursif de `sa_handler` en cas de

génération récursive du même signal. Les signaux bloqués n'étant pas mis en file d'attente, si plusieurs signaux sont générés lors de l'exécution de `sa_handler`, un seul sera délivré à la fin de cette exécution.

Le champ `sa_flags` est un masque binaire spécifiant différentes options contrôlant comment le signal est géré. Les bits suivant peuvent être combinés par un OU bit à bit :

`SA_NOCLDSTOP` Si `nosig` est égal à `SIGCHLD`, ne pas générer ce signal lorsqu'un processus fils est stoppé ou redémarré suite à la réception d'un signal.

`SA_NOCLDWAIT` Si `nosig` est égal à `SIGCHLD`, ne pas transformer les fils en zombies lorsqu'ils se terminent.

`SA_NODEFER` Lorsque ce signal est intercepté, ne pas ajouter automaiquement ce signal au masque de signaux du processus lors de l'exécution de `sa_handler`.

`SA_ONSTACK` Appel de `sa_handler` pour ce signal en utilisant une pile différente installée par `sigaltstack()`.

`SA_RESETHAND` Lorsque ce signal est intercepté, remettre son comportement à celui par défaut (i.e., `SIG_DFL`).

`SA_RESTART` Redémarrer automatiquement les appels systèmes interrompus par le gestionnaire `sa_handler`.

`SA_SIGINFO` Appeler `sa_handler` avec des arguments additionnels fournissant des informations supplémentaires sur le signal.

Voici un exemple d'interception multiple de signaux utilisant `sigaction()`. Supposant le code source nommé `expleSigaction.c`, la compilation et le lancement du programme :

```
gcc -o expleSigaction expleSigaction.c
./expleSigaction
```

va afficher le PID du processus. Dans une autre fenêtre terminal, lui envoyer des signaux

```
kill -HUP <pid>
kill -USR1 <pid>
kill -ALRM <pid>
```

où `<pid>` est le PID du processus. Les deux premières lignes génèrent la sortie suivante :

```
Signal SIGHUP intercepte
Signal SIGUSR1 intercepte
```

La dernière génère la sortie suivante

```
Alarm clock
```

et sort du programme (comportement par défaut à réception de `SIGALRM`).

Exemple de code VI.7

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

```
/* Gestionnaire de signal */
void handler(int nosignal)
{
  const char *signal_name;

  // Trouver le signal intercepte
  switch (nosignal) {
      case SIGHUP:
          signal_name = "SIGHUP";
          break;
      case SIGUSR1:
          signal_name = "SIGUSR1";
          break;
      case SIGINT:
          printf("Caught SIGINT, exiting now\n");
          exit(0);
      default:
          fprintf(stderr, "Caught wrong signal: %d\n", nosignal);
          return;
  }
  // Un affichage par printf() dans un gestionnaire de signal
  // est dangereux. Ceci n'est donne qu'a titre d'exemple
  printf("Signal %s intercepte\n", signal_name);
}


int main(void)
{
  /* Déclaration d'une structure pour la mise en place des gestionnaires */
  struct sigaction sa;

  /* affichage du PID du programme */
  printf("PID du programme : %d\n", getpid());
  /* Remplissage de la structure pour la mise en place des gestionnaires */
  /* adresse du gestionnaire */
  sa.sa_handler = &handler;
  /* Mise a zero du champ sa_flags theoriquement ignoré */
  sa.sa_flags = 0;
  /* On ne bloque pas de signaux spécifiques */
  sigemptyset(&sa.sa_mask);

  // Intercepter SIGHUP et SIGINT
  if (sigaction(SIGHUP, &sa, NULL) == -1) {
      perror("Erreur : impossible de gerer SIGHUP"); // Ne devrait pas arriver
  }
```

```
    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("Erreur : impossible de gerer SIGUSR1"); // Ne devrait pas arriver
    }

    // Echouera toujours
    if (sigaction(SIGKILL, &sa, NULL) == -1) {
        perror("Ne peut gerer SIGKILL");        // Arrivera toujours
        printf("Impossible d'intercepter SIGKILL ...\n");
    }

    if (sigaction(SIGINT, &sa, NULL) == -1) {
        perror("Erreur : impossible de gerer SIGINT"); // Ne devrait pas arriver
    }

    /* Boucle infinie - tuer avec un kill -HUP*/
    while (1) ;

    return 0;
  }
```

5.3.2 Obtention d'informations supplémentaires via le drapeau SA_SIGINFO

Le drapeau ou bit `SA_SIGINFO` du masque `sa_flags` de `sigaction()` permet au gestionnaire de comportement du signal d'obtenir des informations supplémentaires lors du déclenchement de ce dernier. La syntaxe du gestionnaire de comportement à réception du signal est alors :

`void handler(`	Gestionnaire de comportement à réception d'un signal
`int nosig,`	Numéro du signal
`siginfo_t *siginfo,`	informations sur le signal
`void *ucontext)`	Contexte utilisateur du processus

Le premier argument `nosig` est le numéro du signal, comme pour le gestionnaire standard. Le deuxième argument `siginfo` est une structure utilisée pour stocker diverses informations sur le signal généré. Puisque le gestionnaire de signal a un prototype différent du prototype standard, on ne peut donc utiliser le champ `sa_handler` de la structure `sigaction` pour spécifier l'adresse (dans le segment de code) du gestionnaire. On doit, à la place, utiliser un champ différent : `sa_sigaction`. En réalité, la défintiion de la structure `sigaction` est un peu plus complexe que ce qui a été décrit en sous section 5.3.1, p. 151. La description complète de la structure est la suivante :

```
struct sigaction {
  union {
      void (*sa_handler)(int);
      void (*sa_sigaction)(int, siginfo_t *, void *);
  } __sigaction_handler;
  sigset_t  sa_mask;
```

```
  int       sa_flags;
  void    (*sa_restorer)(void);
};
/* Les #define suivants rendent aux champs de l'union l'apparence de
   simples champs dans la structure parente */
#define sa_handler    __sigaction_handler.sa_handler
#define sa_sigaction  __sigaction_handler.sa_sigaction
```

La structure `sigaction` utilise une union pour combiner les champs `sa_sigaction` et `sa_handler`. L'uilisation d'une union est possible parce qu'un seul champ est utilisé à la fois. Voici un exemple d'utilisation de `SA_SIGINFO` pour déclarer un gestionnaire de comportement à réception du signal `SIGINT`.

Exemple de code VI.8

```
struct sigaction act;

sigemptyset(&act.sa_mask);
act.sa_sigaction = handler;
act.sa_flags     = SA_SIGINFO;

if (sigaction(SIGINT, &act, NULL) == -1) {
    perror("sigaction() : ");
    exit(1);
}
```

La structure `siginfo_t`

La structure `siginfo_t` passée en deuxième argument à un gestionnaire déclaré au moyen de `SA_SIGINFO` a la forme suivante :

```
typedef struct {
int          si_signo;   /* Signal number */
int          si_code;    /* Signal code */
int          si_trapno;  /* Trap number for hardware-generated signal
                            (unused on most architectures) */
union sigval si_value;   /* Accompanying data from sigqueue() */
pid_t        si_pid;     /* Process ID of sending process */
uid_t        si_uid;     /* Real user ID of sender */
int          si_errno;   /* Error number (generally unused) */
void        *si_addr;    /* Address that generated signal
                            (hardware-generated signals only) */
int          si_timerid; /* (Kernel-internal) Timer ID
                            (Linux 2.6, POSIX timers) */
int          si_overrun; /* Overrun count (Linux 2.6, POSIX timers) */
long         si_band;    /* Band event (SIGPOLL/SIGIO) */
int          si_fd;      /* File descriptor (SIGPOLL/SIGIO) */
```

```
int          si_status;  /* Exit status or signal (SIGCHLD) */
clock_t      si_utime;   /* User CPU time (SIGCHLD) */
clock_t      si_stime;   /* System CPU time (SIGCHLD) */
} siginfo_t;
```

La macro test `_POSIX_C_SOURCE` doit être définie à une valeur supérieure ou égale à `199309` de façon à ce que les déclarations de la structure `siginfo_t` soient visibles dans `<signal.h>`. À l'entrée d'un gestionnaire de signal (la fonction appelée au déclenchement du signal), les champs de la structure `siginfo_t` sont fixés de la manière suivante :

`si_signo` Ce champ est utilisé pour tous les signaux. Il contient le numéro du signal à l'origine de l'appel du gestionnaire ; il s'agit de la même valeur que l'argument `nosig` de ce gestionnaire.

`si_code` Ce champ est utilisé pour tous les signaux. Il contient un code fournissant plus d'information sur l'origine du signal.

`si_value` Ce champ contient les données pour un signal envoyé via `sigqueue()`. Il s'agit de signaux dits temps réel et sigqueue permet, en principe d'envoyer des données, avec le choix d'envoyer un `int` ou un pointeur sur `void` ; en pratique seuls des `int` sont transmis, puisque transmettre un pointeur d'un processus à un autre n'e en général pas grand sens.

`si_pid` Pour des signaux envoyés via `kill()` ou `sigqueue()`, ce champ contient le PID du processus d'où l'envoi est fait.

`si_uid` Pour des signaux envoyés via `kill()` ou `sigqueue()`, ce champ contient l'identificateur utilisateur réel du processus émetteur.

`si_errno` Si ce champ est mis à une valeur non nulle, il contient un numéro d'erreur (comme `errno`) qui identifie la cause du signal. Ce champ est généralement inutilisé sous Linux.

`si_addr` Ce champ n'est actif que pour les signaux générés par du matériel, comme `SIGBUS`, `SIGSEGV`, `SIGILL` et `SIGFPE`. Pour les signaux `SIGBUS` et `SIGSEGV`, ce champ contient l'adresse qui a causé la référence mémoire invalide. Pour les signaux `SIGILL` et `SIGFPE` ce champ contient l'adresse de l'instruction de programme quii a généré ce signal.

Les champs suivants, qui sont des extensions Linux non standard, ne sont actifs qu'à réception d'un signal généré par l'expiration d'un timer Posix (voir Section 7, p. 162).

`si_timerid` Ce champ contient un identifiant que le noyau utilise de manière interne pour identifier le timer.

`si_overrun` Ce champ fournit le dépassement du timer.

Les champs suivants ne sont utilisés que lors du déclenchement d'un signal `SIGIO` (signal envoyé lorsque que le noyau est prêt à réaliser une entrée ou une sortie sur un descripteur de fichier).

`si_band` Ce champ contient `POLLIN | POLLRDNORM` pour une opération de lecture normale ; il contient `POLLPRI | POLLRDBAND` pour une lecture au delà de la fin du descripteur

("out of band read") ; il contient `POLLOUT | POLLWRNORM` pour une écriture normale ; il contient `POLLPRI | POLLWRBAND` pour une écriture au delà de la fin du descripteur ("out of band write") .

`si_fd` Ce champ contient le descripteur de fichier associé à l'événement d'entrée/sortie.

Les champs suivants sont actifs uniquement lors du déclenchement d'un signal `SIGCHLD`

`si_status` Ce champ contient soit le status de sortie (par `exit()`) du fils (si `si_code` est `CLD_EXITED`) ou le numéro du signal envoyé au fils (qui a causé sa terminaison ou son arrêt).

`si_utime` Ce champ contient le temps CPU utilisateur consommé par le processus fils. Ce temps est mesuré en ticks de l'horloge système (depuis le noyau 2.6.27).

`si_stime` Ce champ contient le temps CPU système consommé par le processus fils.

Le champ `si_code` fournit de plus amples informations sur l'origine du signal généré. Les diverses valeurs prises par ce champ sont les suivantes : Pour tout signal

Valeur de `si_code`	Origine du signal
`SI_USER`	Appel de `kill()` par un process utilisateur.
`SI_KERNEL`	Envoyé par le noyau.
`SI_QUEUE`	Appel de sigqueue().
`SI_TIMER`	Expiration de timer POSIX.
`SI_MESGQ`	Arrivée de message sur une queue POSIX.
`SI_ASYNCIO`	E/S asynchrone terminée.
`SI_SIGIO`	Signal SIGIO mis en attente.
`SI_TKILL`	Appel de `tkill()` ou `tgkill()`.

Les valeurs suivantes sont significatives pour un signal de type `SIGILL`

Valeur de `si_code`	Origine du signal
`ILL_ILLOPC`	Code d'opération illégal.
`ILL_ILLOPN`	Opérande illégale.
`ILL_ILLADR`	Mode d'adressage illégal.
`ILL_ILLTRP`	Point d'arrêt illégal
`ILL_PRVOPC`	Code d'opération exigeant des droits privilégiés.
`ILL_PRVREG`	Registre exigeant des droits privilégiés.
`ILL_COPROC`	Erreur de co processeur.
`ILL_BADSTK`	Erreur de pile interne.

Les valeurs suivantes sont significatives pour un signal de type `SIGFPE`

Valeur de `si_code`	Origine du signal
`FPE_INTDIV`	Division par zéro entière.

FPE_INTOVF	Débordement entier.
FPE_FLTDIV	Division par zéro flottante.
FPE_FLTOVF	Débordement flottant.
FPE_FLTUND	Débordement inférieur flottant.
FPE_FLTRES	Résultat inexact en flottant.
FPE_FLTINV	Opération flottante invalide.
FPE_FLTSUB	Élévation à une puissance invalide.

Les valeurs suivantes sont significatives pour un signal de type SIGSEGV

Valeur de si_code	Origine du signal
SEGV_MAPERR	Adresse hors zone mémoire.
SEGV_ACCERR	Accès interdit à la zone mémoire.

Les valeurs suivantes sont significatives pour un signal de type SIGBUS

Valeur de si_code	Origine du signal
BUS_ADRALN	Erreur d'alignement d'adresse.
BUS_ADRERR	Adresse physique invalide.
BUS_OBJERR	Erreur d'adressage matériel.
BUS_MCEERR_AR	Erreur mémoire.
BUS_MCEERR_AO	Erreur mémoire.

Les valeurs suivantes sont significatives pour un signal de type SIGTRAP

Valeur de si_code	Origine du signal
TRAP_BRKPT	Point d'arrêt de débogage.
TRAP_TRACE	Point d'arrêt de profilage.
TRAP_BRANCH	Branchement.
TRAP_HWBKPT	Point d'arrêt/d'inspection matériel.

Les valeurs suivantes sont significatives pour un signal de type SIGCHLD

Valeur de si_code	Origine du signal
CLD_EXITED	Un fils est sorti (par exit()).
CLD_KILLED	Un fils a été tué.
CLD_DUMPED	Un fils s'est terminé de manière anormale.
CLD_TRAPPED	Un fils a atteint un point d'arrêt.
CLD_STOPPED	Un fils a été stoppé.
CLD_CONTINUED	Un fils stoppé a rédémarré.

Les valeurs suivantes sont significatives pour un signal de type `SIGIO/SIGPOLL`

Valeur de `si_code`	Origine du signal
`POLL_IN`	Données prêtes à être lues.
`POLL_OUT`	Tampons de sortie disponibles.
`POLL_MSG`	Message disponible en entrée.
`POLL_ERR`	Erreur d'entrée/sortie.
`POLL_PRI`	Entrées disponibles à haute priorité.
`POLL_HUP`	Déconnexion du périphérique.

Argument `ucontext`

L'argument final passé à un gestionnaire enregistré avec le drapeau `SA_SIGINFO`, `ucontext` est un pointeur sur une structure de type `ucontext_t`, défini dans `<ucontext.h>`. Cette structure fournit le contexte utilisateur du processus (masque de signal, compteur d'instruction, pointeur de pile, etc.). Cette structure `ucontext_t` est utilisée par les fonctions `getcontext()`, `makecontext()`, `setcontext()`, et `swapcontext()` qui permettent respectivement de récupérer, créer, changer et échanger des contextes d'exécution.

Remarques VI.1. (*i*) Les signaux Posix sont portables, alors que les signaux Linux peuvent dépendre du type de système. Il vaut donc mieux utiliser `sigaction()` plutôt que `signal()`.

(*ii*) Il faut retenir que les signaux constituent un dispositif puissant de contrôle des processus mais qui est destiné avant tout à en maîtriser la terminaison. Il ne faut pratiquement jamais utiliser les signaux comme mécanisme de synchronisation d'évènements.

(*iii*) Le seul cas où il est possible de réactiver un processus dans des conditions fiables est le cas d'un processus suspendu par `pause()`, et il faut être certain que le signal de réactivation arrivera pendant l'exécution de ce `pause()`.

(*iv*) L'autre cas d'utilisation fiable des signaux pour la poursuite de l'exécution d'un processus, est celui du changement de contexte.

6 Gestion de l'heure

6.1 Appel `gettimeofday()`

La fonction `gettimeofday()` renvoie l'heure en microsecondes :

```
#include <sys/time.h>

int gettimeofday(           Renvoie l'heure dans le 1er argument
    struct timeval *tv,     Heure courante
    struct timezone *tz)    fuseau horaire
```

Les structures `timeval` et `timezone` sont les suivantes :

```
struct timeval {
    time_t      tv_sec;     /* seconds */
    suseconds_t tv_usec;    /* microseconds */
};

struct timezone {
    int tz_minuteswest;     /* minutes west of Greenwich */
    int tz_dsttime;         /* type of DST correction */
};
```

6.2 Appel système `clock_gettime()`

La fonction `clock_gettime()` renvoie l'heure, avec une précision au maximum de la nanoseconde :

```
#include <time.h>

int clock_gettime(          Renvoie l'heure dans le 2e argument.
    clockid_t clk_id,       Horloge utilisée
    struct timespec *tp)    Structure stockant l'heure
```

L'horloge spécifiée par l'argument `clk_id`, dont le type `clockid_t` peut être l'un de ceux repérés dans la table VI.18 ci-dessous. La valeur temporelle mesurée par `clock_gettime()`

Mnémonique d'horloge	Type d'horloge
`CLOCK_REALTIME`	Horloge temps réel système dont on peut modifier l'heure
`CLOCK_MONOTONIC`	Horloge monotone dont on ne peut pas modifier l'heure (sous Linux, le temps depuis de dernier démarrage du système)
`CLOCK_PROCESS_CPUTIME_ID`	Horloge mesurant le temps CPU par processus
`CLOCK_THREAD_CPUTIME_ID`	Horloge mesurant le temps CPU par thread

TABLE VI.18 – Types d'horloges pour appels de gestion de l'heure.

est renvoyée dans la structure de type `timespec` pointée par `tp`. Bien que la structure `timespec` permette une précision de la nanoseconde, la granularité de la valeur renvoyée par `clock_gettime()` peut être plus grossière. L'appel système `clock_getres()` renvoie, dans son deuxième argument, un pointeur sur une structure de type `timespec` contenant la résolution de l'horloge spécifiée par `clk_id` :

```
#include <time.h>

int clock_getres(           Renvoie la précision de l'horloge clk_id dans res.
    clockid_t clk_id,       Horloge utilisée
    struct timespec *res)   Structure stockant la résolution de l'horloge
```

La structure `timespec` est la suivante :

```
struct timespec {
    time_t   tv_sec;        /* seconds */
    long     tv_nsec;       /* nanoseconds */
};
```

L'appel système `clock_settime()` fixe l'heure de l'horloge spécifiée par `clockid` à la valeur pointée par `tp` :

```
#define _POSIX_C_SOURCE 199309
#include <time.h>

int clock_settime(                  Fixe l'heure de l'horloge clk_id à *tp.
    clockid_t clk_id,               Horloge utilisée
    struct timespec *tp)            Structure stockant l'heure fixée
```

Cet appel système renvoie `0` en cas de succès et `-1` en cas d'erreur.

Si le temps spécifié par `tp` n'est pas un multiple de la résolution de l'horloge tel que renvoyé par `clock_getres()`, ce temps est arrondi par valeur inférieure. Un processus privilégié (`CAP_SYS_TIME`) peut modifier l'heure de l'horloge `CLOCK_REALTIME`. Aucune des autres horloges citées ci-dessus n'a son heure modifiable.

6.3 Mise en sommeil à granularité fine : `clock_nanosleep()`

D'une manière analogue à l'appel `nanosleep()`, l'appel système spécifique Linux `clock_nanosleep()` suspend le processus appelant jusqu'à ce qu'un intervalle de temps spécifié soit écoulé ou bien jusqu'à l'arrivée d'un signal.

```
#define _XOPEN_SOURCE 600
#include <time.h>

int clock_nanosleep(                        Fixe l'heure de l'horloge clk_id à *tp.
    clockid_t clk_id,                       Horloge utilisée
    int flag,                               Temps absolu ou relatif
    const struct timespec *request)         Structure stockant l'heure fixée
    struct timespec *remain)                Structure stockant l'heure fixée
```

Cet appel système renvoie `0` en cas de succès et un numéro d'erreur positif en cas d'erreur ou d'interruption.

L'argument `request` spécifie la durée de l'intervalle de sommeil ; c'est un pointeur sur une structure de la forme suivante :

```
struct timespec {
    time_t tv_sec;  /* Secondes     */
    long   tv_nsec; /* Nanosecondes */
};
```

Le champ `tv_nsec` spécifie une valeur en nanosecondes. Ce doit être une valeur entre `0` et `999999999` (Le nombre 999 999 999). L'appel système `clock_nanosleep()` ne doit

pas être implanté en utilisant des signaux ; néanmoins, il peut être interrompu par un gestionnaire de signaux. Dans ce cas, `clock_nanosleep()` renvoie `-1` avec `errno` mis à `EINTR` ; si l'argument `remain` n'est pas `NULL`, le tampon sur lequel il pointe contient le temps de sommeil non consommé. Il est possible d'utiliser cette valeur pour appeler à nouveau cet appel système et compléter le sommeil.

Par défaut, (c.à.d. si l'argument `flag` est égal à `0`), l'intervalle de sommeil spécifié dans `request` est relatif. Par contre, si `flag` est égal à `TIMER_ABSTIME` la requête spécifie un temps absolu mesuré par l'horloge `clockid`. Cette possibilité est essentielle pour des applications qui doivent dormir un temps bien précis jusqu'à un instant donné. Lorsque le drapeau `TIMER_ABSTIME` est spécifié, l'argument `remain` est inutilisé. Si l'appel `clock_nanosleep()` est interrompu par un gestionnaire de signal, alors le sommeil peut être relancé en répétant l'appel avec le même argument `request`.

Il est possible de spécifier l'horloge au travers l'argument `clk_id` ; ce peut être : `CLOCK_REALTIME`, `CLOCK_MONOTONIC`, ou `CLOCK_PROCESS_CPUTIME_ID` (voir la table VI.18, p. 160). L'exemple suivant illustre l'utilisation de `clock_nanosleep()` afin de dormir pendant 20 secondes en utilisant l'horloge `CLOCK_REALTIME` utilisant une valeur de temps absolue.

Exemple de code VI.9

```
struct timespec request;

/* Recuperer l'heure de l'horloge CLOCK_REALTIME */
if (clock_gettime(CLOCK_REALTIME, &request) == -1){
        fprintf(stderr, "Erreur a l'appel de clock_gettime()");
        exit(1);
    }
request.tv_sec += 20;
/* Dormir pendant 20 secondes */
s = clock_nanosleep(CLOCK_REALTIME, TIMER_ABSTIME, &request, NULL);
if (s != 0) {
    if (s == EINTR)
        printf("Interrompu par un gestionnaire de signal\n");
    else {
        fprintf(stderr, "Erreur a l'appel de clock_nanosleep()");
        exit(1);
    }
}
```

7 Timers Posix

Les timers permettent l'activation de tâches périodiques, soit par envoi périodique d'un signal, soit par activation périodique d'une thread.

La création d'un timer posix s'effectue via la fonction `timer_create()` :

```
#include <timer.h>

int timer_create(                       Créer un timer posix.
    clockid_t clock,                    Type d'horloge
    struct sigevent *notification,      Type d'activation
    timer_t *timer)                     Descripteur de timer
```

Le type d'horloge est à choisir par les suivants :

`CLOCK_REALTIME`	Horloge système (heure absolue, réglable).
`CLOCK_MONOTONIC`	Horloge non modifiable.
`CLOCK_PROCESS_CPUTIME_ID`	Horloge indiquant le temps CPU consommé par les threads du processus.
`CLOCK_THREAD_CPUTIME_ID`	Horloge indiquant le temps CPU consommé par la thread appelante.
`CLOCK_MONOTONIC_RAW`	Horloge indiquant l'heure, non modifiable, fournie par le matériel (spécifique Linux, non Posix).

Les champs de la structure `notification` ont la signification suivante (chaque nom de champ commence par la chaîne `sigev_`) :

`champ`	Signification et valeurs possibles
`sigev_notify`	Type de notification : – `SIGEV_NONE` : aucune – `SIGEV_SIGNAL` : par envoi d'un signal – `SIGEV_THREAD` : par activation d'une thread – `SIGEV_THREAD_ID` : par envoi d'un signal à une thread spécifique
`sigev_signo`	Numéro du signal à envoyer si la notification se fait par signal.
`sigev_value`	Si notification par signal, valeur accompagnant le signal. Si notification par activation de thread, valeur passée en argument à la fonction.
`sigev_notify_function`	Si notification par activation de thread, fonction à démarrer.
`sigev_notify_attributes`	Si notification par activation de thread, attributs de la thread lancée.
`sigev_notify_thread_id`	Si notification de type `SIGEV_THREAD_ID`, identificateur de la thread devant recevoir le signal.

La description de la structure `sigevent` que l'on trouve dans le chapitre 7 du manuel (en tapant `man 7 sigevent` au clavier) est la suivante :

```
union sigval {                  /* Data passed with notification */
  int     sival_int;            /* Integer value */
  void   *sival_ptr;            /* Pointer value */
};

struct sigevent {
  int            sigev_notify; /* Notification method */
  int            sigev_signo;  /* Notification signal */
  union sigval   sigev_value;  /* Data passed with notification */
  void         (*sigev_notify_function) (union sigval);
                                /* Function used for thread
                                   notification (SIGEV_THREAD) */
  void          *sigev_notify_attributes;
                                /* Attributes for notification thread
                                   (SIGEV_THREAD) */
  pid_t          sigev_notify_thread_id;
                      /* ID of thread to signal (SIGEV_THREAD_ID) */
};
```

Une fois le timer créé on le configure et l'active via la fonction `timer_settime()` (l'on fixe la période et le délai avant la première activation) :

`int timer_settime(`	Configurer un timer posix.
`timer_t *timer,`	Descripteur du timer
`int absolu,`	Heure absolue (si `absolu == TIMER_ABSTIME`) ou durée (si `absolu == 0`).
`const struct itimerspec *spec,`	Période et expiration initiale actuelle.
`struct timerspec *precedente)`	Période et expiration initiale passée (au cas où l'on voudrait la réactiver).

Les structures `timespec` et `itimerspec` telles qu'elles apparaissent dans la page de manuel de `timer_settime()` sont les suivantes :

```
struct timespec {
  time_t tv_sec;                /* Seconds */
  long   tv_nsec;               /* Nanoseconds */
};

struct itimerspec {
  struct timespec it_interval;  /* Timer interval */
  struct timespec it_value;     /* Initial expiration */
};
```

Voici un exemple d'utilisation des fonctions `int timer_create()` et `int timer_settime()`, où l'on créé deux timers avec des périodes respectives d'une seconde et d'un quart de

seconde :

Exemple de code VI.10

```
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <unistd.h>
#include <sys/time.h>

void handler_signal_1(int inutilise)
{
    fprintf(stderr, "1 ");
}

void handler_signal_2(int inutilise)
{
    fprintf(stderr, "2 ");
}


int main(int argc, char * argv[])
{
    timer_t timer_1, timer_2;
    struct sigevent event_1, event_2;
    struct itimerspec spec_1, spec_2;

    // Installer les handlers
    signal(SIGRTMIN+1, handler_signal_1);
    signal(SIGRTMIN+2, handler_signal_2);

    // Indiquer la notification desiree
    event_1.sigev_notify = SIGEV_SIGNAL;
    event_1.sigev_signo  = SIGRTMIN+1;

    // Configurer la periode du timer
    spec_1.it_interval.tv_sec  = 1;
    spec_1.it_interval.tv_nsec = 0;
    spec_1.it_value = spec_1.it_interval;

    // Allouer le timer
    if (timer_create(CLOCK_REALTIME, & event_1, & timer_1) != 0) {
        perror("timer_create");
        exit(EXIT_FAILURE);
```

```
    }

    // Memes operations pour le second timer
    event_2.sigev_notify = SIGEV_SIGNAL;
    event_2.sigev_signo  = SIGRTMIN+2;
    spec_2.it_interval.tv_sec  = 0;
    spec_2.it_interval.tv_nsec = 250000000; // 0,25 sec.
    spec_2.it_value = spec_2.it_interval;
    if (timer_create(CLOCK_REALTIME, & event_2, & timer_2) != 0) {
        perror("timer_create");
        exit(EXIT_FAILURE);
    }

    // Programmer les timers
    if ((timer_settime(timer_1, 0, &spec_1, NULL) != 0)
     || (timer_settime(timer_2, 0, &spec_2, NULL) != 0)) {
        perror("timer_settime");
        exit(EXIT_FAILURE);
    }
    while (1)
        pause();
    return EXIT_SUCCESS;
}
```

Pour arrêter et détruire un timer, on utilise `timer_delete()` :

`int timer_delete(`	Détruire un timer posix.
`timer_t *timer)`	Descripteur du timer.

Partie E

Linux temps réel

VII – Notions pour le temps réel

1 Introduction

Nous examinerons dans ce chapitre les notions suivantes :

- *Notion de temps réel*. Nous verrons notamment que temps réel n'est pas synonyme de temps court.
- *Ordonnancement*. Nous examinerons deux algorithmes classiques d'ordonnancement dit statique : FIFO et RR, puis deux algorithmes classiques d'ordonnancement dit dynamique : RM et EDF.
- *Ordonnançabilité*, ou comment faire en sorte qu'un ordonnancement respecte les contraintes temporelles de chaque tâche. Nous examinerons des conditions nécessaires et suffisantes ou simplement suffisantes dans les cas statique et dynamique.
- *Ordonnançabilité et stabilisabilité*. De l'insuffisance notoire de l'ordonnançabilité pour les systèmes dynamiques.
- *Programmation concurrente*. Quelques mécanismes classiques seront passés en revue : sémaphores, mutex, variable de condition.
- *Architecture logicielle*. Nous distinguerons d'une part des schémas architecturaux, tels qu'ils sont par exemple envisagés dans l'excellent livre de Doug Lea ; patrons de conception temps réel.

2 Temps réel et contraintes de temps

2.1 Notion de temps réel

Un système est dit temps réel s'il est capable de fournir une réponse valide en un temps déterminé. Cela ne veut pas nécessairement dire que ce système doive répondre dans un temps très court. Le point essentiel est que sa réactivité (c.à.d. le temps qu'il met à répondre) doit être prédictible. La notion suivante de temps réel vient de Donald Gillies (Senior Staff Engineer de Qualcomm, Inc et Senior Software Engineer de Google) et a été adoptée comme définition canonique par le newsgroup `comp.realtime` (`http://www.faqs.org/faqs/realtime-computing/faq/`).

"Un système est dit *temps réel* si l'exactitude de ses calculs ne dépend pas seulement de l'exactitude logique de ses calculs mais également de l'instant auquel le résultat est produit.

Si les contraintes temporelles ne sont pas remplies, cela correspond à un cas d'échec."

Autrement dit :

> *Un programme est dit* ***temps réel*** *s'il doit fournir un résultat juste (comme tout programme correct) en un temps déterminé, que l'on nomme échéance temporelle.*

Comme nous venons de le souligner, le caractère temps réel n'est pas synonyme de temps court ou d'optimisation sur le seul critère temporel. Il est intimement lié à la notion de *prédictibilité temporelle*, qui est cruciale pour ce type de systèmes.

> *Le comportement d'un système est dit* ***déterministe*** *si ses performances temporelles restent identiques quel que soit le contexte d'exécution. Un système est dit* ***prédictible*** *s'il est déterministe et s'il est possible de déterminer ses performances temporelles a priori.*

2.2 Délais et temps de réponse

Il s'agit de respecter les délais de bout en bout. Un peu plus de détails est donné dans la figure suivante :

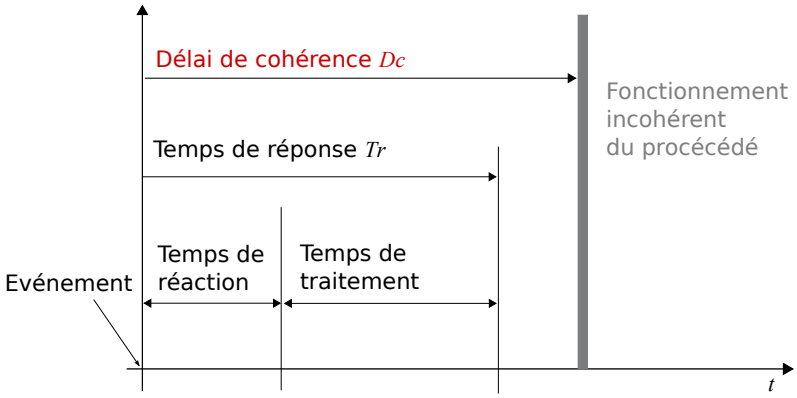

FIGURE VII.1 – Temps de réaction, de réponse et délai de cohérence.

> *Le* ***temps de réaction*** *est le temps mis par le système informatique pour reprendre la main après arrivée d'un événement. Le* ***temps de réponse*** *est le temps mis par ce même système pour répondre à un événement. Le* ***délai de cohérence*** *est le délai maximum entre l'arrivée d'un événement et son traitement, avant un comportement incohérent du procédé.*

Les temps de réponse dont il est question ci-dessus ne sont pas nécessairement courts dans un système temps réel. Ils dépendent fortement du type d'application envisagé. Voici quelques ordres de grandeur de temps de réponse typiques :

Type d'application	Temps de réponse typique
Radar	1ms
Système d'anti patinage (ABS)	5ms
Contrôle de stocks	30s
Réacteur chimique	1h

Différents facteurs peuvent entrer en jeu dans le temps de réponse du système informatique :

- Le cadencement du processeur, du bus et des périphériques.
- Les propriétés de l'ordonnanceur
- La préemptivité du noyau (voir la notion en Section 3.1, p. 174)
- La charge du système (le nombre de tâches prêtes, qui attendent de tourner sur le processeur).
- Le temps de changement de contexte (voir ci-dessous).

*Le **temps de changement de contexte** est le temps que met le processeur pour sauvegarder le contexte d'une tâche (les registres, le pointeur de pile, le pointeur d'instructions, etc.) et pour charger le contexte de la prochaine tâche devant s'exécuter.*

2.3 Catégories de temps réel

On distingue classiquement plusieurs types de temps réel. Le *temps réel dur* concerne les systèmes où il est impératif que les réponses arrivent avant échéance, comme par exemple dans des systèmes de contrôle d'avions. Dans des systèmes de type *temps réel mou*, les échéances peuvent être occasionnellement dépassées ; un exemple est un système d'acquisition de données. Le *temps réel effectif* est un type de *temps réel dur* avec des temps de réponse très courts, tel un système de guidage de missile. Le *temps réel ferme* est un type de *temps réel mou* sans bénéfice de retard à l'accomplissement de ses services. Voici deux exemples typiques de systèmes temps réel :

- Un bras de robot doit s'emparer d'un objet sur un tapis roulant. La pièce bouge et le robot a une petite fenêtre temporelle pour attraper l'objet. Si le robot est en retard, la pièce ne sera plus là et la tâche sera réalisée de manière incorrecte, bien que le bras de robot se soit déplacé au bon endroit. Si le robot est en avance, la pièce n'est pas encore là et le déplacement du bras peut bloquer le mouvement de la pièce.

– Les boucles de rétroaction d'un pilote automatique d'avion constituent un autre exemple. Les capteurs de l'avion doivent continuellement fournir des données au microprocesseur de commande. Si une rafale de données est manquantes, les performances de l'avion peuvent se dégrader, certaines fois à un niveau inacceptable.

Notons que l'immense majorité des systèmes temps réel durs sont des systèmes de commande ; ils comprennent des tâches d'acquisition capteur, des tâches de mise en forme et de filtrage des données ; des tâches d'extraction de caractéristiques et d'estimation de variables et de paramètres ; des tâches de calcul de lois de commande de suivi de trajectoire ; des tâches de restitution actionneur. Ce type de système repose donc de manière fondamentale sur des algorithmes de traitement du signal, de traitement des images et d'automatique.

2.4 Caractéristiques d'un système temps réel

Un système temps réel se distingue d'autres types de systèmes informatiques par différentes caractéristiques :

– Taille et complexité
– Contrôle concurrent
– Interaction avec le matériel
– Fiabilité et sûreté
– Temps de réponse
– Implantation

Taille et complexité. Un système temps réel n'est pas nécessairement grand et complexe. Ce sera le cas pour une centrale nucléaire, mais nettement moins pour un ascenseur.

Contrôle concurrent. Contrôle concurrent des composantes séparées du système périphériques opérant en parallèle dans le monde réel, modélisées par des entités concurrentes.

Interaction avec le matériel. Il est nécessaire de disposer de facilités pour interagir avec des parties matérielles spécialisées. En d'autres termes de pouvoir programmer ces périphériques d'une manière abstraite, fiable et souple.

Fiabilité et sûreté. Une fiabilité et sûreté absolues sont requises. En effet, tout système embarqué temps réel contrôle son environnement ; des défaillances peuvent résulter en des pertes de vies humaines, ou à des pertes financières importantes. Prenons deux exemples : premièrement, un dépassement de délai de cohérence d'un système de transport peut entraîner des pertes de vies humaines. Deuxièmement, un réacteur de polymérisation (réacteur de 300m de long produisant des fils de PVC) pris en masse (par adjonction au mauvais moment de catalyseur qui a fait prendre le polymère en masse dans le réacteur) doit être arrêté pendant plusieurs jours, le temps de casser et d'enlever le polymère du réacteur, d'où une perte financière considérable en jours de production.

Temps de réponse. Les temps de réponse doivent être garantis sur ce type de système. Le point de vue souvent adopté, faute de mieux, est la nécessité de prévoir de manière fiable les pires temps de réponse des systèmes. L'efficacité temporelle est importante, mais une bonne prévision est cruciale. La prévision du temps de calcul peut être simple pour des tâches d'acquisition capteur (de temps de calcul presque constant) ou difficile à prévoir pour d'autres tâches comme celles de détection de piétons (dont bon nombre d'algorithmes ont un temps de calcul dépendant directement de la complexité de la scène visuelle).

Implantation. Une implantation efficace est essentielle. L'efficacité n'est pas synonyme de rapidité temporelle mais d'abord et avant tout de bonne conception au sens du génie logiciel. La conception d'un système temps réel doit tout d'abord se faire en privilégiant quatre buts généraux du génie logiciel, tels que mentionnés en Section 8, p. 194.

3 Ordonnancement

3.1 Notion d'ordonnancement, de préemption, de file d'attente

Une première notion essentielle est celle d'ordonnancement (et de répartition).

*Un **ordonnanceur** est une tâche ou une fonction qui sélectionne quelle tâche doit tourner sur le processeur. L'**ordonnancement** consiste en l'ordre et la distribution temporelle avec lesquelles les tâches doivent s'exécuter, démarrer et s'arrêter. L'opération spécifique d'allouer le CPU à une tâche élue par l'ordonnanceur est effectuée par un **répartiteur**.*

Une deuxième est celle de préemption.

*Une tâche `A` est dite **préemptée** par la tâche `B` si, la tâche `A` étant en train de tourner sur le processeur, elle est interrompue et la tâche `B` s'exécute sur le processeur immédiatement. Un noyau est dit **préemptif** si toute tâche de ce noyau peut être préemptée par un autre noyau ou par une tâche utilisateur.*

Nous aurons également besoin de la notion de file d'attente.

*Une **file d'attente** est une structure de donnée linéaire (tableau, liste chaînée) dans laquelle sont stockées des structures représentant des tâches. La **file d'attente des tâches prêtes** est la file d'attente contenant les structures des tâches prêtes à s'exécuter (en attente d'exécution) sur le processeur.*

3.2 Machine d'état d'une tâche

Les tâches (processus ou threads) d'un système d'exploitation sont dans un nombre fini d'états. Une tâche qui vient d'être créée est mise dans la file d'attente des tâches prêtes et passe dans l'état "Prêt". Les différents états, représentés en Figure VII.2, sont typiquement les suivants.

- *Prêt.* La tâche est prête à être élue et se trouve stockée dans la file d'attente des tâches prêtes (représentée par exemple par une liste chaînée).
- *En cours (d'exécution).* La tâche a été élue par l'ordonnanceur et tourne sur le (ou l'un des) processeur(s).
- *Bloqué.* La tâche est bloquée, par exemple en attente d'une ressource d'entrée/sortie, d'un périphérique.
- *Prêt suspendu.* Une tâche prête est suspendue, par exemple lors du débogage ou s'il s'agit d'une tâche périodique en attente de sa prochaine date de réveil.
- *Bloqué suspendu.* Une tâche bloquée est suspendue, typiquement lorsqu'il n'y a plus assez de place en mémoire vive pour toutes les tâches bloquées ; elle est alors permutée (swapped) sur disque (de la mémoire secondaire constituée par une partition spéciale sur le disque appelée partition d'échange ou "swap partition").
- *Mort.* Une tâche qui se termine passe et reste dans cet état.

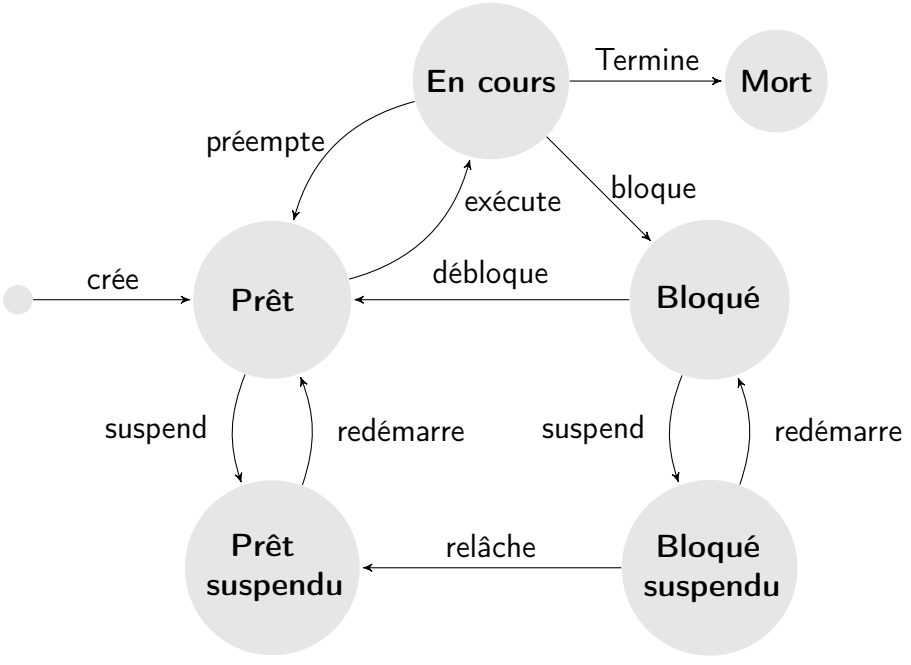

FIGURE VII.2 – Machine d'états d'une tâche.

3.3 Types d'ordonnancement

3.3.1 Grandes classes d'ordonnancement

Différents critères permettent de caractériser les types d'ordonnancement existants. On distingue typiquement les classes suivantes.

- *Préemptive ou non-préemptive.*
  Dans un ordonnancement de type préemptif, la tâche en cours d'exécution peut être interrompue à tout instant afin d'assigner le processeur à une autre tâche. Dans un ordonnancement de type non préemptif, une tâche une fois démarrée s'exécute sur le processeur jusqu'à complétion. Toute décision d'ordonnancement est prise lorsque la tâche termine son exécution.
- *Statique ou dynamique.*
  Un algorithme statique est tel que les décisions d'ordonnancement se fondent sur des paramètres fixes, assignés aux tâches avant leur activation. Un algorithme dynamique est tel que les décisions d'ordonnancement se fondent sur des paramètres dynamiques, qui sont susceptibles de changer au cours de l'évolution du système.
- *Hors ligne ou en ligne.*
  Un algorithme d'ordonnancement est utilisé hors ligne s'il est exécuté sur tout l'ensemble des tâches avant leur activation. L'ordonnancement généré de cette manière est stocké dans une table qui sera exécuté ensuite par un répartiteur. Un algorithme d'ordonnancement est utilisé en ligne si les décisions d'ordonnancement sont prises en cours d'exécution à chaque fois qu'une nouvelle tâche est créée ou se termine.
- *Optimal ou heuristique.*
  Un algorithme est dit optimal s'il minimise une fonction de coût définie sur l'ensemble des tâches. Lorsqu'aucune fonction de coût n'est définie et que le seul but est d'obtenir un ordonnancement réalisable, un algorithme est dit optimal s'il est capable de trouver un ordonnancement réalisable, s'il existe. Un algorithme est dit heuristique s'il est guidé par une fonction heuristique pour prendre ses décisions d'ordonnancement. Un algorithme heuristique tend à être optimal, mais ne peut garantir l'être.

3.3.2 Grands types d'ordonnancement

Nous allons énoncer brièvement les différents types les plus répandus d'alogorithmes d'ordonnancement, dont certains seront détaillés dans ce qui suit.

Exécutif cyclique ou FIFO. Dans un exécutif cyclique ou FIFO, l'ordonnanceur boucle sur un ensemble de tâches. Chaque tâche élue s'exécute jusqu'à sa complétion, auquel cas la prochaine tâche prête est élue. L'ordre dans lequel les tâches sont élues suit une politique du "premier arrivé, premier servi" (ou First Come, First Served, FIFO).

Lorsqu'une tâche est ordonnancée, elle garde le processeur jusqu'à ce qu'elle le relâche volontairement ou qu'elle soit bloquée en attente d'une ressource. Les interruptions ne sont pas autorisées et ne peuvent l'être, sous peine de rendre non prédictibles les contraintes

temporelles. Les périphériques matériels doivent être interrogés tour à tour afin de déterminer si une opération est terminée et si d'éventuelles données sont disponibles pour traitement.

Ce type d'ordonnancement est très simple, mais toute modification est difficile, car elle requiert une réévaluation de la gestion temporelle.

Multitâche coopératif. Dans un algorithme multitâche coopératif, les tâches tournent jusqu'à ce qu'elles cèdent d'elles-mêmes le CPU pour qu'une autre tâche puisse tourner. Ceci était le cas dans d'anciennes versions de Windows (jusqu'à Windows 3.11) et Mac OS (jusqu'à Mac OS 9). Ce type d'algorithme est, comme le précédent, très simple, mais souffre de plusieurs inconvénients :

- Si l'une des tâches ne redonne pas la main à une autre, par exemple en cas d'erreur de programmation, le système entier peut s'arrêter.
- Le partage des ressources (temps CPU, mémoire, accès disque, etc.) peut être inefficace et les échéances temporelles peuvent ne pas être respectées si une tâche refuse de rendre le processeur en temps voulu.
- Le multitâche coopératif est une forme de couplage fort. Le couplage fort va à l'encontre d'un guide de design général en génie logiciel (voir la section 9, p. 195 à ce sujet).

Temps partagé, Round Robin ou tourniquet. Dans un algorithme en temps partagé, les tâches se voient allouées une certaine portion de temps, nommée *quantum* (un multiple du tick de l'horloge système) pendant laquelle elle tourne sur le CPU. À l'expiration du quantum de la tâche en cours, ou lorsque celle ci est bloquée, la prochaine tâche prête est élue et tourne à son tour pendant un quantum, et ainsi de suite.

L'algorithme en temps partagé est souvent vu comme une version très appauvrie du schéma temps partagé à la base des ordonnanceurs de Linux. Notons que le temps partagé n'est pas synonyme de partage équitable :

- Il n'y a pas de manipulation dynamique des priorités par le système.
- Les tâches ne partagent pas nécessairement le CPU de manière équitable. En particulier, si une tâche est bloquée (par exemple en attente d'une E/S), elle perd une partie de son quantum.

Multitâche préemptif avec priorités

*Une **priorité** assignée à une tâche est un entier mesurant l'importance à s'exécuter. L'approche la plus simple est d'assigner les priorités de manière statique à chaque tâche, à leur création. Lorsque c'est le cas, le **schéma d'ordonnancement** est dit **à priorités fixes** ; dans le cas contraire, il est dit **à priorités dynamiques**. À chaque priorité correspond une file d'attente stockant des structures représentant les tâches en attente d'exécution.*

L'algorithme d'ordonnancement sélectionne la première file d'attente non vide (en priorité décroissante) et sélectionne la première tâche dans cette file (en ordre FIFO). Cet algorithme multitâche préemptif avec priorités est parfois dénommé FIFO avec priorités.

Partionnement temporel L'ordonnancement à partitionnement temporel garantit que des threads ou des groupes de threads obtiennent un pourcentage déterminé du CPU. Ceci empêche une partie de système d'affamer les autres processus.

Ordonnancement par échéances Dans cet algorithme, disponible sur des OS spécialisés, l'ordonnancement est calculé dynamiquement pour que les applications satisfassent des échéances préalablement spécifiées.

4 L'ordonnanceur CFS de Linux

L'ordonnanceur par défaut sous Linux est, depuis la version 2.6.23 du noyau (sortie en octobre 2007), le CFS ou "Completely Fair Scheduler", l'ordonnanceur complètement équitable. Contrairement aux ordonnanceurs qui l'ont précédé, le CFS considère seulement le temps d'attente d'un processus, c.à.d. le temps qu'il a passé dans la file d'attente des tâches prêtes avant d'être exécuté. La tâche ayant attendu le plus longtemps est ordonnancée.

Le principe général de cet ordonnanceur est de fournir le maximum d'équité à chaque tâche en termes de puissance de calcul qui lui est fournie. Considérons une machine idéale capable de faire tourner un nombre arbitraire de tâches en parallélisme physique ; si N processus sont présents dans le système, alors chacun reçoit $1/N$ de la puissance de calcul totale disponible et toutes les tâches s'exécutent en parallélisme réel. Supposons qu'une tâche ait besoin de 10 minutes pour compléter son travail et que 5 tâches soient simultanément présentes ; sur une telle machine parfaite, chacune obtiendra 20% de la puissance de calcul et tournera donc 50 minutes au lieu de 10. Par contre, chacune des cinq tâches aura fini son exécution précisément après ce laps de temps et aucune n'aura jamais été inactive. Ceci est évidemment impossible à réaliser sur un matériel réel : si un système possède un seul CPU, seul un processus peut tourner à la fois.

Le multitâche est seulement obtenu en basculant d'une tâche à une autre à haute fréquence. Pour les utilisateurs finaux, ceci donne l'illusion d'une exécution parallèle, ce qui n'est pas le cas en réalité. Bien que la présence de plusieurs CPU (ou plusieurs cœurs) améliore la situation, cela ne règle pas le problème. Lorsque le multitâche est simulé par l'exécution d'un processus après l'autre, le processus en train de tourner sur le CPU est favorisé par rapport à ceux en attente, qui sont traités avec une certaine inéquité. Cette inéquité est directement proportionnelle au temps d'attente.

Chaque fois que l'ordonnanceur est appelé, il prend la tâche ayant le temps d'attente le plus important et lui donne le CPU. Si ceci arrive suffisamment souvent, l'inéquité sera maintenue à une valeur acceptable et sera uniformément distribuée entre les différentes tâches du système.

Toutes les tâches prêtes à être exécutées sont stockées dans un arbre rouge-noir, essentiellement par rapport à leur temps d'attente. Une tâche qui a le plus grand temps d'attente du CPU sera l'entrée la plus à gauche et sera la prochaine élue sur le processeur. Les tâches ayant attendu moins longtemps sont triées dans l'arbre de gauche à droite.

Les arbres rouge-noir sont des arbres binaires caractérisés par les propriétés suivantes :

- Chaque nœud est soit rouge, soit noir.
- Chaque feuille (nœud de bord de l'arbre) est noire.
- Si un nœud est rouge, chacun de ses enfants doit être noir. Il s'ensuit qu'il ne peut y avoir deux nœuds rouge consécutifs sur aucun chemin de la racine à une feuille, mais il peut y avoir un nombre quelconque de nœds noirs.
- Le nombre de nœuds noirs le long d'un chemin simple d'un nœud à une feuille est le même pour toutes feuilles.

Un des avantages des arbres rouge-noir est que toutes les opérations importantes (insertion, destruction et recherche d'élément) peuvent s'effectuer en $O(\log n)$ opérations, où n est le nombre d'éléments de l'arbre. Plusieurs arbres rouge-noir sont disponibles comme structures de données standard pour le noyau.

Ces nœuds sont indicés par le temps d'exécution processeur en nanosecondes. Un temps maximum d'exécution est également calculé pour chaque processus. Ce temps repose sur l'idée qu'un processeur idéal partagerait la puissance processeur de manière parfaitement équitable entre tous les processus. Le temps maximum d'exécution est le temps que le processus a attendu avant de s'exécuter, divisé par le nombre total de processus (c'est donc le temps pendant lequel un processus aurait pu s'exécuter sur une processeur idéal).

L'algorithme d'ordonnancement est comme suit. Lorsque l'ordonnanceur est appelé pour exécuter un processus, les opérations qu'il réalise sont :

- Le nœud le plus à gauche de l'arbre d'ordonnancement est choisi (c.à.d. celui qui s'est exécuté le moins longtemps) et est envoyé pour exécution.
- Si le processus termine son exécution, il est retiré du système et de l'arbre d'ordonnancement.
- Si le processus atteint son temps maximum d'exécution ou est stoppé (volontairement ou par une interruption), il est réinséré dans l'arbre d'ordonnancement selon son nouveau temps passé à s'exécuter.
- Le nouveau nœud le plus à gauche de l'arbre est alors choisi, répétant l'itération.

Si un processus passe beaucoup de temps à dormir, sa valeur de temps d'exécution est faible et il obtient automatiquement une augmentation de priorité. Donc, de telles tâches n'obtiennent pas moins de temps processeur que celles qui s'exécutent tout le temps.

Ceci est une implantation de l'algorithme classique nommé “weighted fair queuing”, originellement développé pour des réseaux à paquets.

5 Ordonnanceurs pour un contexte temps réel

5.1 Éléments sur l'ordonnancement statique/dynamique

Ordonnancement statique. Un ordonnancement statique se réfère plus précisément au fait que l'algorithme d'ordonnancement a une connaissance complète de l'ensemble des tâches et de leurs contraintes telles que les échéances temporelles, les temps de calcul, les contraintes de précédence et les temps d'activation futurs. La plupart de la théorie sur l'ordonnancement traite d'ordonnancement statique.

Un ordonnancement statique peut être réalisé hors ligne. Pour des tâches périodiques $\tau_1, \ldots, \tau_n$ avec des périodes $T_1, \ldots, T_n$, la table doit couvrir un intervalle temporel de

$$T_1 \vee T_2 \vee \ldots \vee T_n$$

le PPCM (plus petit commun multiple) des T_i ; is les T_i sont premiers entre eux, ce nombre peut être très élevé. Donc, lorsque c'est possible, choisir les T_i comme des petits multiples d'une valeur commune.

Les avantages sont que l'ordre d'exécution peut être déterminé hors ligne et la surcharge due à l'ordonnancement est très faible. Ce schéma est bien adapté à des processus purement cycliques.

Les désavantages sont le manque total de flexibilité, ce qui implique que les choix les plus conservatifs seront faits, impliquant une grande sous utilisation des ressources. Par ailleurs, il est très difficile de traiter les tâches sporadiques de haute priorité (comme celles dues à des événements imprévus).

Les suppositions liées à l'ordonnancement statique sont réalistes pour un grand nombre de systèmes temps réel. Ceci correspond typiquement à un système de contrôle technologique simple avec un ensemble de capteurs et d'actionneurs fixe, un environnement et des ressources de calcul bien défini. Dans ce type de système, l'algorithme d'ordonnancement statique produit un cadencement fixé une fois pour toutes.

Ordonnancement dynamique. Par contraste, un ordonnancement dynamique a une connaissance complète sur sur l'ensemble des tâches actives, mais de nouvelles arrivées peuvent se produire dans le futur, qui ne sont pas connues de l'algorithme au moment où il ordonnance l'ensemble actuel. Le cadencement change donc au cours du temps.

L'ordonnancement dynamique est a priori requis pour des systèmes complexes. Ceci étant, bien moins de résultats théoriques sont connus pour ce type d'ordonnancement que pour l'ordonnancement statique.

5.2 Multitâche préemptif avec priorités

Comme il a été vu en sous section 3.3.2, p. 176, cet algorithme d'ordonnancement à priorités fixes sélectionne la première file d'attente non vide (en priorité décroissante) et sélectionne la première tâche dans cette file (en ordre FIFO). Une fois qu'une tâche est élue par l'ordonnanceur, elle garde le processeur jusqu'à ce que l'un des événements suivant

survienne : elle est préemptée par une tâche de plus haute priorité, elle laisse le processeur volontairement ou elle doit attendre d'avoir accès à une ressource. L'utilisation de priorités implique une forme de préemption : l'ordonnanceur interrompt une tâche lorsqu'une tâche de plus haute priorité est prête à s'exécuter. Ici, c'est l'utilisateur qui fixe la priorité de la tâche au moment de sa création.

Cet algorithme est implanté sur la plupart des systèmes temps réel du commerce (RTOSs ou Real Time Operating Systems). Il n'est pas conçu pour être équitable, mais pour être déterministe.

5.3 Ordonnançabilité

Nous allons avoir besoin d'une définition pour les deux sous sections suivantes. L'utilisation CPU U d'une tâche périodique de période T et de temps d'exécution dans le pire cas C est :

$$U = \frac{C}{T}$$

Un ensemble de tâches périodiques est dite ***ordonnançable*** *par un algorithme d'ordonnancement si, en appliquant cet algorithme chaque tâche a son temps d'exécution réel inférieure à son échéance.*

Évidemment, si un ensemble de tâches a une utilisation CPU totale plus grande que 1, elle n'est pas ordonnançable.

5.4 Ordonnancement RM

L'un des algorithmes d'ordonnancement à priorités fixes le plus connu est dit *à taux monotone* ou *Rate Monotonic* (RM). Une tâche obtient une plus grande priorité si elle tourne plus fréquemment ; ce qui revient à dire que sa priorité est proportionnelle à l'inverse de sa période d'activation. Ceci favorise les tâches périodiques avec une haute fréquence d'exécution. L'ordonnanceur a besoin de connaître la période de chaque tâche qui doit s'exécuter. Ce schéma suppose que toutes les tâches sont périodiques. Elles sont également supposées indépendantes, de façon à pouvoir tester leur ordonnançabilité.

Il a été montré par Liu et Layland en 1973 que

Proposition VII.1. *L'algorithme d'ordonnancement à taux monotone (RM) est optimal, en ce sens que s'il existe un algorithme à priorité fixes capable de respecter toutes les échéances données, alors l'algorithme à taux monotone en est également capable.*

Ceci est du au fait qu'un ensemble de tâches ordonnançables, non ordonnancées par RM, peuvent être transformées en un ordonnancement RM par permutations, sans dépasser aucune échéance.

5.5 Ordonnancement à échéance proche (Earliest Deadline First)

Une approche plus élaborée est de disposer d'une assignation dynamique des priorités. Ces dernières sont mises à jour en ligne, selon divers paramètres (temps jusqu'à la prochaine échéance, temps d'exécution écoulé sur le processeur, ...). Dans des systèmes temps réel, les heuristiques pour changer ces priorités doivent rester simples, de façon à conserver la prédictibilité de l'ordonnanceur.

Un algorithme à priorités dynamiques classique est dit à *échéance proche* ou *Earliest Deadline First* (EDF) scheduling, présenté par Liu and Layland en 1973. Une tâche avec une échéance plus proche obtient une priorité plus élevée. L'ordonnanceur a besoin de connaître l'échéance et le temps d'exécution de chaque tâche. Nous avons également le résultat suivant d'optimalité, établit en 1974 par Dertouzos :

Proposition VII.2. *L'algorithme à échéance proche (EDF) est optimal parmi tous les algorithmes d'ordonnancement préemptifs dans le cas uni processeur.*

s'il existe un algorithme à priorité dynamique capable de respecter toutes les échéances données, alors l'algorithme à échéance proche (EDF) en est également capable. Le résultat vient du fait que tout autre ordonnancement peut être transformé en EDF par une permutation adéquate des tâches.

Ceci étant, cet algorithme est assez difficile à mettre en $\frac{1}{2}$uvre et est donc peu utilisé industriellement. De plus, il ne prévoit aucun compromis satisfaisant en cas de surcharge du système (taux d'utilisation supérieur à 100%) ; son utilisation peut donc s'avérer dangereuse dans les systèmes temps réel industriels.

6 How to meet deadlines : schedulability tests

An overall goal is here to obtain scheduling schemes which fulfill the deadlines. We also wish to ba able to verify the temporal coherence of a program. To exactly predict the execution time of a task is difficult, or even impossible in general. It is much easier to find the execution time interval. Most of the past and present research on real time systems focus on giving precise bounds for the Worst Case Execution Time (WCET) of a task. We shall here derive some bounds first in a very simplistic setting, and second in a more realistic one.

6.1 A simplistic model for fixed priority scheduling

6.1.1 Model of the (software) system

We will consider a program P that recieves a sensor event every T time unit. This program P executes in C time units in the worst case, and it must have finished its execution by D time units. In other words :

T is the minimal inter-arrival time (or minimal period)
C is the Worst Case Execution Time
D is the coherence delay (or deadline)
If

$$D < C$$

(Worst Case Execution Time is greater than the coherence delay), the system will not be able to meet the deadlines in all cases. Even if

$$T < D$$

the execution time mustr be less than T

$$C < T$$

if we don't want to miss any event. The more delicate case is the following :

$$C \leqslant D \leqslant T$$

where the coherence delay is less than or equal to the interarrival time.

A real time program in this model consists in :

– Periodic tasks (invoked upon events)
– independant
– which do not communicate

Remaque VII.1. This scheme is highly simplistic, but very simple for the mathematical analysis.

A real time system consists in :

– A unique processor
– Periodic event occurence (for example through timers)
– No event memorisation
– Each event sets a task in the ready state.

To each task τ_i, we associate a period T_i (an inter arrival time) and a worst case execution time C_i.

6.1.2 Simple method analysis : necessary conditions

The condition $\mathcal{C}_1^{\text{néc}}$ is the following :

$$\forall i, \quad C_i < T_i$$

in other words the execution time be less than the activation period (which is clearly necessary but not sufficient).

Condition $\mathcal{C}_2^{\text{néc}}$ states :

$$\forall i, \quad \sum_{j=1}^{n} \frac{C_j}{T_j} \leqslant 1$$

where C_j/T_j is the use percentage of the processor by the task τ_j. The sum of all these should then be less than 100%. This may not be sufficient if a task is preempted by another one with a higher priority.

The condition $\mathcal{C}_2^{\text{néc}}$ takes into account the computing time of higher priority tasks :

$$\forall i, \quad \sum_{j=1}^{i-1} \left(\frac{T_i}{T_j} C_j \right) \leqslant T_i - C_i$$

Assume T_i/T_j is an integer. The computations made at levels with a higher priority than i (from 1 to $i-1$) have to be finished within the time $T_i - C_i$. At each level j ($< i$), there will be T_i/T_j invocations in an interval of width T_i, each invocation requiring an execution time of C_j. For each level j, the necessary computing time is

$$\frac{T_i}{T_j} C_j$$

But, if $T_j > T_i$ $\mathcal{C}_3^{\text{néc}} = \mathcal{C}_1^{\text{néc}}$ which is not sufficient.

Remaque VII.2. If T_i/T_j is not an integer, $\lceil T_i/T_j \rceil$ is an overestimation, and $\lfloor T_i/T_j \rfloor$ is an overestimation. We can thus take

$$M_i = T_1 \vee T_2 \vee \ldots \vee T_i \qquad (\text{LCM de } T_1, \ldots, T_i)$$

The number of invocations at any level j within an interval of width M_i is exactly M_i/M_j, wherefrom the variant :

$$\forall i, \quad \sum_{j=1}^{i-1} \left(\frac{M_i/T_j}{M_i} C_j \right) \leqslant 1$$

This condition averages the needs in computation over each LCM period ; it is not sufficient since it is equivalent to $\mathcal{C}_2^{\text{néc}}$.

6.1.3 Simple method analysis : sufficient condition

Assume that the deadlines are the same as the periods. A sufficient conservative condition of schedulability is then $\mathcal{C}_5^{\text{néc}}$:

- The first deadline for every task must be met
- It will be the case if

$$n\big(2^{1/n} - 1\big) \geqslant \sum_{i=1}^{n} \frac{C_i}{T_i}$$

  For $n = 2$ this upper bound to the utilisation is equal to $82,84\%$. The limit for large n is $69,31\%$.

6.1.4 Simple method analysis : a necessary and sufficient condition

Let R_i be the maximum reponse time : the maximum time between the arrival of an invocation and the end of the corresponding computation. An implmentation will be schedulable if

$$\forall i, \quad R_i \leqslant D_i$$

the maximum response time is less or equal to the coherence delay. If τ_j, $j < i$ has a higher priority than τ_i, the maximum reponse time satisfies

$$R_i = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i}{T_j} \right\rceil C_j$$

For each level j ($< i$) with a higher priority than level i, there will be $\lceil R_i/T_j \rceil$ invocations within the time range R_i, each invocation needing C_j time units.

Since, due to the integer part, this equation is nonlinear, an iterative method is the simplest way. We consider the sequence R_i^n solution of

$$R_i^{n+1} = C_i + \sum_{j=1}^{i-1} \left\lceil \frac{R_i^n}{T_j} \right\rceil C_j, \qquad R_i^0 = C_i$$

if its limits exists, then it is R_i. This is a positive, strictly increasing sequence. Thus, either its diverges, or it is stationary. We then compute the successive values of R_i^n until we find $R_i^{m+1} = R_i^m$.

As an Exercise we can try to find the response time of the task τ_4, given the following static priority table

	Priority	Period	Execution time
τ_1	1	10	1
τ_2	2	12	2
τ_3	3	30	8
τ_4	4	600	20

We have

$$
\begin{aligned}
R_4^0 &= C_4 = 20 \\
R_4^1 &= C_4 + \left\lceil \frac{R_4^0}{T_1} \right\rceil C_1 + \left\lceil \frac{R_4^0}{T_2} \right\rceil C_2 + \left\lceil \frac{R_4^0}{T_3} \right\rceil C_3 \\
&= 20 + 2C_1 + \lceil 1,6666 \rceil C_2 + \lceil 0,6666 \rceil C_3 \\
&= 20 + 2 + 4 + 8 \\
&= 34 \\
R_4^2 &= C_4 + \left\lceil \frac{R_4^1}{10} \right\rceil C_1 + \left\lceil \frac{R_4^1}{12} \right\rceil C_2 + \left\lceil \frac{R_4^1}{30} \right\rceil C_3 \\
&= 20 + 4C_1 + 3C_2 + 2C_3 \\
&= 20 + 4 + 6 + 16 \\
&= 46
\end{aligned}
$$

Then,

$$
\begin{aligned}
R_4^3 &= C_4 + \left\lceil \frac{R_4^2}{10} \right\rceil C_1 + \left\lceil \frac{R_4^2}{12} \right\rceil C_2 + \left\lceil \frac{R_4^2}{30} \right\rceil C_3 \\
&= 20 + 5C_1 + 4C_2 + 2C_3 \\
&= 20 + 5 + 8 + 16 \\
&= 49 \\
R_4^4 &= C_4 + \left\lceil \frac{R_4^3}{10} \right\rceil C_1 + \left\lceil \frac{R_4^3}{12} \right\rceil C_2 + \left\lceil \frac{R_4^3}{30} \right\rceil C_3 \\
&= 20 + 5C_1 + 5C_2 + 2C_3 \\
&= 20 + 5 + 10 + 16 \\
&= 51
\end{aligned}
$$

$$
\begin{aligned}
R_4^5 &= C_4 + \left\lceil \frac{R_4^4}{10} \right\rceil C_1 + \left\lceil \frac{R_4^4}{12} \right\rceil C_2 + \left\lceil \frac{R_4^4}{30} \right\rceil C_3 \\
&= 20 + 6C_1 + 5C_2 + 2C_3 \\
&= 20 + 6 + 10 + 16 \\
&= 52 \\
R_4^6 &= C_4 + \left\lceil \frac{R_4^5}{10} \right\rceil C_1 + \left\lceil \frac{R_4^5}{12} \right\rceil C_2 + \left\lceil \frac{R_4^5}{30} \right\rceil C_3 \\
&= 20 + 6C_1 + 5C_2 + 2C_3 \\
&= 52
\end{aligned}
$$

Hence $R_4 = 52$.

6.2 dynamic priority schedulability

In the case of dynamic priority algorithms, such as the earliest deadline first (EDF) one, a set of n periodic (non interacting) tasks is schedulable if and only if

$$\sum_{i=1}^{n} U_i \leqslant 1$$

where U_i is the utilisation of task number i : $U_i = C_i/T_i$.

In this framework, tasks are often supposed to be independent, non-periodic and non-preemptive [**?**] in order to check the schedulability.

The independence assumption is one of the more restrictive, since obviously, several tasks must synchronize and/or communicate (e.g. a sensor gathering task must synchronize and communicate with a control law synthesis task).

Some extensions (with similar bounds) can be proven in the case where the tasks communicate with each other through synchronization mechanisms.

7 Programmation concurrente

7.1 Notion et problèmes associés

La programmation concurrente est une forme de programmation dans laquelle plusieurs calculs s'exécutent sur des intervalles de temps qui se chevauchent – de manière concurrente – au lieu d'une exécution séquentielle (un calcul ne commence qu'après la terminaison du précédent). On distingue trois types de concurrence :

- disjointe : les entités concurrentes ne communiquent et n'interagissent pas,
- compétitive : un ensemble d'entités concurrentes en compétition pour l'accès à certaines ressources partagées (par exemple le temps CPU, un port d'entrées/sorties, une zone mémoire),
- coopérative : un ensemble d'entités concurrentes qui coopèrent pour atteindre un objectif commun. Des échanges ont lieu entre les processus. La coopération est un élément primordial de la programmation concurrente.

Dans un programme séquentiel (non concurrent), l'ordre d'exécution reste le même d'une exécution à l'autre. Dans un programme concurrent, cet ordre dépend de la politique d'ordonnancement ; cette dernière étant, dans un système classique non déterministe (ou non prédictible), il y a donc indéterminisme de l'ordre d'exécution. Les problèmes liés à la programmation concurrente surviennent en concurrence compétitive et coopérative. L'ordre d'exécution n'étant pas fixe, l'accès à des données partagées peut mener à des incohérences. On utilise alors des mécanismes de synchronisation permettant de bloquer ou de débloquer des tâches (mutex, sémaphores ou moniteurs).

7.2 Risques liés à l'utilisation des threads

7.2.1 Sûreté des threads (ou "Thread safety") et atomicité

Ce type de problème peut survenir en cas de partage de données entre différentes threads sans synchronisation ou avec une synchronisation inadaptée. Par exemple, considérons deux threads, `A` et `B` qui accèdent à une variable unique, `staticVar`, pour l'incrémenter par le biais de la fonction `getNext()` :

Exemple de code VII.1

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

void *fn_thread(void *arg)
{
  int val;

  val = *(* int)arg;
  fprintf(stdout, "Thread no %d - valeur == %d\n",
                  pthread_self(), getNext(val));

  pthread_exit((void *) 0);
}

int initVal(int *val)
{
  *val = 0;
}

int getNext(int *val)
{
  return (*val)++;
}

int main (void)
{
  int         i, ret;
  void       *retour;
  pthread_t   thread;
  static int  staticVar;
```

```
  initVal(&staticVar);

  /* Lancement de la thread A */
  if ((ret = pthread_create(&thread, NULL, fn_thread,
                            (void *)(&staticVar))) != 0) {
      fprintf(stderr, "%s\n", strerror(ret));
      exit(1);
  }
  /* Lancement de la thread B */
  if ((ret = pthread_create(&thread, NULL, fn_thread,
                            (void *)(&staticVar))) != 0) {
      fprintf(stderr, "%s\n", strerror(ret));
      exit(1);
  }
  pthread_join(thread , &retour);
  if (retour != PTHREAD_CANCELED) {
      i = (int)retour;
      fprintf(stdout, "main : valeur lue = %d\n", i);
  }
  return(0);
}
```

Voici le même exemple en java :

Exemple de code VII.2

```
class incrementNonThreadSafe implements Runnable {
  private static int staticVar;
  private Thread     th;

  public incrementNonThreadSafe(int value){
      staticVar = value;
      th  = new Thread(this);
  }

  public void run() {
      System.out.println("Thread " + this + " valeur == " + getNext());
  }

  public int getNext()
  {
    return staticVar++;
  }

  public static void main(String args[])
```

```
    {
      incrementNonThreadSafe thA = new incrementNonThreadSafe(0);
      incrementNonThreadSafe thB = new incrementNonThreadSafe(0);

      thA.start();
      thB.start();

      thA.join();
      thB.join();
    }
  }
```

Le problème est ici que *l'incrémentation* `staticVar++` *n'est pas une opération atomique* (par exemple une opération s'exécutant en une seule instruction processeur), empêchant une thread d'être interrompue par une autre tant que l'opération n'est pas terminée.

En réalité, `staticVar++` réalise trois opérations : lecture de la valeur de `staticVar`, incrémentation, puis écriture de la nouvelle valeur dans `staticVar`.

Ainsi, la séquence temporelle suivante est susceptible d'arriver (même si cela n'est pas fréquent) :

```
   /*       OPERATION REALISEE            */    /* VALEUR de staticVar */
1  La thread A lit la valeur de staticVar              /* 0 */
2  La thread A est interrompue par la thread B         /* 0 */
3  La thread B lit la valeur de staticVar              /* 0 */
4  La thread B est interrompue par la thread A         /* 0 */
5  La thread A incrémente la valeur lue                /* 0 */
6  La thread A écrit la nouvelle valeur dans staticVar /* 1 */
7  La thread A renvoie la valeur de staticVar          /* 1 */
8  La thread A est interrompue par la thread B         /* 1 */
9  La thread B incrémente la valeur lue                /* 1 */
10 La thread B écrit la nouvelle valeur dans staticVar /* 2 */
11 La thread B renvoie la valeur de staticVar          /* 2 */
```

Où l'on voit clairement que les deux threads renvoient une valeur différente de `staticVar` qui a été incrémentée deux fois ! Cette situation, classique en programmation multi threads, est dite une *situation de compétition* (ou "race condition").

> *Une séquence d'instructions est dite* ***atomique*** *vis à vis de l'ordonnanceur si, lorsqu'une thread est en cours d'exécution de cette séquence, elle ne peut être interrompue par une autre thread. Une* ***section critique*** *est une section de code qui accède à une ou plusieurs ressources partagées et dont l'exécution doit être atomique.*

Les instructions assembleur peuvent être considérées comme atomiques. Dans le code du noyau, le fait de bloquer les interruptions (avec les fonctions `local_irq_disable()`

ou `local_irq_save()`) ou d'empêcher la préemption d'une thread par l'ordonnanceur (avec `preempt_disable()` permet de rendre une suite d'instructions atomique. Ceci n'est pourtant possible qu'au sein du noyau, les fonctions citées n'étant pas accessible depuis l'espace utilisateur.

7.3 Risques liés au manque de vivacité

Le manque de vivacité d'un ensemble de threads d'un programme survient lorsqu'au moins une de ces threads se trouve dans un état tel qu'elle ne peut plus progresser. L'un des exemple les plus triviaux est une boucle sans fin.

Les quatre cas classiques de manque de vivacité sont :

> *L'**étreinte fatale**, ou interblocage (ou "**deadlock**"). Ceci survient lorsque plusieurs threads s'attendent mutuellement de manière indéfinie (par exemple pour des ressources de synchronisation).*
> *La **famine**. Celle ci survient lorsqu'une thread se voit constamment refuser l'accès à des ressources dont il a besoin pour continuer son traitement.*
> *Les **signaux manqués**. Une thread attend une condition qui est déjà vraie, mais l'on oublie d'effectuer le test associé à la condition avant de se mettre en attente. La thread attend alors une notification d'un événement ayant déjà eu lieu.*
> *L'**interblocage actif**, aussi dénommé famine (ou "**livelock**"). Un tel thread n'est pas bloqué, mais il ne peut continuer son traitement car il essaie d'effectuer indéfiniment une opération qui échoue toujours.*

Ces problèmes peuvent être très difficiles à détecter, car ils dépendent des déroulements temporels respectifs des différentes threads (comme illustré dans l'exemple précédent). Ils nécessitent une bonne conception dès le départ du projet logiciel pour ne pas survenir.

En essence, un code sûr du point de vue de la gestion des threads (ou "thread safe code") est un code permettant l'accès à l'*état partagé et modifiable* de ces threads. L'état est l'ensemble des valeurs prises par les variables associées à ces threads (valeurs des variables d'instance ou de classe dans le cas d'objets). Cet état est partagé (il peut être accédé par plusieurs threads) et il est modifiable (sa valeur peut varier au cours du temps). Les différentes threads doivent alors coordonner l'accès à cet état par des mécanismes de synchronisation (de type sémaphores ou mutex).

7.4 Réentrance

> *Une fonction est dite **réentrante** si son effet, lorsqu'elle est appelée par deux threads ou plus, est garanti d'être identique à celui obtenu si l'ordre d'exécution est séquentiel, même si les exécutions sont entremêlées.*

Une fonction qui accède uniquement à des variables locales est réentrante. Une fonction qui manipule des variables globales et statiques peut ne pas être réentrante. Dans ce cas, il peut y avoir un conflit d'accès à des ressources partagées.

Différentes fonctions de la bibliothèque C sont non réentrantes. Par exemple, `malloc()` et `free()` maintiennent une liste de chaînée des blocs mémoire libérés disponible pour une réallocation dans le tas. Si un appel à `malloc()` dans le programme principal est interrompu par un gestionnaire de signal qui appelle également `malloc()`, cette liste chaînée peut se trouver corrompue.

D'autres fonctions sont non réentrantes parce qu'elles renvoient de l'information en utilisant de la mémoire allouée statiquement (par exemple `crypt()`, `getpwnam()`, `gethostbyname()`, `getservbyname()`). Si un gestionnaire de signal utilise l'une de ces fonctions, il va écraser l'information renvoyée par un appel précédent à la même fonction depuis le programme principal.

Certaines fonctions peuvent être non réentrantes parce qu'elles utilisent des structures de données statiques pour leur fonctionnement interne. L'exemple le plus courant est celui des fonctions de la librairie `stdio` (c.à.d. `printf()`, `scanf()`, etc.) qui mettent à jour des structures de données internes pour leurs entrées/sorties tamponnées. Il est donc possible de voir des affichages étranges, des plantages ou des données corrompues si un gestionnaire de programme interrompt le programme principal au milieu d'un `printf()` ou d'une autre fonction de la bibliothèque `stdio`.

7.5 Modèle producteur/consommateur

Le problème ou *modèle du producteur/consommateur* est un exemple classique de synchronisation multi-tâches. Il décrit deux tâches, le producteur et le consommateur qui partagent un tampon mémoire de taille fixe. Le producteur génère des données, les place dans le tampon et recommence. Pendant ce temps là, le consommateur enlève des données du tampon et les traite. Le problème qui se pose est d'être sûr que le producteur n'ajoute pas de données alors que le tampon est plein ou que le consommateur ne consomme de données alors que le tampon est vide.

Une solution de *politique bloquante* est la suivante. Le producteur est bloqué dès que le tampon est plein. La prochaine fois que le consommateur enlève un élément du tampon, il le signale au producteur, qui commence à remplir le tampon à nouveau. De la même manière, le consommateur est bloqué dès que le tampon est vide. La prochaine fois que le producteur place des données dans le tampon, il le signale au consommateur. Cette solution peut être implantée en utilisant des sémaphores ou des mutex et variables de condition. Une solution inadéquate peut générer une étreinte fatale lorsque les deux tâches ont besoin d'être débloquées.

Une autre *politique non bloquante* est, lorsque le tampon est plein, que le producteur écrase les données les plus anciennes. Lorsque le tampon est vide, le consommateur délivre la dernière donnée qu'il a reçu.

Ce type de problème est typique de systèmes de contrôle. Considérons un système avec quatre tâches : τ_{se} de récupération des données capteur (sur la carte capteur) et de mise en tampon mémoire B_{se} ; τ_{sp} qui récupère les données de B_s, effectue des opérations de traitement du signal dessus (par exemple mise à l'échelle des données et filtrage) et les place dans un tampon B_{sp} ; τ_{ct} qui récupère les données de B_{sp}, calcule une loi de commande et en place le résultat dans un tampon B_{ct} ; τ_{ac} qui récupère les données de B_{ct} et les envoie sur la carte actionneur. Les couples (τ_{se}, τ_{sp}) de récupération de données, (τ_{sp}, τ_{ct}) de traitement du signal et (τ_{ct}, τ_{ac}) de calcul de commande sont tous des modèles producteur/consommateur.

7.6 Inversion de priorité

L'utilisation de verrous peut bloquer certaines tâches alors que cretaines tâches de plus basse priorité s'exécutent. Considérons trois tâches, de priorités décroissantes `hte` de priorité `H`, `moy` de priorité `M` et `bas` de priorité `B`, telles que `L` < `M` < `H`. La Figure VII.3 décrit un cas typique d'exécution avec préemption due à la priorité.

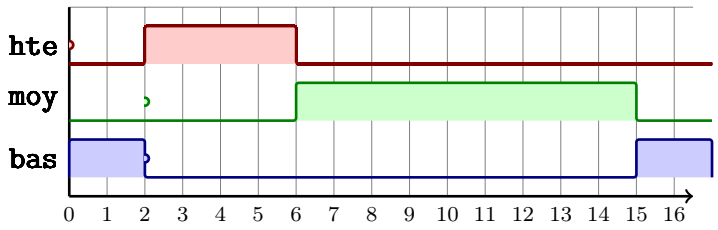

FIGURE VII.3 – Cas de préemption sans partage de ressource.

Supposons maintenant que la tâche `hte` soit bloquée, en attente d'un verrou détenu par la tâche `bas`. Puis que la tâche `bas` soit préemptée par la tâche `moy`, sachant que cette dernière n'a rien à voir avec la section critique (c.à.d. avec le verrou). Donc, la tâche `hte` est bloquée, alors qu'une tâche de plus basse priorité `moy` s'exécute (cf. Figure VII.4). Deux techniques sont habituellement employées pour résoudre les inversions de priorité :

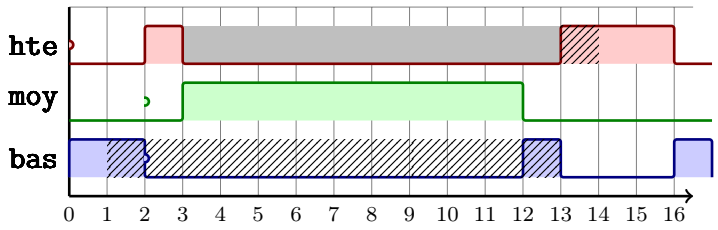

FIGURE VII.4 – Cas d'inversion de priorité. Les hachures désignent une détention de verrou et le gris plein désigne une attente bloquante de verrou.

- ***L'héritage de priorité***. Dans ce schéma, une tâche de basse priorité qui détient un verrou demandé par une tâche de plus haute priorité acquiert temporairement la priorité de la tâche en attente. Donc, la tâche `bas` hérite temporairement de la priorité `H` ; elle ne peut donc être préemptée par la tâche `moy`.
- ***Le Plafonnement de priorité***. Chaque tâche utilisant un verrou voit sa priorité élevée à une valeur préconfigurée (correspondant à la plus haute priorité d'une tâche pouvant détenir ce verrou).

Le plafonnement de priorité est plus basique que l'héritage de priorité, mais il nécessite moins de surcharge de calcul. Une remarque générale est que l'inversion de priorité est presque toujours le résultat d'une mauvaise conception et le programmeur devrait être au minimum averti.

8 Guides de bonne conception

8.1 Buts du génie logiciel

Les quatres buts généraux du génie logiciel (tels que définis dans l'article classique de D.T. Ross, J.B. Goodenough and C.A. Irvine [**?**]) sont les suivants :

- *Intelligibilité*. Le plus crucial des buts. Pour qu'un logiciel soit compréhensible, il doit être un bon modèle de notre vue du monde réel.
- *Fiabilité*. Il s'agit d'un but crucial pour les systèmes industriels. Il devient une nécessité impérieuse lorsque des vies humaines sont en jeu.
- *Modifiabilité*. Un logiciel peut être modifié à cause d'un changement de spécifications ou après corrections d'un bogue. Ces changements ne doivent pas ajouter de complexité supplémentaire.
- *Efficacité*. Les ressources temporelles et spatiales doivent être utilisées de manière optimale, particulièrement pour des systèmes embarqués et temps réel. Très souvent, les programmeurs se focalisent bien trop tôt sur l'efficacité, sans se préoccuper d'autres aspects.

En plus des buts précédents, différents auteurs (voir par exemple [**?**], [**?**]) ont mis en exergue le terme de *flexibilité*, qui comprend ce qui suit :

- *Modularité*, ou décomposition du logiciel en de plus petites entités ; ceci améliore la modifiabilité et l'intelligibilité du code.
- *Configurabilité*, ou comment modifier et adapter le logiciel d'une manière pré-déterminée et selon trois directions : d'*architecture* (choix des connexions entre composants), de *communication* (choix des primitives de communication entre composants), et *fonctionnelle* (choix de la stratégie de réalisation des fonctions demandées).
- *Portabilité*, le système logiciel doit être indépendant du matériel et du système d'exploitation ; des couches d'abstraction doivent être fournies.

9 Guides de design

9.1 Guides généraux

Deux guides généraux qui sont souvent considérés en génie logiciel appliqué sont les suivants :

- faible couplage. Un composant ne devrait pas compter sur la connaissance des détails internes d'autres composants.
- forte cohésion. Tout composant devrait avoir une définition et un comportement bien définis et bien focalisés.

Ces guides ont été découverts par Constantine et Yourdon [?] et systématisés au travers du patron de conception de faible couplage/forte cohésion dans l'ensemble des patrons de conception logiciels de responsabilité générale (General Responsibility Assignment Software Patterns or GRASP) [?], [?].

9.2 Guides détaillés

Les guides généraux peuvent se préciser comme suit (cf. [?], [?]).

Les guides permettant de faire décroître le couplage entre composants incluent ceux qui suivent.

- *Médiateurs.*
  Entourer les sections critiques dans un *médiateur*. Une sections critique (ou action atomique) est un ensemble d'instructions ou d'actions ininterruptible. En général, les sections critiques mettent en jeu plusieurs threads qui sont en compétition pour le(s) processeur(s). Le plongement dans un composant médiateur réduit le couplage entre ces tâches.
- *minimisation de couplage entre méthodes.*
  Toujours essayer de minimiser le couplage entre méthodes : ne prendre comme entrée que les données qui sont nécessaires à la méthode ; ne renvoyer comme sorties que les données produites par la méthode.
- *Hiérarchie.*
  Utiliser des hiérarchies appropriées, ce qui réduira la complexité et augmentera la modifiabilité. Cela peut rendre le système plus lent, mais bien moins complexe et donc plus compréhensible, sûr et modifiable.
- *Interface.*
  Découpler l'interface de l'implantation. Cette pratique relativement classique au sein des systèmes logiciels orientés objet permet à l'implantation de la classe étudiée d'être modifiée tout en laissant inchangées les classes qui y accèdent (via son interface).
- *Directive de champ.*
  Ne pas assigner plusieurs attributs à un même champ. Au lieu de cela, utiliser un champ séparé pour représenter chaque attribut d'une classe.

– *Le corollaire constant.*
Préférer des constantes à des valeurs codées en dur, d'autant plus si ces constantes doivent être utilisées en plusieurs endroits.

Des guides permettant d'accroître la cohésion des composants incluent ce qui suit.

– *Focalisation du nommage et du service.*
Un composant doit avoir un but bien focalisé et bien défini ; il en va de même pour les méthodes, chacune devant être focalisée sur un unique tâche conceptuelle. Cela permet, en théorie, un nommage qui ne nécessite aucune documentation. Plus précisément, il vaut nettement mieux avoir quarte méthodes exécutant quatre services bien définis qu'une seule méthode réalisant les quatre.

– *Transmission de données.*
Éviter de passer des données de contrôle (c.à.d. des données qui décident comment traiter les données) aux méthodes.

– *Éviter l'explosion de méthodes.*
Trouver un compromis entre maximiser la cohésion des méthodes (qui accroît le nombre de méthodes dans la classe) et maintenir le nombre de méthodes à une valeur raisonnable.

D'autres guides incluent ce qui suit :

– *Patrons de conception logiciels.*
Identifier le *patron de conception logiciel* qui correspond au problème considéré.

– *Séparation des comportements synchrone et asynchrone.*
Dans un service synchrone, le serveur fait attendre le client jusqu'à ce que le service ait été réalisé. Dans un service asynchrone, les client effectuent une requête qui est mise en file d'attente par le serveur. Le client n'est pas bloqué et se trouve rappelé par le serveur une fois que le requête a été traitée. Le comportement synchrone est typique d'un cycle bas niveau tel qu'une lecture d'encodeur, un canal de conversion analogique-numérique, une écriture d'une valeur PWM (Pulse Width Modulation) pour un moteur. Le comportement asynchrone est typique de tâches haut niveau, qui peuvent être itératives, telles qu'un capteur logiciel intelligent qui renvoie la position et l'orientation d'un robot terrestre ou aérien, ou bien être sporadiques, telles que le re-calcul d'une trajectoire de référence dans le cas d'un évitement d'obstacle.

– *Cohérence de l'état interne.*
Les composant de type serveur devraient toujours revenir à un seul état à partir duquel il est prêt à traiter la prochaine requête d'un client.

– *Accès à un état en Push/pull.*
L'état d'objets encapsulés devraient être accessibles en push ou en pull. Pour des objets décrivant une partie d'un système physique, comme une pièce mécanique, un actionneur, un capteur, un paramètre d'un modèle dynamique, il est nécessaire de fournir des accesseurs, c.à.d. des méthodes permettant d'avoir accès à l'état interne de ces objets. Ceci est utilie à la fois pour d'autres objets et pour l'utilisateur. Deux

modes devraient être fournis : pull synchrone et pull asynchrone ; ou bien les composant envoient leurs données à un broker périodiquement (de manière synchrone), ou bien le broker effectue un appel à un composant lorsqu'il en a besoin (de manière asynchrone).

– *Réentrance de thread.*
Les concepteurs devraient toujours supposer que plusieurs instances de leurs composants (chacun au sein d'une thread) soient actifs en même temps. La réentrance de thread signifie qu'un composant ou objet devrait se comporter correctement lorsque plusieurs instances s'exécutent de manière concurrente (voir Section 7.4, p. 191).

10 Outils de conception

10.1 Patrons logiciels

10.1.1 Notion de patron logiciel

*Un **patron logiciel** est une solution éprouvée et constructive à un problème classique dans un contexte bien défini [?], [?]. L'accent n'est pas mis sur le type d'outil utilisé, mais sur une conception saine.*

Un patron logiciel a deux buts principaux :

– Faciliter la communication entre concepteurs
– Décrire la vision architecturale d'une conception, plus que son implantation.

Il s'agit donc de solutions bien éprouvées à des problèmes bien connus dans un domaine bien défini. Puisque ces solutions correspondent à des problèmes fréquemment rencontrés, cela accroît la modifiabilité ; cela accroît également l'intelligibilité et la fiabilité parce que ces solutions sont éprouvées.

10.1.2 Description d'un patron logiciel

La description d'un patron logiciel met en jeu au moins les trois éléments suivants :

– Le contexte, décrivant la situation dans laquelle la conception s'effectue.
– Le problème, décrivant le problème à résoudre.
– La solution, fournissant une *solution éprouvée*, pour lequel plusieurs implantations existent.

10.1.3 Niveaux de patron logiciels

Trois niveaux de patrons logiciels ont été élaborés :

– Des patrons architecturaux, décrivant la subdivision d'un système logiciel en composants et spécifiant leurs interactions. Des exemples de tels patrons sont le tableau noir et le patron modèle/vue/contrôleur (utilisé dans les interfaces graphiques).

- Des patrons de conception, décrivant l'organisation interne des différents sous-systèmes d'une architecture. Les patrons d'itérateur et de stratégie en sont des exemples
- Des idiomes de language, patrons spécifiques à un langage de programmation pour implanter des aspects particuliers des composants. Par exemple les pointeurs intelligents (smart pointers) ou le verouillage avec portée (scoped locking) en C++.

10.2 Quelques patrons utiles en temps réel

Ces patrons, également utilisés dans un cadre de programmation concurrente, sont utiles pour une conception temps réel.

- Le médiateur, qui permet d'interconnecter deux composants qui doivent coopérer sans introduire de couplage. Ce patron est utilisé lorsque des composants doivent coopérer.
- Le Moniteur, également nommé objet passif immunisé aux threads. Ce patron assure que seule une méthode d'un objet peut être exécutée en même temps. Il sérialise donc les accès à l'objet. Le concept originel est dû à Hoare et se nomme parfois un verrou mutex (pour "MUTal EXclusion") dans un contexte temps réel.
- Le producteur/consommateur. Au travers de ce patron, un producteur de données peut continuer à produire des données aussi indépendamment que possible du consommateur. Il met en jeu des objets ou composants munis de threads et des tampons mémoire avec des accès synchronisés à ces derniers. Au travers de l'accès en mode pull, le consommateur signale au producteur de remplir à nouveau le tampon dès qu'il devient presque vide ; par accès en mode push, le producteur signale au consommateur de commencer à travailler lorsque le tampon est presque plein.
- Les machines à état fini (Hiérarchique) (FSM), maintenant la complexité d'une machine d'état à une valeur raisonnable et évolutive. La plupart des situations de vie à haut niveau sémantique se traduisent en des machines d'état.
- La Façade d'emballage masquant les différences entre diverses implantations de fonctionnalités similaires, ou fournissant une interface orientée objet à du code écrit dans un langage non orienté objet. Par exemple des appels de communication entre tâches sur différents systèmes d'exploitation.
- Le Configurateur, découplant la configuration des services de leur exécution.

VIII – Le système Xenomai

1 Introduction

Xenomai est un système temps réel hybride sous Linux, en ce sens qu'il fonctionne au sein d'un système Linux hôte ; il y a donc deux ordonnanceurs : celui du système hôte et celui de Xenomai. L'ordonnanceur de Xenomai préempte le système Linux hôte. Pour installer Xenomai sur un système linux, on récupère d'abord un noyau standard sur `kernel.org`, que l'on patche avec le patch Xenomai. Xenomai est livré avec Adeos, un système de répartition des interruptions hardware qui agit comme une couche de virtualisation et d'abstraction du matériel. Adeos se présente comme un nanokernel qui capture les événements matériels et les route en priorité à Xenomai ; s'ils ne concernent pas Xenomai, alors seulement, ils sont envoyés au système Linux hôte.

Une originalité de Xenomai est de proposer une interface de programmation dite native fort complète et des "skins" ou interface d'emballage d'autres systèmes ou normes. On trouve ainsi des skins pour VxWorks, psOS, la norme Posix,

2 API native

Cette interface de programmation est constituée des catégories suivantes :

– Gestion de tâches et ordonnancement

- Serices de gestion temporelle, incluant la gestion des timers et des alarmes de type watchdog.
- Gestion concurrente. Cette catégorie regroupe les mécanismes suivants : sémaphores à compte, mutex, variables de condition, événements.
- Messages et communication. Échange de données entre tâches temps réel ou entre tâche temps réel et non temps réel : passage de message synchrone, queues de messages, mémoire partagée, tubes de messages
- Gestion des entrées/sorties.
- Support de registres.

3 Affichage

Les affichages se font à l'aide des fonctions suivantes :

```
int rt_printf(const char *format, ...);
int rt_vprintf(const char *format, va_list args);

int rt_fprintf(FILE *stream, const char *format, ...);
int rt_vfprintf(FILE *stream, const char *format, va_list args);

int rt_puts(const char *ch);
```

Les messages à afficher sont stockés dans des tampons circulaires spécifiques à chaque tâche. Une thread non temps réel est chargée d'effectuer l'affichage du tampon périodiquement. La valeur de la période d'affichage est par défaut de 100ms et peut être changée en modifiant la variable d'environnement `RT_PRINT_PERIOD`.

Le démarrage de la thread d'affichage ainsi que l'allocation des tampons de stockage se fait par :

```
int rt_print_init(size_t buffer_size, const char *name);
```

fonction à laquelle on donne en argument la taille du tampon à allouer ainsi que son nom (un `malloc()` est effectué dans la fonction `rt_print_init()`). Une initialisation par défaut est fournie par

```
void rt_print_auto_init(int enable);
```

avec une taille de tampon égale à 16Ko. La fonction :

```
void rt_print_cleanup(void);
```

libère la zone mémoire allouée pour le tampon circulaire et tue la thread d'affichage.

4 Gestion de tâches

4.1 Aperçu des fonctions disponibles

Les fonctions suivantes sont diponibles :

`rt_task_create()`	Créer une nouvelle tâche temps réel.
`rt_task_start()`	Démarrer une tâche temps réel.
`rt_task_suspend()`	Suspendre une tâche temps réel.
`rt_task_resume()`	Continuer l'exécution d'une tâche temps réel.
`rt_task_delete()`	Détruire une tâche temps réel.
`rt_task_yield()`	Donner volontairement la main.
`rt_task_set_periodic()`	Rendre une tâche temps réel périodique.
`rt_task_wait_period()`	Attendre la prochaine période d'activation périodique.
`rt_task_set_priority()`	Changer la priorité de base d'une tâche temps réel.
`rt_task_sleep()`	Retarder la tâche appelante (relatif).
`rt_task_sleep_until()`	Retarder la tâche appelante (absolu).
`rt_task_unblock()`	Débloquer une tâche temps réel.
`rt_task_inquire()`	Se renseigner sur une tâche temps réel.
`rt_task_add_hook()`	Installer un crochet sur une tâche.
`rt_task_remove_hook()`	Enlever le crochet d'une tâche.
`rt_task_catch()`	Installer un gestionnaire de signaux.
`rt_task_notify()`	Envoyer un signal à une tâche.
`rt_task_set_mode()`	Changer le mode d'une tâche.
`rt_task_self()`	Obtenir le descripteur de la tâche appelante.
`rt_task_slice()`	Fixer le quantum d'une tâche (lorsqu'ordonnancée en round robin).
`rt_task_send()`	Envoyer un message à une tâche.
`rt_task_receive()`	Recevoir un message d'une tâche.
`rt_task_reply()`	Réponde à une tâche.
`rt_task_spawn()`	Créer et démarrer une nouvelle tâche temps réel.
`rt_task_shadow()`	Transformer la tâche Linux appelante en tâche native Xenomai.
`rt_task_bind()`	Lier à une tâche temps réel.
`rt_task_unbind()`	Délier une tâche temps réel.
`rt_task_join()`	Attendre la termination d'une tâche temps réel.
`rt_task_same()`	Comparer deux descripteurs de tâche.

4.2 Création, destruction, attente

4.2.1 Création

Une tâche temps réel est crée par `rt_task_create()`, puis démarrée par `rt_task_start()`, ou bien crée et démarrée immédiatement par `rt_task_spawn()`.

`int rt_task_create(`	Créer une nouvelle tâche temps réel.
`RT_TASK *task,`	Descripteur de tâche
`const char *name,`	Nom de la tâche dans `/proc/xenomai/sched`
`int stksize,`	Taille de la pile. Si nul, la taille est de 1024 `int`
`int prio,`	Priorité de la tâche (de 1 à 99)
`int mode)`	Mode de la tâche (cf. ci-dessous).

Le `mode` est un OU binaire entre les constantes suivantes :

`T_CPU(no)`	La tâche s'exécutera sur le CPU indiqué, le `no` doit être strictement inférieur à `RTHAL_NR_CPUS`.
`T_FPU`	La tâche utilisera les ressources de la FPU (floating point unit). Attribut automatiquement sélectionné pour les tâches crées dans l'espace utilisateur.
`T_JOINABLE`	Il est possible d'attendre la fin de l'exécution de cette tâche via `rt_task_join(()`
`T_SUSP`	La tâche est créé dans un état suspendu. Il faut l'activer avec `rt_task_resume()`
`T_WARNSW`	Si la tâche passe du mode primaire au mode secondaire, il recevra un signal `SIGXCPU`. Ceci tue par défaut la tâche, à moins que le signal ne soit intercepté.

La syntaxe de `rt_task_start()` est la suivante :

`int rt_task_start(`	Démarrer une tâche temps réel.
`RT_TASK *task,`	Descripteur de tâche
`void(*fun)(void *arg),`	Fonction exécutée par la tâche.
`void *arg)`	Argument que `fun()` reçoit à son démarrage.

Afin d'éviter des défauts de page (accès à une zone mémoire non encore disponible), ce qui entraine un passage de la tâche en mode secondaire, il est nécessaire de verrouiller les pages utilisées en mémoire physique. Ceci est réalisé par l'appel :

```
mlockall(MCL_CURRENT | MCL_FUTURE);
```

Voici un exemple simple de création d'une tâche

Exemple de code VIII.1

```
#include <native/task.h>

#define TASK_PRIO 99              /* plus haute priorite TR    */
#define TASK_MODE T_FPU|T_CPU(0) /* Utilise la FPU, sur CPU 0 */
#define TASK_STKSZ 4096           /* Taille de pile (en octets) */

RT_TASK task_desc;
```

```
void task_body(void *arg)
{
  for (;;) {
    /* ... "arg" devrait etre NULL ... */
  }
}

int init_module (void)
{
  int err;
  /* ... */
  mlockall(MCL_CURRENT | MCL_FUTURE);
  err = rt_task_create(&task_desc,
                       "MyTaskName",
                       TASK_STKSZ,
                       TASK_PRIO,
                       TASK_MODE);
  if (err != 0)
      rt_task_start(&task_desc, &task_body, NULL);
  /* ... */
}

void cleanup_module (void)
{
  rt_task_delete(&task_desc);
}
```

4.2.2 Attente et endormissement d'une tâche

La fonction `rt_task_join()` :

```
int rt_task_join(RT_TASK *task);
```

bloque la tâche appelante jusqu'à la terminaison de la tâche d'identifiant `task`. La fonction `rt_task_sleep()` :

```
rt_task_sleep(RTIME duree);
```

permet d'endormir la tâche appelante pendant une durée spécifiée en nanosecondes.

4.2.3 Création et démarrage immédiat d'une tâche

La fonction `rt_task_spawn()` crée une tâche et la démarre immédiatement.

`static int rt_task_spawn(`	Créer et démarrer une nouvelle tâche temps réel.
`RT_TASK *task,`	Descripteur de tâche
`const char *name,`	Nom de la tâche dans `/proc/xenomai/sched`
`int stksize,`	Taille de la pile. Si nul, la taille est de 1024 `int`
`int prio,`	Priorité de la tâche (de 1 à 99)
`int mode`	Mode de la tâche.
`void(*fun)(void *arg),`	Fonction exécutée par la tâche.
`void *arg)`	Argument que `fun()` reçoit à son démarrage.

Voici un exemple d'utilisation de `rt_task_spawn()` :

Exemple de code VIII.2

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/task.h>

void fonction_hello_world (void * unused)
{
    while (1) {
        rt_printf("Hello from Xenomai Realtime Space\n");
        rt_task_sleep(1000000000LL); // 1 milliard ns = 1 s.
    }
}


int main(void)
{
    int err;
    RT_TASK task;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_task_spawn(&task, "Hello_01",
                             0, 99, T_JOINABLE,
                             fonction_hello_world, NULL)) != 0)
    {
        fprintf(stderr, "rt_task_spawn: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }
    rt_task_join(& task);
```

```
    return 0;
}
```

4.2.4 Compilation

Pour la compilation, on pourra se servir d'un `makefile` analogue à celui ci, supposant que le code ci-dessus est stocké dans `exemple_rt_spawn.c` :

Exemple de code VIII.3

```
XENOCONF= /usr/xenomai/bin/xeno-config
CC =     $(shell $(XENOCONF) --cc)
MY_CFLAGS= -g
CFLAGS=  $(shell $(XENOCONF) --skin=native --cflags)  $(MY_CFLAGS)
LDFLAGS= $(shell $(XENOCONF) --skin=native --ldflags) $(MY_LDFLAGS)
LIBDIR=  $(shell  $(XENOCONF) --skin=native --libdir) $(MY_LIBDIR)
LDFLAGS+= -lnative
LDLIBS=   -lnative -lxenomai

all:: exemple_rt_spawn

clean::
    rm -f exemple_rt_spawn *.o *~
```

Le script `xeno-config` permet d'obtenir les paramètres de compilation adéquats à transmettre à `gcc`. Une fois la compilationeffectuée par `make`, il faut mettre à jour la variable `LD_LIBRARY_PATH` afin que le processus associé à la commande `exemple_rt_spawn` puisse trouver les bibliothèques adéquates :

```
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/usr/xenomai/lib
```

4.3 Destruction d'une tâche

Pour détruire une tâche, on utilise

```
int rt_task_delete(RT_TASK *task)
```

4.4 Fonctions de suspension et transformation

4.4.1 Transformation d'une tâche Linux en tâche xenomai

La fonction `int rt_task_shadow(RT_TASK *task, const char *name, int prio, int mode)` permet de transformer la tâche Linux appelante en tâche native Xenomai. En voici un exemple :

Exemple de code VIII.4

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/mman.h>

#include <rtdk.h>
#include <native/task.h>

int main(void)
{
    int err;
    RT_TASK task;

    mlockall(MCL_CURRENT|MCL_FUTURE);
    rt_print_auto_init(1);

    if ((err = rt_task_shadow(& task, "Hello World 02",
                              99, T_JOINABLE)) != 0) {
        fprintf(stderr, "rt_task_shadow: %s\n", strerror(-err));
        exit(EXIT_FAILURE);
    }
    while (1) {
        rt_printf("Hello World 02\n");
        rt_task_sleep(1000000000LL); // 1 sec.
    }
    return 0;
}
```

4.4.2 Suspension, redémarrage d'une tâche

Pour suspendre (resp. redémarrer) une tâche, on utilise

```
int rt_task_suspend(RT_TASK *task);
```

et

```
int rt_task_resume(RT_TASK *task);
```

4.4.3 Rendre une tâche temps réel périodique

Pour rendre une tâche temps réel périodique, on utilise la fonction `rt_task_set_periodic()`, dont la syntaxe est la suivante :

`int rt_task_set_periodic(`	Rendre une tâche temps réel périodique.
`RT_TASK *task,`	Identificateur de la tâche. Si `task` est égal à `NULL`, la tâche appelante est rendue périodique.
`RTIME idate,`	date (en basolu) de la première activation, exprimée en ticks d'horloge (en nanosecondes) . Si `idate` est égal à `TM_NOW`, la tâche est démarrée immédiatement.
`RTIME period)`	Période de la tâche, en ticks d'horloge (en nanosecondes). Si `period` est égal à `TM_INFINITE`, la tâche est arrêtée.

La fonction renvoie 0 en cas de succès et, en cas d'erreur :

`EINVAL`	si `task` n'est pas un descripteur de tâche, ou si la période est différente de `TM_INFINITE` mais inférieure à la valeur de latence d'ordonnancement, figurant dans `/proc/xenomai/latency`.
`EIDRM`	si `task` est un descripteur de tâche détruite.
`ETIMEDOUT`	si `idate` est différente de `TM_INFINITE` et représente une date passée.
`EWOULDBLOCK`	si le timer système n'est pas actif.
`EPERM`	si `task` est égal à `NULL` mais l'appel n'est pas réalisé d'un contexte de tâche.

5 Bref descriptif des autres fonctions de gestion de tâche

`int rt_task_yield(void)`
donner volontairement la main.

`int rt_task_wait_period(unsigned long *overruns_r)`
attendre la prochaine période d'activation périodique.

`int rt_task_set_priority(RT_TASK *task, int prio)`
changer la priorité de base d'une tâche temps réel.

`int rt_task_sleep(RTIME delay)`
retarder la tâche appelante (relatif).

`int rt_task_sleep_until(RTIME date)`
retarder la tâche appelante (absolu).

`int rt_task_unblock(RT_TASK *task)`
débloquer une tâche temps réel.

`int rt_task_inquire(RT_TASK *task, RT_TASK_INFO *info)`
se renseigner sur une tâche temps réel.

`int rt_task_add_hook(int type, void(*fun)(void *cookie))`
installer un crochet sur une tâche.

`int rt_task_remove_hook(int type, void(*fun)(void *cook))`
enlever le crochet d'une tâche.

`int rt_task_catch(void(*handler)(rt_sigset_t))`
installer un gestionnaire de signaux.

`int rt_task_notify(RT_TASK *task, rt_sigset_t signals)`
envoyer un signal à une tâche.

`int rt_task_set_mode(int clrmask, int setmask, int *mode_r)`
changer le mode d'une tâche.

`RT_TASK * rt_task_self(void)`
obtenir le descripteur de la tâche appelante.

`int rt_task_slice(RT_TASK *task, RTIME quantum)`
fixer le quantum d'une tâche (lorsqu'ordonnancée en round robin).

```
ssize_t rt_task_send(RT_TASK *task, RT_TASK_MCB *mcb_s,
                     RT_TASK_MCB *mcb_r, RTIME timeout)
```
envoyer un message à une tâche.

`int rt_task_receive(RT_TASK_MCB *mcb_r, RTIME timeout)`
recevoir un message d'une tâche.

`int rt_task_reply(int flowid, RT_TASK_MCB *mcb_s)`
réponde à une tâche.

`int rt_task_bind(RT_TASK *task, const char *name, RTIME timeout)`
lier à une tâche temps réel.

`static int rt_task_unbind(RT_TASK *task)`
délier une tâche temps réel.

`int rt_task_same(RT_TASK *task1, RT_TASK *task2)`
comparer deux descripteurs de tâche.

IX – Mécanismes et outils de programmation concurrente

1 Sémaphores

Un sémaphore à compte *est une paire* `(c, f)`*, munie de deux opérations* `P()` *(ou* `sem_wait()` *en norme POSIX) et* `V()` *(ou* `sem_post()` *en norme POSIX).* `c` *est un entier (le compte du sémaphore) et* `f` *une file d'attente.*
Lorsque l'on effectue l'opération `P()` *(acquisition du sémaphore) :*

- *Si le compte* `c` *est négatif, la thread appelante est bloquée et placée dans la file* `f`*.*
- *Sinon, le compte* `c` *est décrémenté d'une unité.*

Lorsque l'on effectue l'opération `V()` *(relâchement du sémaphore) :*

- *Si la file* `f` *est non vide, la première thread en est extraite et elle est placée dans la file d'attente des tâches prêtes de l'ordonnanceur.*
- *Sinon, le compte* `c` *est incrémenté d'une unité.*

On peut voir une analogie entre un sémaphore et un sac de billes.
Voici le pseudo code d'implémentation d'un sémaphore

Exemple de code IX.1

```
typedef struct _Semaphore {
  int     compte;
  boolean verrou;
  Queue   fileSemaphore
```

```
} Semaphore;

  void semWait() {
      while (test_and_set(verrou)) ;
      if (compte <= 0) {
          // Placer la thread appelante dans fileSemaphore
          verrou = 0;
          // Bloquer la thread appelante
      } else {
          compte--;
          verrou = 0;
      }
  }

  void semPost() {
      while (test_and_set(verrou)) ;
      if (fileSemaphore non vide) {
          // Extraire une thread de fileSemaphore
          // Placer cette thread dans la file d'attente des tâches pretes
     } else {
          compte++;
     }
     verrou = 0;
  }
```

2 Mutex

Un mutex (pour MUTual EXculsion) est un mécanisme de synchronisation analogue à un sémaphore binaire, mais avec une notion de propriété : seule la thread qui a acquis le mutex peut le libérer.

Voici un exemple d'utilisation d'un mutex initialisé statiquement, mutex qui sert à synchroniser l'accès au flux stdout. Nous lançons une dizaine de threads, qui attendent une durée aléatoire avant de chercher à acquérir le mutex. L'attente aléatoire sert à perturber un peu l'algorithme de l ?ordonnanceur de façon à éviter de voir les threads se dérouler dans l'ordre croissant.

Exemple de code IX.2

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

static void *routine_threads(void * argument);
```

```
static int aleatoire(int maximum);

pthread_mutex_t mutex_stdout = PTHREAD_MUTEX_INITIALIZER;

int main(void)
{
  int i;
  pthread_t thread;

  for(i = 0; i < 10; i ++)
      pthread_create(& thread, NULL, routine_threads, (void *) i);
  pthread_exit(NULL);
}

static void *routine_threads(void * argument)
{
  int numero = (int) argument;
  int nombre_iterations;
  int i;

  nombre_iterations = 1+aleatoire(3);
  for (i = 0; i < nombre_iterations; i ++) {
      sleep(aleatoire(3));
      pthread_mutex_lock(&mutex_stdout);
      fprintf(stdout, "Le thread %d a obtenu le mutex \n", numero);
      sleep(aleatoire(3));
      fprintf(stdout, "Le thread %d relâche le mutex \n", numero);
      pthread_mutex_unlock(&mutex_stdout);
  }
  return NULL;
}

static int aleatoire(int maximum)
{
  double d;
  d = (double) maximum * rand();
  d = d / (RAND_MAX + 1.0);
  return((int) d);
}
```

Appendices

Flashage d'une carte SD pour raspberry PI