I.U.T. Amiens

Département Informatique

Année Universitaire 2004/2005

Unix : Programmation Système

C. Drocourt

1 - Compilation et programmation 6
1.1 - Préparation du fichier source....6
1.1.1 - L'éditeur vi (mode console)....6
1.1.2 - L'éditeur nedit ou xedit (sous X)....6
1.2 - La compilation sous unix....6
1.3 - Constitution d'une bibliothèque....8
1.3.1 - Statique 8
1.3.2 - Dynamique 8
1.4 - Make....8
1.4.1 - Qu'est-ce que make ? 8
1.4.2 - Le makefile....9
1.4.3 - Les règle 9
1.4.4 - Ecrire un makefile....9
1.4.5 - Comment make interprète-t-il un makefile ? 9
1.4.6 - Exécuter un makefile 10
1.4.7 - Les variables....10
1.4.8 - Les règles....13
1.5 - L'interface C-Unix....14
1.5.1 - Les primitives....14
1.5.2 - Les fonctions....15
1.6 - L'interface shell....15
1.6.1 - Accès aux arguments....15
1.6.2 - Accès à l'environnement....15
1.6.3 - Renvoi d'un code de retour....16
1.7 - Aide à la mise au point....16
2 - Les processus....17
2.1 - Accès aux données du BCP....17
2.1.1 - Identité du processus....17
2.1.2 - Propriétaires du processus....17
2.2 - La primitive fork....17
2.3 - Les primitives exec....19
2.4 - La primitive wait....19
2.5 - Le mécanisme du fork/exec....21
2.5.1 - Fonctionnement canonique d'un shell....21
3 - Les signaux....22
3.1 - Introduction....22
3.1.1 - Les signaux disponibles....22
3.1.2 - Le comportement à la réception d'un signal....22
3.2 - L'envoi des signaux....23
3.2.1 - La primitive kill....23

3.2.2 - Exemple.....23
3.3 - Le masquage des signaux.....23
3.3.1 - Manipulation des ensembles de signaux.....23
3.3.2 - La primitive sigprocmask.....24
3.3.3 - La primitive sigpending.....24
3.3.4 - Exemple.....24
3.4 - Le captage des signaux.....25
3.4.1 - La structure sigaction.....25
3.4.2 - La primitive sigaction.....25
3.4.3 - Exemple.....26
3.5 - L'attente d'un signal.....26
4 - Les entrées-sorties.....27
4.1 - Les différentes tables utilisées.....27
4.1.1 - Au niveau d'un processus.....27
4.1.2 - Au niveau du système.....27
4.1.3 - Appels système et fonctions.....27
4.2 - Les opérations de base.....27
4.2.1 - 0uverture d un fichier.....27
4.2.2 - Création d'un fichier régulier.....28
4.2.3 - Fermeture d'un fichier.....28
4.2.4 - Lecture dans un fichier.....28
4.2.5 - Ecriture dans un fichier.....28
4.2.6 - Déplacement de la position courante.....29
4.3 - Manipulation de descripteurs.....29
4.3.1 - Duplication.....29
4.3.2 - Création d'un tube local.....29
4.4 - Manipulation de i-nœuds.....30
4.4.1 - Création d'un fichier de type tube nommé.....30
4.4.2 - Création d'un fichier de type catalogue.....30
4.5 - Consultation d'un i-nœud.....30
4.5.1 - Introduction.....30
4.5.2 - La structure stat.....30
4.5.3 - La primitive stat.....31
4.5.4 - Les mnémoniques.....31
4.5.5 - Set-uid bit, set-gid bit, sticky bit.....32
4.6 - Modification des caractéristiques d'un i-nœud.....32
4.6.1 - Modification des droits d'accès.....32
4.6.2 - Modification des propriétaires.....32
4.7 - Les fichiers de type catalogue.....32
4.7.1 - Le fichier standard <dirent.h>.....32
4.7.2 - Ouverture d'un fichier catalogue.....32

4.7.3 - Lecture d'une entrée....33
4.7.4 - Fermeture d'un fichier catalogue....33
4.8 - La bibliothèque d'entrée-sortie standard....33
4.8.1 - Le fichier <stdio.h>....33
4.8.2 - Ouverture d'un fichier....33
4.8.3 - Fermeture d'un fichier....33
4.8.4 - Ouverture bibliothèque et système....33
4.8.5 - Lecture dans un fichier....34
4.8.6 - Ecriture dans un fichier....34
4.8.7 - Autre fonctions de la bibliothèque standard d'E/S....34
5 - Les verrous....35
5.1 - Introduction : Les accès concurrents....35
5.2 - Les verrous externes....35
5.2.1 - Le principe....35
5.2.2 - Création d'un fichier verrou....35
5.2.3 - Suppression d'un fichier verrou....35
5.2.4 - Première utilisation :assurer l'unicité d'un processus....35
5.2.5 - Seconde utilisation : accès exclusif à un fichier....36
5.2.6 - Critique du mécanisme....36
5.3 - Les verrous internes....36
5.3.1 - Introduction....36
5.3.2 - Caractéristiques générales....36
5.3.3 - L'implémentation System V....37
6 - Les IPC....38
6.1 - Introduction....38
6.2 - Caractéristiques communes....38
6.3 - Le fichier standard <sys/ipc.h>....38
6.3.1 - Les constantes macro-définies....38
6.3.2 - La structure ipc_perm....39
6.4 - La gestion des clés....39
6.5 - Les commandes IPC System V....39
6.6 - Les sémaphores....40
6.6.1 - Principe initial....40
6.6.2 - Principe des sémaphores Unix....40
6.6.3 - La table des sémaphores....41
6.6.4 - Les commandes....41
6.6.5 - La primitive semget....41
6.6.6 - La primitive semctl....42
6.6.7 - La primitive semop....43
6.7 - Segments de mémoire partagée....43
6.7.1 - Le principe....43

6.7.2 - La table des segments....44
6.7.3 - Les commandes....44
6.7.4 - Création d'un segment et obtention d'un shmid....44
6.7.5 - Opérations de contrôle d’un segment....45
6.7.6 - Attachement à un segment....45
6.7.7 - Le détachement d'un segment....46
6.7.8 - Détachement et suppression....46
7 - Les threads....47
7.1 - Introduction....47
7.1.1 - Processus UNIX et threads....48
7.1.2 - Remarque....48
7.1.3 - Compilation pour les threads....48
7.2 - Les attributs d’un activité....48
7.2.1 - Identification d’une activité....48
7.2.2 - Terminaison d’un activité....49
7.3 - Création et terminaison des activités....49
7.3.1 - Création d’une activité : pthread_create....49
7.3.2 - Terminaison d'une activité 50
7.4 - Synchronisation des activités....52
7.4.1 - Introduction....52
7.4.2 - Synchronisation sur terminaison d'une activité : pthread_join....52
7.4.3 - LES MUTEX 53
7.4.4 - Les conditions 54
7.4.5 - Autres mécanisme associés aux threads....57
7.5 - Exemple d'utilisation des mutex et des conditions 58

1 - Compilation et programmation

1.1 - Préparation du fichier source

1.1.1 - L'éditeur vi (mode console)

Bien que peu convivial, l'éditeur de texte standard d'Unix est bien adapté à la rédaction de sources en C. Il est souhaitable néanmoins de le configurer correctement au moyen de la variable d'environnement *EXINIT* et du fichier de configuration *.exrc* qui est propre au catalogue de travail que l'on a choisi.

Dans le fichier *.profile*, ajouter ces deux lignes:

```
EXINIT="set exrc"
export EXINIT
```

Dans le fichier *.exrc*, écrire la ligne:

```
set redraw autoindent number showmatch showmode
```

1.1.2 - L'éditeur nedit ou xedit (sous X)

1.2 - La compilation sous unix

Outils associés

Le langage C est un langage **compilé**. Il y a:

- les compilateurs: `cc` ou `gcc`
- les debuggers: `gdb`, `dbx`, `xdbx` ou `ups`, `dbxtool` en Open-Windows
- les outils de trace: `trace` en SunOS, `truss` en Solaris, voire `strace`.
- des outils associés: `lint` (qui recherche les éventuels problèmes) , `time`,(qui permet de connaître la rapidité d 'exécution d'un programme), ldd (qui permet de connaître les librairies dynamiques utilisées par un programme), `size`, `nm`, `ar`.

Compiler, c'est:

Depuis un source `a.c`:

- passer le pré-processeur `a.i`
- générer un fichier assembleur `a.s`
- faire l'assemblage de ce fichier pour obtenir `a.o`
- faire l'édition de liens avec les librairies utiles

Les options du compilateur sont:

- `-O` : optimisation demandée
- `-o NAME` : nom du fichier final
- `-c` : s'arrêter au fichier `.o`
- `-g` : générer les informations pour les debuggers

- `-lx` : ajouter la librairie `x` lors de l'édition de liens. Ceci fait référence au fichier:
  - `/usr/lib/libx.a` en cas de compilation statique
  - `/usr/lib/libx.so.X.Y` et `/usr/lib/libx.sa.X.Y` en cas de compilation dynamique. Le chemin des librairies est donné soit par une variable d'environnement pour les librairies personnelles (`LD_LIBRARY_PATH) et comme sous linux dans le fichier de configuration /etc/ld.so.conf.`

    Les avantages de la compilation dynamique sont:

    - gain de place disque : les librairies ne sont pas dupliquées

    - les erreurs se corrigent par remplacement simple des librairies sans recompilation

    - on peut modifier des fonctions par remplacement de la librairie. Exemple: `gethostbyname (3)`

    Les inconvénients sont:

    - non-portabilité: un programme n'est plus autonome, et il dépend des librairies dynamiques (`ldd`) et de leurs emplacements.
- `-L <catalogue>` : Recherche des librairies dans le catalogue indiqué puis dans le catalogue «standard» */lib*.
- `-q list` : Création d'un fichier «point lst» qui contient des informations sur le déroulement de la chaîne de compilation.
- -static : Force l'editione de lien a utiliser des librairies statiques (par défaut, les librairies utilisées sont dynamiques)

Exemple:

```
# en un coup
gcc -o myprog myprog.c
# en 2 coups
gcc -c myprog.c
gcc -o myprog myprog.o
# avec la librairie math en dynamique
gcc -o myprog myprog.c -lm
# avec la librairie math en statique
gcc -o myprog myprog.c -static -lm
```

La compilation séparée

Un gros programme peut être découpé en fichiers indépendants.
Exemple: Soit un fichier principal, une première partie, et une seconde partie, la compilation sera:

```
gcc -c main.c
gcc -c part1.c
gcc -c part2.c
gcc -o main main.o part1.o part2.o
```

1.3 - Constitution d'une bibliothèque

1.3.1 - Statique

La commande *ar* permet de réaliser le compactage de plusieurs fichiers ordinaires en un seul fichier d'archive avec la possibilité de mise à jour, d'ajout...
La commande *ar* gère les droits d'accès au différents fichiers archivés.
On peut donc utiliser *ar* pour gérer une bibliothèque qui n'est autre qu'une collection de fichiers «point o» obtenus par compilation sans édition de liens. Pour être utilisable, cette bibliothèque doit comporter une table des matières et une table des symboles.

Affichage du contenu de la bibliothèque standard du langage C:

ar -tv /lib/libc.a

Création d'une bibliothèque référencée par *libamoi.a* dans le catalogue courant, à partir de tous les fichiers modules objets de ce catalogue:

*ar -r libamoi.a *.o*

Ajout du module *sous.o* à cette bibliothèque:

ar -r libamoi.a sous.o

Mise à jour de la table des symboles par exemple après un déplacement de la bibliothèque:

ar -s libamoi.a

1.3.2 - Dynamique

Création d'une librairie nommée *libamoi.so* dans le catalogue courant, à partir de tous les fichiers modules objets de ce catalogue :

```
cc -o libamoi.so -shared *.o
```

1.4 - Make

1.4.1 - Qu'est-ce que make ?

La commande *make* est utilisée pour la compilation de gros programmes (pas nécessairement en C) dont les sources sont réparties en plusieurs fichiers.
Elle détermine quelles sont les parties qui doivent être recompilées lorsque des modifications ont étés effectuées et appelle les commandes nécessaires à leur recompilation. La commande *make* interprète pour cela un fichier, le *makefile*, qui décrit les relations entre fichiers sources et contient les commandes nécessaires à la compilation. Une fois écrit le *makefile*, il suffit d'invoquer *make* pour exécuter toutes les recompilations nécessaires. Toutefois, on peut utiliser *make* avec des arguments pour contrôler les fichiers à compiler ou pour exécuter des tâches annexes.

1.4.2 - Le makefile

Le *makefile* est un ensemble de règles qui décrivent les relations entre les fichiers sources et les commandes nécessaires à la compilation. Il contient aussi des règles permettant d'exécuter certaines actions utiles comme par exemple nettoyer le répertoire, ou imprimer les sources.

1.4.3 - Les règle

Une règle *rule* a la forme suivante :

```
cible ...   : dependance ...
      action
      ...
      ...
```

Une cible *target* est en général le nom d'un fichier à générer. Ce peut être aussi le nom d'une action à exécuter. Il y a en général une seule cible par règle.
Une dépendance *dependency* est un fichier utilisé pour générer la cible. Plus généralement les commandes correspondant à une cible sont exécutées lorsque au moins une des dépendances de la cible a été modifiée depuis le dernier appel de *make*. Une cible a en général plusieurs dépendances.
Une action *command* est une ligne de commande Unix qui sera exécutée par *make*. Attention, dans la syntaxe du *makefile*, une action est toujours précédée d'une tabulation.

1.4.4 - Ecrire un makefile

Voilà, par exemple, un *makefile* :

```
mystrings:   main.o list.o
        gcc -Wall -ansi -g -o mystrings main.o list.o
main.o: main.c table.h
        gcc -Wall -ansi -g -c main.c
list.o: list.c table.h
        gcc -Wall -ansi -g -c list.c
clean:
        rm mystrings main.o list.o
```

Pour créer l'exécutable mystrings, on tapera *make*, pour supprimer les fichiers objets et exécutables du répertoire, on tapera *make clean*.

1.4.5 - Comment make interprète-t-il un makefile ?

Par défaut, *make* commence par la première cible, mystrings dans notre exemple.
Avant d'exécuter la commande associée, il doit mettre à jour les fichiers qui apparaissent dans ses dépendances (ici main.o et list.o) puisque ceux-ci apparaissent aussi comme des cibles.

Ainsi il interprète successivement

- La règle associée à main.o, à condition qu'au moins l'un des fichiers main.c ou table.h ait été modifié depuis le dernier appel à make
- La règle associée à list.o, à condition qu'au moins l'un des fichiers list.c ou table.h ait été modifié depuis le dernier appel à make
- La règle associée à mystrings, à condition qu'il ait mis à jour main.o ou list.o.

1.4.6 - Exécuter un makefile

La commande *make* interprète par défaut le fichier *makefile* ou *Makefile* si le précédent n'existe pas. On peut donner un nom différent au *makefile* et utiliser l'option -f.
Si aucun argument n'est spécifié, *make* interprète en premier lieu la première cible. N'importe qu'elle autre cible peut être donnée comme argument, auquel cas la règle correspondante est interprétée.
Avec certaines options, *make* peut être utilisé pour exécuter d'autres tâches que celles associées aux règles, par exemple avec l'option -n, il imprime les commandes qu'il devrait exécuter, mais ne les exécute pas, et avec l'option -q, il n'exécute rien et n'imprime rien, mais retourne 0 ou 1 selon que les cibles sont à jour ou non.

1.4.7 - Les variables

1.4.7.1 - Des variables dans le makefile

Dans notre exemple, nous avons écrit trois fois la même commande de compilation, et trois fois les noms des fichiers objets. De telles répétitions sont fastidieuses (dans les gros makefile) et sources d'erreurs.
On peut éviter cela en utilisant des variables *macros* qui permettent à une chaîne de caractères d'être définie une seule fois puis utilisée en plusieurs endroits.
Ainsi le makefile est équivalent à celui de notre exemple :

```
objects = main.o list.o
cccom = gcc -Wall -ansi -g
mystrings: $(objects)
        $(cccom) -o mystrings $(objects)
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
clean:
        rm mystrings $(objects)
```

Le nom d'une variable ne doit pas contenir `:', `#', `=', ni d'espaces. Tout autre chaîne de caractère est valide et *make* fait la différence entre les majuscules et les minuscules.
On récupère la valeur de la variable toto par $(toto) ou par ${toto}.
La substitution d'une variable par sa valeur est purement syntaxique. Elle est effectuée lorsque make lit la variable, sauf si une variable apparaît dans la définition d'une autre variable. Ceci permet d'avoir des définitions récurrentes.

1.4.7.2 - Définitions récurrentes

En utilisant des définitions récurrentes de variables, on peut réécrire notre makefile comme ci-dessous :

```
objects = main.o list.o
cccom = gcc $(ccopts)
ccopts = -Wall -ansi -g

mystrings: $(objects)
        $(cccom) -o mystrings $(objects)
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
clean:
        rm mystrings $(objects)
```

1.4.7.3 - Substitutions

On peut aussi effectuer des substitutions à l'intérieur des variables :

```
objects = main.o list.o
sources = $(objets:.o=.c)
cccom = gcc $(ccopts)
ccopts = -Wall -ansi -g

mystrings: $(objects)
        $(cccom) -o mystrings $(objects)
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
clean:
        rm mystrings $(objects)
print:
        lpr $(sources)
```

Dans cet exemple, la variable source est définie à partir de la variable objects en substituant le suffixe .c au suffixe .o. L'appel de *make print* a pour effet d'imprimer les sources.

La commande : @ echo texte a pour effet d'afficher le texte a l'écran pendant la lecture du makefile par make.

1.4.7.4 - Noms de variables calculés

On peut utiliser des variables pour définir des noms de nouvelles variables.

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
cccom = gcc $(ccopts)
ccopts = -Wall -ansi -g

mystrings: $(representation)_obj
        $(cccom) -o mystrings $(representation)_obj
main.o: main.c table.h
        $(cccom) -c main.c
list.o: list.c table.h
        $(cccom) -c list.c
tree.o: tree.c table.h
        $(cccom) -c tree.c
clean:
        rm mystrings $(objects)
print:
        lpr $(sources)
```

Dans ce cas, une variable list_obj est défini et prends la valeur main.o list.o. Ce sont ces objets qui seront liés pour créer l'exécutable mystrings. Cette fonctionnalité est intéressante dans la mesure ou certaine variable peuvent être définies à l'extérieur du makefile.

1.4.7.5 - Les valeurs des variables

On peut définir une variable de différentes manières :

- En spécifiant sa valeur dans le makefile à l'aide de `=', comme on l'a vu précédemment,
- En spécifiant sa valeur sur la ligne de commande, ainsi dans l'exemple précédent de makefile, la ligne de commande :

```
make representation=tree
```

assignera tree à représentation à la place de list et tree.o sera utilisé pour créer l'exécutable.

De plus, make connaît de nombreuses variables qu'il n'est pas nécessaire de définir :

- toutes les variables d'environnement du shell
- certaines variables prédéfinis, comme CC pour la commande de compilation (cc par défaut), ou CFLAGS pour les arguments de cette commande (nulle par défaut).
- certaines variables qui prennent automatiquement une valeur qui dépend de la règle que make est en train d'interpréter. Extraites de la liste de ces variables, voici les plus courantes :
  $@ : le nom du fichier correspondant à la cible courante
  $< : la première dépendance de la cible courante,
  $^ : toutes les dépendances de la cible courante,
  $? : les dépendances de la cible courante qui sont plus récentes que la cible.

Ainsi, si on utilise cc au lieu de gcc, on peut utiliser ce makefile :

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
cccom = ${CC} $(ccopts)
ccopts = -Wall -ansi -g

mystrings: $(representation)_obj
	$(cccom) -o mystrings $^
main.o: main.c table.h
	$(cccom) -c $<
list.o: list.c table.h
	$(cccom) -c $<
tree.o: tree.c table.h
	$(cccom) -c $<
clean:
	rm mystrings $(objects)
print: .print
.print: $(sources)
	lpr $?
```

1.4.8 - Les règles

1.4.8.1 - Des règles implicites

Certaines manières de construire des cibles à partir de leurs dépendances sont standards et se retrouvent dans la plupart des makefiles.
make possède un certain nombre de règles prédéfinis qui sont appliquées pour construire une cible, si celle-ci n'a pas de règle associée, ou pour construire un fichier qui n'apparaît que comme une dépendance. Dans ce cas, make choisit la règle à appliquer en fonction du nom et du suffixe de la cible.
Ainsi une cible toto.o sera construite à partir de toto.c en utilisant implicitement la règle :

```
${CC} -c ${CFLAGS} toto.c
```

si la ligne

```
toto.o : toto.c
```

apparaît seule dans le makefile ou si toto.o apparaît dans les dépendances d'une autre cible et n'est pas une cible.
On peut empêcher make d'utiliser des règles implicites prédéfinis en utilisant l'option -r ou en redéfinissant la cible prédéfini .SUFFIXES dont les dépendances sont les suffixes connus par make.

Extraites du catalogue de ces règles, voici les plus courantes :

- "fichier".o est généré à partir de "fichier".c par la commande ${CC} -c ${CFLAGS}
- l'éxécutable "fichier" est généré automatiquement à partir des fichiers objets qui sont dans ses dépendances

Ainsi, en utilisant des règles implicites, notre makefile pourrait s'écrire :

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
CFLAGS = -Wall -ansi -g

mystrings: $(representation)_obj
main.o: main.c table.h
list.o: list.c table.h
tree.o: tree.c table.h
clean:
	rm mystrings $(objects)
print: .print
.print: $(sources)
	lpr $?
```

et même, si les fichiers .c ne dépendaient pas de table.h, et si l'éxécutable s'appelait main on pourrait simplement écrire :

```
representation = list
objects = main.o list.o tree.o
```

```
sources = $(objets:.o=.c)
CFLAGS = -Wall -ansi -g

main: $(representation).o
clean:
        rm mystrings $(objects)
print: .print
.print: $(sources)
        lpr $?
```

1.4.8.2 - Définir des règles implicites

Pour éviter l’ambiguïté et le peu de lisibilité du makefile précédent, et pour assurer la portabilité (les règles implicites prédéfinis peuvent varier d'une version à l'autre de make), on peut redéfinir les règles implicites et on peut aussi en définir d'autres.
Une règle implicite contient le caractère % dans la cible comme motif -pattern- pour remplacer n'importe quel nom : ainsi la règle

```
%.o : %.c
        gcc ${CFLAGS} -c %.c
```

remplacera la règle implicite générant les objets à partir des sources. Une bonne version de notre makefile sera :

```
representation = list
objects = main.o list.o tree.o
$(representation)_obj = main.o $(representation).o
sources = $(objets:.o=.c)
CFLAGS = -Wall -ansi -g

mystrings: $(representation)_obj
        gcc -g -o mystrings $^
%.o : %.c table.h
        gcc ${CFLAGS} -c $<
main.o: main.c table.h
list.o: list.c table.h
tree.o: tree.c table.h
clean:
        rm mystrings $(objects)
print: .print
.print: $(sources)
        lpr $?
```

1.5 - L'interface C-Unix

1.5.1 - Les primitives

L'exécution d'une primitive n'est rien d'autre que l'exécution d'une partie de code du noyau. Les primitives permettent de réaliser des opérations "dangereuses" portant en particulier sur:

- La gestion du système de fichier
- La gestion des processus
- La gestion de la communication inter-processus

Un appel système qui échoue renvoie en général la valeur **-1** et la variable externe *errno* est alimentée pour fournir un numéro qui renseigne sur la cause de l'échec.

ON TESTERA TOUJOURS LA RÉUSSITE D'UN APPEL SYSTÈME !

```
#include <stdio.h>
extern int errno
int main()
{
…
if (Appel_Système == -1)
    fprintf(stderr, "Echec: erreur numéro %d\n",errno);
…
}
```

Des mnémoniques d'erreur sont définis dans le fichier à inclure <*sys/errno.h*> et peuvent être utilisés pour affiner la gestion des erreurs

```
if (Appel_Système == -1) && (errno==enoent) /* fichier inexistant*/
```

Enfin, il existe une fonction nommée *perror* qui affiche sur la sortie d'erreur standard un message personnalité suivi d'un message en clair provenant du système.

```
#include<stdio.h>
void perror(const char *MesgPerso)
```

1.5.2 - Les fonctions

Différentes bibliothèques de fonction sont disponibles qui permettent de simplifier le travail du développeur tout en minimisant le nombre d'appel système. L'utilisation d'une fonction nécessite une édition de lien avec la librairie qui contient le module objet où la fonction est définie.

1.6 - L'interface shell

La situation classique est celle d'un utilisateur qui lance une commande depuis son shell de connexion. Le programme de l'utilisateur doit pouvoir récupérer des arguments sur la ligne de commande, accéder aux variables d'environnement et renvoyer au shell un code de retour qui renseignera sur le succès de la commande.

1.6.1 - Accès aux arguments

Le passage des arguments entre le shell et la commande est réalisé au moyen de l'entête :

```
int main(int argn, char *argv[])
```

- argn est le nombre d'arguments, y compris le nom de la commande
- argv[0] est le nom de la commande
- argv[1] est le premier paramètre, etc…

1.6.2 - Accès à l'environnement

Le passage des arguments et de l'environnement est réalisé en une seule opération au moyen de l'entête :

```
int main(int argn, char *argv[], char *arge[])
```

- arge[0] correspond à la première variable d'environnement, etc.

1.6.3 - Renvoi d'un code de retour

La fonction *exit* cause l'arrêt du processus en cours, avec fermeture de tous les fichiers ouverts et renvoie au processus père du code de retour passé en argument. Si le processus père est le shell, on rappelle que le code de retour est contenu dans la variable d'environnement *$?*.

```
#include<stdlib.h>
extern void exit(int CodeRetour)
```

ON S'ATTACHERA À RESPECTER LE PRINCIPE UNIX SUIVANT:

- **CODE DE RETOUR ÉGAL À ZÉRO :** **TOUT S'EST BIEN PASSÉ**
- **CODE DE RETOUR AUTRE :** **ÉCHEC**

1.7 - Aide à la mise au point

La fonction *system* permet de lancer une commande Unix depuis un programme rédigé en C. Cette façon de faire est très lente et ne doit être réservée qu'à la mise au point.

```
#include<stdlib.h>
extern int system(const char *commande)
```

Ex: *system("who");*

2 - Les processus

2.1 - Accès aux données du BCP.

Chaque processus possède un bloc de contrôle (BCP) qui contient toutes ses caractéristiques.

2.1.1 - Identité du processus

```
#include  <unistd.h>
pid_t  getpid(void);
pid_t  getppid(void);
```

2.1.2 - Propriétaires du processus

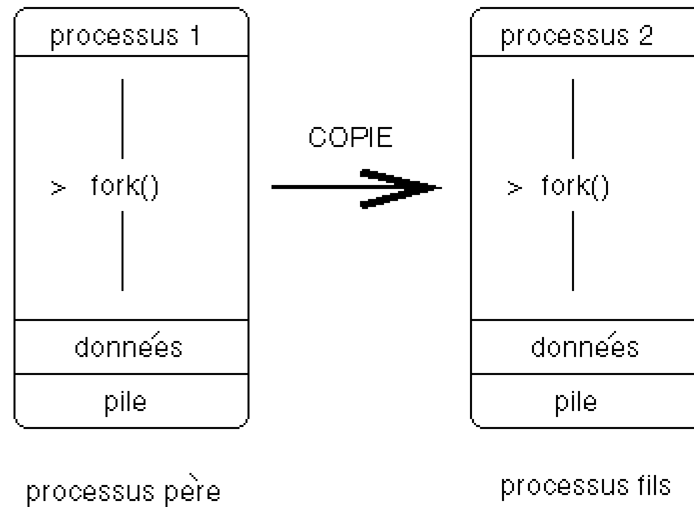
```
#include  <unistd.h>
uid_t  getuid(void);
uid_t  geteuid(void);
gid_t  getgid(void);
gid_t  getegid(void);
```

2.2 - La primitive fork

```
#include  <unistd.h>
pid_t  fork(void) ;
```

En cas d'échec, comme tout appel système, la primitive `fork` renvoie la valeur -1 et la variable *errno* contient le numéro de l'erreur. Il s'agit certainement alors d'un problème de dépassement du nombre de processus permis pour l'utilisateur ou d'une saturation de la table des processus.

En cas de succès, **LE PROCESSUS APPELANT A ÉTÉ CLONÉ** : le processus initial est appelé *père* tandis que le processus cloné est appelé *fils*. Le père garde son PID et continue sa vie tandis que le fils a un nouveau PID et commence la sienne...
Immédiatement après le `fork`, les deux processus exécutent le même code avec les mêmes données, les mêmes fichiers ouverts, le même environnement mais on peut les différentier par la valeur renvoyée par la primitive : **FORK RENVOIE 0 CHEZ LE FILS ET LE PID DU FILS CHEZ LE PÈRE.**




```
processus pere

if ( fork() == 0 )
   code du fils
else
   code du pere

retourne le pid du fils (≠0)
```

```
processus fils

if ( fork() == 0 )
   code du fils
else
   code du pere

retourne 0
```

```
#include <stdio.h>
#include <unistd.h>
#include <stdlib.h>
extern int errno;
main(void) {
pid_t  pid;

if((pid = fork()) == -1)
{
      fprintf(stderr," Echec de fork erreur numéro %d.\n ",errno) ;
      exit(1) ;
}
printf(" Cet affichage est réalisé par le fiston et par son papa.\n ");
if(pid == 0)
      printf(" Cet affichage est réalisé par le fiston.\n ");
else
      printf(" Cet affichage est réalisé par le papa.\n ");
}
```

L’ordre dans lequel les affichages vont avoir lieu est totalement imprévisible car les deux processus père et fils sont concurrents.

2.3 - Les primitives exec

```
#include  <unistd.h>
int execl(const char *ref, const char *arg0,...,(char *)0);
```

Les primitives de la famille *exec** permettent à un processus de charger en mémoire, en vue de son exécution, un nouveau programme binaire. En cas de succès de l'opération, il y a écrasement du code, des données et par conséquent aucune marche arrière n'est possible. Le processus garde son PID et donc la plupart de ses caractéristiques : il n'y a aucune création de processus.

Exemple

```
#include <stdio.h>
#include <unistd.h>
int main(void){
      execl("/usr/bin/ls","ls","-l","/tmp",NULL) ;
      perror("echec de execl.\n");
}
```

2.4 - La primitive wait

Un processus fils qui se termine envoie à son père le signal `SIGCHLD` et reste dans un état dit « Zombie » tant que le processus père n'aura pas consulté son code de retour. La prolifération de processus à l'état zombie doit être formellement évitée, aussi il faut synchroniser le père et le fils à l'aide de la primitive `wait` : le père attend la mort du processus fils.

```
#include  <sys/types.h>
#include  <sys/wait.h>
pid_t  wait(int *ptretat) ;
```

- Si le processus appelant n'a aucun fils, la primitive `wait` renvoie -1 et *errno*...
- Si le processus appelant possède des fils qui ne sont pas à l'état « zombie », il est mis en attente jusqu'à ce que l'un des fils passe dans cet état.
- Si le processus appelant possède des fils à l'état « zombie », la primitive renvoie le PID d'un des fils et écrit à l'adresse *ptretat* un entier qui renseigne sur la façon dont s'est terminé ce fils (code de retour ou numéro de signal ayant tué le processus).
- Si le processus fils est mort « naturellement » par un appel à `exit(code)`, alors l'octet de poids faible de ce code est recopié dans l'octet de poids fort de l'entier pointé par *ptretat* (l'autre octet est nul).
- Si le processus fils a été tué par un signal, alors le numéro de signal (entier inférieur strictement à 128) est recopié dans l'octet de poids faible de l'entier pointé par *ptretat* (l'autre octet est nul). Si un fichier *core* a été créé la valeur décimale 128 est ajoutée à l'entier pointé par *ptretat*.

Le fichier `<sys/wait.h>` contient des fonctions de traitement de cet entier **ptretat* qui sont macro-définies et qui permettent la réalisation d'une « autopsie » des processus fils.

Exemple d'autopsie

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

main(void)
{
      pid_t  pid,pidfils ;
      int    status ;
      printf("Processus père de PID %d\n ",getpid()) ;
      switch(pid = fork())
      { case (pid_t)-1 :
            perror("naufrage... ") ;
            exit(2) ;
        case (pid_t)0 :
            printf(" Début du processus fils PID=%d \n ",getpid()) ;
            /*tempo pour avoir le temps de tuer éventuellement
              le processus fils avant qu'il se termine normalement */
            sleep(30) ;
            exit(33) ;
        default :
            pidfils = wait(&status) ; /* attention adresse ! ! ! */
            printf(" Mort du processus fils PID=%d\n ",pidfils) ;
            if(WIFEXITED(status))
            {
               printf(" code de retour :%d\n ",WEXITSTATUS(status)) ;
               exit(0) ;
       }
       if(WIFSIGNALED(status)) {
              printf(" tué par le signal %d\n ",WTERMSIG(status)) ;
              if(WCOREDUMP(status))
              printf(" fichier core créé\n ") ;
              exit(1) ;
            }
  }
}
```

Le code de retour du fils est égal à 33 sauf si on le tue par l'envoi d'un signal. Le code de retour du père sera égal à 2 si la primitive `fork` a échoué, à 1 si le processus fils a été tué et à 0 si le processus fils s'est auto-détruit en appelant la fonction `exit`. C'est parce que la dernière éventualité est considérée comme « normale » que la valeur du code de retour du père au processus shell « grand-père » est zéro.

En plus de l'autodestruction d'un processus, la fonction *exit* réalise la libération des ressources allouées au processus au cours de son exécution.

Autre façon d'éliminer les zombies

La prolifération de processus à l'état zombie doit être formellement évitée, aussi lorsque le père n'a pas à être synchronisé sur ses fils, on peut inclure dans le code du père l'instruction `signal(SIGCHLD,SIG_IGN)` pour éliminer les fils à l'état Z au fur et à mesure de leur apparition.

2.5 - Le mécanisme du fork/exec

L'association du `fork` et de l'`exec` permet de créer son propre shell : le processus père saisie la commande de l'utilisateur et délègue à son fils l'exécution de la commande.

2.5.1 - Fonctionnement canonique d'un shell

Un interpréteur de commandes, utilisé de façon interactive, fonctionne en boucle infinie, selon le shéma suivant :

1. Lecture ligne de commandes
2. Interprétation (substitutions...)
3. Clonage du shell avec `fork`
4. Chez le père : attente du fils avec `wait` puis retour en 1
   Chez le fils : changement de code avec `exec` puis mort.

3 - Les signaux

3.1 - Introduction

3.1.1 - Les signaux disponibles

Sur un système donné, on dispose de **NSIG** signaux numérotés de 1 à **NSIG**. La constante **NSIG**, ainsi que les différents signaux et les prototypes des fonctions qui les manipulent, sont définis dans le fichier `<signal.h>`.

```
$grep  SIGHUP  /usr/include/signal.h
#define SIGHUP  1 /* Fin du processus leader de session */
$
```

3.1.2 - Le comportement à la réception d'un signal

3.1.2.1 - Terminologie des signaux

Un signal envoyé par le noyau ou par un autre processus est un signal **pendant** : cet envoi est mémorisé dans le BCP du processus.

Un signal est **délivré** (ou pris en compte) lorsque le processus concerné réalise l'action qui lui est associée dans son BCP , c'est à dire, au choix :

- l'action par défaut : en général la mort du processus
- ignorer le signal
- l'action définie par l'utilisateur (handler) : le signal est dit **capté**.

Un signal peut également être **masqué** (ou bloqué) : sa prise en compte sera différée jusqu'à ce que le signal ne soit plus masqué.

3.1.2.2 - Limites des signaux

Lorsqu'un processus est en sommeil et qu'il reçoit plusieurs signaux :

- Aucune mémorisation du nombre de signaux reçus : 10 signaux SIGINT ≡ 1 signal SIGINT.
- Aucune mémorisation de la date de réception d'un signal : les signaux seront traités ultérieurement par ordre de numéro.
- Aucun moyen de connaître le PID du processus émetteur du signal.

3.1.2.3 - La délivrance des signaux

Attention, les signaux sont asynchrones : la délivrance des signaux non masqués a lieu un « certain temps » après leur envoi, quand le processus récepteur passe de l'état actif noyau à l'état actif utilisateur. Cela explique pourquoi un processus n'est pas interruptible lorsqu'il exécute un appel système.

3.2 - L'envoi des signaux

3.2.1 - La primitive kill

```
#include  <sys/types.h>
#include  <signal.h>
int kill(pid_t  pid,int  sig) ;
```

Les processus émetteur et récepteur doivent avoir le même propriétaire ! ! !

Le « faux » signal 0 peut être envoyé pour tester l'existence d'un processus.

3.2.2 - Exemple

Le processus père teste l'existence de son fils avant de lui envoyer le signal SIGUSR1.

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

int main(void) {
  pid_t pid,pidBis;
  int status;
  switch(pid = fork())
  {
    case (pid_t)0:
      while(1) sleep(1);
    default:
      sleep(10);
      if(kill(pid,0) == -1) {
        printf("processus fils %d inexistant",pid);
        exit(1);
      }
      else {
        printf("envoi de SIGUSR1 au fils %d",pid);
        kill(pid,SIGUSR1);
      }
           pidBis=wait(&status);
           printf("Mort de fils %d avec status=%d",pidBis,status);
           exit(0);
       }
}
```

3.3 - Le masquage des signaux

3.3.1 - Manipulation des ensembles de signaux

Le type `sigset_t` correspond à de tels ensembles.

Des fonctions sont disponibles pour construire et manipuler les ensembles de signaux avant de mettre en place un masquage collectif de ces signaux ou une suppression du masque courant.

3.3.2 - La primitive sigprocmask

```
#include  <signal.h>
int sigprocmask(int opt,const sigset_t *new,sigset_t old) ;
```

Cette primitive permet l'installation manuelle d'un masque à partir de l'ensemble pointé par `new` et éventuellement du masque antérieur que l'on récupère au retour de la primitive à l'adresse `old` si le troisième paramètre n'est pas le pointeur nul.

Le paramètre `opt` précise ce que l'on fait avec ces ensembles :

Valeur du paramètre opt	*Nouveau masque*
SIG_SETMASK	*new
SIG_BLOCK	*new U *old
SIG_UNBLOCK	*old - *new

3.3.3 - La primitive sigpending

```
#include  <signal.h>
int sigpending(sigset_t *ens) ;
```

Ecrit à l'adresse `ens` la liste des signaux pendants qui sont masqués.

3.3.4 - Exemple

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

int main(void) {
      sigset_t ens1,ens2;
      int sig;
      /* Construction ensemble {SIGINT,SIGQUIT,SIGUSR1}  */
      /* sigemptyset initialise l'ens. de signaux à vide */
      sigemptyset(&ens1);
      sigaddset(&ens1,SIGINT);
      sigaddset(&ens1,SIGQUIT);
      sigaddset(&ens1,SIGUSR1);

      /* Mise en place du masquage des signaux de cet ens. */
      sigprocmask(SIG_SETMASK,&ens1,(sigset_t *)0);
      printf("Masquage mis en place.\n");
      sleep(15);
      /*Lecture des signaux envoyés mais non délivrés car masqués*/
      sigpending(&ens2);
      printf("Signaux pendants:\n");
      for(sig=1;sig<NSIG;sig++)    if(sigismember(&ens2,sig))   printf("%d \n",sig);
      sleep(15);
      /* Suppression du masquage des signaux */
      sigemptyset(&ens1);
      printf("Déblocage des signaux.\n");
      sigprocmask(SIG_SETMASK,&ens1,(sigset_t *)0);
      sleep(15);
      printf("Fin normale du processus\n");
      exit(0);
}
```

Pour la suppression du masquage, on pouvait garder le même ensemble *ens1* et faire : `sigprocmask (SIG_UNBLOCK,&ens1,&ens1);`

3.4 - Le captage des signaux

3.4.1 - La structure sigaction

Le comportement général d'un processus lors de la délivrance d'un signal correspond à la structure *sigaction* :

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

sa_handler peut être `SIG_DFL` ou `SIG_IGN`…ou un pointeur sur la fonction chargée de gérer le signal envoyé.
Le champ *sa_mask* correspond à une liste de signaux qui doivent être ajoutés, pendant l'exécution du handler à ceux déjà masqués.
Si on utilise la primitive `sigaction`, le signal en cours de délivrance sera automatiquement ajouté à cette liste.

3.4.2 - La primitive sigaction

La fonction `sigaction` permet d'installer le captage d'un signal tout en masquant un ensemble de signaux (y compris, par défaut, le signal capté).

```
#include <signal.h>
int sigaction(int sig, struct sigaction *p_action, struct sigaction *p_action_anc) ;
```

La délivrance du signal *sig* entraîne l'exécution du handler de *p_action*. *P_action_anc* permet de retrouver l'ancien comportement du signal.

3.4.3 - Exemple

```
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
sigset_t ens;
struct sigaction action;

void handler(int sig) {
  int i;
  printf("Entrée dans le handler avec le signal: %d\n",sig);
  /* sigprocmask fournit l'ensemble des signaux masqués */
  sigprocmask(SIG_BLOCK,(sigset_t *)0,&ens);
  printf("Signaux bloqués: ");
  for(i=1;i<NSIG;i++)
  { if(sigismember(&ens,i))  printf("%d "); }
  putchar('\n');
  if(sig == SIGINT)
  {
    action.sa_handler=SIG_DFL;
    sigaction(SIGINT,&action,NULL);
    /* Comportement standard au signal SIGINT rétabli. */
  }
  printf("Sortie du handler\n");
}

int main(void) {
       action.sa_handler=handler;
       sigemptyset(&action.sa_mask);
       sigaction(SIGQUIT,&action,(struct sigaction *)0);
  /* Seul SIGQUIT sera masqué pdt l'exécution du handler.*/
       sigaddset(&action.sa_mask,SIGQUIT);
       sigaction(SIGINT,&action,(struct sigaction *)0);
  /*SIGINT et SIGQUIT masqués pendant l'exécution du handler*/
       while(1) sleep(1);
}
```

3.5 - L'attente d'un signal.

```
#include  <signal.h>
int sigsuspend(const  sigset_t  *ens) ;
```

Cette primitive réalise atomiquement :

- l'installation du masque de signaux pointé par *ens*
- la mise en sommeil jusqu'à l'arrivée d'un signal non masqué qui va provoquer la mort du processus ou l'exécution du handler installé pour ce signal.

4 - Les entrées-sorties

4.1 - Les différentes tables utilisées

4.1.1 - Au niveau d'un processus

Chaque processus dispose, dans son bloc de contrôle, d'une table de descripteurs qui correspondent aux fichiers qu'il utilise:

- Fichiers ouverts par défaut:
  - 0 *stdin*
  - 1 *stdout*
  - 2 *stderr*
- Fichiers ouverts par le processus:
  - 3 *beurk*
  - …

Par l'intermédiaire du descripteur, le processus accède à la table des fichiers du système.

4.1.2 - Au niveau du système

La table des fichiers du système possède autant d'entrées qu'il y a de fichiers utilisés sur la machine Unix. Chaque entrée contient notamment:

- le nombre total de descripteurs lui correspondant dans le système
- le mode d'ouverture du fichier
- la position courante (offset)

4.1.3 - Appels système et fonctions

Les **primitives** réalisent les opérations d'entrée-sortie de bas niveau, permettant de travailler directement en zone système, sur les caches des fichiers.
Les **fonctions** de la bibliothèque d'entrée-sortie, qui correspondent au fichier à inclure <*stdio.h*>, sont installées «au dessus» de ces appels système.

4.2 - Les opérations de base

4.2.1 - 0uverture d un fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *chemin,int typeOpen,mode t droitsFic);
```

Cette primitive restitue un descripteur pour le fichier dont la référence est pointée par le paramètre chemin ; en cas de problème, elle restitue -1 et *errno*...

Le paramètre *typeOpen* définit le type d'ouverture désiré, il est écrit avec une disjonction de bits dont les mnémoniques sont définis dans le fichier <*fcntl.h*> :

- 0_RDONLY lecture seule
- 0_WRONLY écriture seule
- 0_RDWR lecture et écriture
- 0_APPEND écriture en fin de fichier
- 0_TRUNC remise à zéro du fichier à l'ouverture
- 0_CREAT création d'un nouveau fichier si nécessaire
- 0_EXCL utilisé avec le bit 0_CREAT: si le fichier existe déjà, la primitive revient en erreur avec *errno*=EEXIST.

Le paramètre *droitsFic*, qui spécifie les droits d'accès, n'est utilisé que dans le cas de la création d'un fichier ordinaire. L'inclusion du fichier <*sys/stat.h*> ,incluant lui-même le fichier <*sys/mode.h*>, autorise l'utilisation des mnémoniques de bits définis dans ce dernier fichier.
Ainsi, on peut remplacer *droitsFic=0742* par :
droitsFic=S_IRWXU|S_IRGRP|S_IWOTH.

4.2.2 - Création d'un fichier régulier

Elle est réalisée soit par la primitive *creat*, soit par la primitive *open* avec le bit 0_CREAT levé.
creat("/tmp/Fic",0777); équivaut à
open("/tmp/Fic",0_WRONLY|0_CREAT|0 TRUNC,0777);

4.2.3 - Fermeture d'un fichier

```
#include <unistd.h>
int close(int desc);
```

L'appel système *exit* réalise la fermeture de tous les fichiers ouverts par le processus.

4.2.4 - Lecture dans un fichier

```
#include <sys/types.h>
#include <unistd.h>
int read(int desc,void *buffer,unsigned nb);
```

Attention, le paramètre *buffer* doit pointer sur une zone de taille suffisante pour recevoir les *nb* octets lus en séquence (lecture+**déplacement**) dans le fichier de descripteur *desc* ! La primitive restitue le nombre d'octets effectivement lus si tout va bien, zéro si la fin du fichier est atteinte et -1 si il y a eu un problème (avec errno...)

4.2.5 - Ecriture dans un fichier

```
#include <sys/types.h>
#include <unistd.h>
int write(int desc,void *buffer,unsigned nb);
```

4.2.6 - Déplacement de la position courante

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int desc,off_t depl,int vers);
```

Le paramètre *depl* qui peut être positif, négatif ou nul précise le déplacement qui sera réalisé par rapport à une origine qui est donnée par le paramètre *vers*. Le fichier <*unistd.h*> contient trois constantes macro-définies qui sont les seules valeurs autorisées pour le paramètre *vers*:

- SEEK_SET La position courante devient *depl*
- SEEK_CUR Déplacement à partir de la position courante
- SEEK_END Déplacement à partir de la fin du fichier

La primitive renvoie la nouvelle position courante ou -1 si problème.

4.3 - Manipulation de descripteurs

4.3.1 - Duplication

```
#include <unistd.h>
int dup(int desc);
```

La primitive restitue un nouveau descripteur pour le fichier ouvert de descripteur *desc*. Il est garanti, et c'est primordial, que le nouveau descripteur sera le plus petit possible.

L'exemple ci-dessous montre comment on réalise une redirection de *stderr* sur le fichier nommé *FicErr* dans le catalogue courant

```
…
descFicErr=creat("FicErr", 0600) ;
close(2);
dup(descFicErr);
…
```

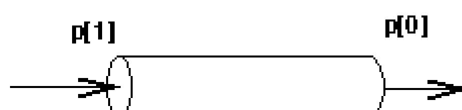
Le tableau suivant montre l'évolution de la table des descripteurs

desc	Après creat	après close	après dup
0	sdtin	sdtin	sdtin
1	stdout	stdout	stdout
2	stderr	libre	FicErr
3	FicErr	FicErr	FicErr
4	libre	libre	libre

4.3.2 - Création d'un tube local

```
int pipe(int desc[2])
```

En cas de succès, deux descripteurs sont disponibles : *desc[1]* (entrée du tube) et *desc[0]* (sortie du tube). On peut alors utiliser la primitive *write* pour écrire dans le tube en se servant de *desc[1].* Pour placer une fin de fichier, il suffit de fermer ce descripteur. Pour lire, on utilise *read* sur *desc[0].* Le processus créateur et sa descendance sont les seuls à pouvoir utiliser un tube local.



4.4 - Manipulation de i-nœuds

4.4.1 - Création d'un fichier de type tube nommé

Contrairement à un tube ordinaire, un tube nommé possède une référence et peut donc être utilisé par un processus appartenant à n'importe quel utilisateur. Néanmoins, cette technique de communication ne peut être mise en œuvre entre deux réseaux Unix.

```
#include<sys/types.h>
#include<sys/stat.h>
int mkfifo(const char *chemin, mode_t DroitsFic)
```

4.4.2 - Création d'un fichier de type catalogue

```
#include <sys/types.h>
#include <sys/stat.h>
int mkdir(const char *chemin,mode t droitsFic);
```

La primitive POSIX *mkdir* crée un fichier de type catalogue dont la référence est pointée par le paramètre *chemin* avec les droits d'accès précisés par le paramètre *droitsFic* (voir la primitive *open*).

Deux autres primitives *rmdir* et *chdir* permettent respectivement de détruire un catalogue vide et de faire d'un catalogue donné le catalogue courant.

4.5 - Consultation d'un i-nœud

4.5.1 - Introduction

Rappelons tout d'abord qu'à chaque fichier UNIX est associé un i-nombre qui indexe un i-noeud dans un système de fichiers (partition du disque dur ou disque logique).Les couples (i-nombre,référence) sont les seules choses qui soient contenues dans un fichier de type catalogue (répertoire). Le i-noeud d'un fichier contient donc tous les renseignements concernant ce fichier: propriétaire, droits d'accès...

L'appel système stat permet d'obtenir, dans un buffer déclaré par l'utilisateur, tous les renseignements stockés dans le i-noeud d'un fichier. En cas de problème, la valeur de retour est égale à -1 et la variable externe errno donne le numéro de l'erreur.

4.5.2 - La structure stat

Le fichier <*sys/stat.h*> contient la définition de la structure *stat* qui est utilisée pour obtenir les informations contenues dans le nœud d'un fichier.

```
#include <sys/types.h>
#include <sys/stat.h>
struct stat {
  dev_t st_dev,              /* identification du disque logique  */
  ino_t st_ino;              /* i-nombre du fichier sur ce disque */
  mode_t st_mode,            /* type et droits d'accès            */
  nlink_t st_nlink;   /* nombre de liens physiques (Hard)  */
  uid_t st_uid;              /* propriétaire du fichier           */
  gid_t st_gid;              /* groupe propriétaire du fichier    */
  off_t st_size;             /* taille du fichier                 */
  time_t st_atime;    /* date de dernier accès            */
  time_t st_mtime ;   /* date de dernière modification    */
  time_t st_ctime;    /* date de dern. modif. du i-noeud  */
}
```

Ces nombres comptent les secondes écoulées depuis le 1er Janvier 1970. La fonction ctime de la bibliothèque standard permet de les remettre sous un format habituel.

```
#include <time.h>
extern char  *ctime(const time_t *);
```

4.5.3 - La primitive stat

```
#include <sys/types.h>
#include <sys/stat.h>
int stat(const char *chemin, struct stat *buffer);
```

Le paramètre *chemin* pointe sur la référence du fichier à traiter. Le paramètre *buffer* pointe sur un objet de structure *stat* déjà défini.

En cas de succès, la primitive *stat* réalise une copie des informations concernant le fichier du i-noeud vers le buffer.

4.5.4 - Les mnémoniques

Le fichier de référence *<sys/mode.h>* contient un certain nombre de mnémoniques pour coder les droits d'accès, le type du fichier et le positionnement éventuel des bits spéciaux (sticky, set-uid, set-gid). Ces constantes y sont définies par des directives au précompilateur.

Exemples :

```
#define S_IRGRP 0000040
 0    0      0      0      0      4      0     octal
    0 0 0  0 0 0  0 0 0  0 0 0  1 0 0  0 0 0   binaire
                         - - -  r - -  - - -   mnémo
```

Droit de lecture pour les membres du groupe.

```
#define _S_IFMT 0170000
 0    1      7      0      0 0    0      0     octal
    0 0 1  1 1 1  0 0 0  0 0 0  0 0 0  0 0 0   binaire
```

Masque de récupération des bits concernant le type d'un fichier.

```
#define _S_IFBLK 0060000
 0    0      6      0      0      0      0     octal
    0 0 0  1 1 0  0 0 0  0 0 0  0 0 0  0 0 0   binaire
```

Fichier spécial bloc.

Pour savoir si un fichier est de type spécial bloc:

```
(_S_IFMT & buffer.st_mode) == _S_IFBLK.
```

Pour savoir si un fichier est lisible par le groupe propriétaire:

```
(S_IRGRP & buffer.st_mode) == S_IRGRP
```

Le fichier `<sys/mode.h>` contient des fonctions macro-définies qui permettent de tester plus simplement si un fichier appartient à un type donné :

```
#define S_ISBLK(m) (( (m) & (_S_IFMT)) == ( _S_IFBLK))
```

Le programmeur remplacera donc le test donné plus haut par:

```
S_ISBLK(buffer.st_mode)
```

4.5.5 - Set-uid bit, set-gid bit, sticky bit

Le *set-uid bit*, lorsqu'il est positionné pour un fichier binaire exécutable, indique que le processus correspondant à son exécution possède les droits du propriétaire du fichier et non de l'utilisateur qui le lance (c'est ce mécanisme qui permet à tout utilisateur de modifier */etc/passwd*, dont le super-utilisateur est propriétaire, via la commande *passwd*).
Le *set-guid* bit a le même rôle, mais relativement au groupe.
Le *sticky bit* assure le maintient d'un programme en zone de swap.

4.6 - Modification des caractéristiques d'un i-nœud

4.6.1 - Modification des droits d'accès

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

chmod permet de modifier les droits d'accès à condition que le propriétaire effectif du processus soit le propriétaire du fichier.

4.6.2 - Modification des propriétaires

```
#include <unistd.h>
int chown(const char *chemin, uid_t newPropr, gid_t newGrpPropr);
```

La primitive *chown* permet de modifier le propriétaire et le groupe propriétaire d'un fichier avec les mêmes restrictions qu'au dessus. Dans certaines versions d'Unix cette primitive est réservée à root.

4.7 - Les fichiers de type catalogue

4.7.1 - Le fichier standard <dirent.h>

Il contient les éléments nécessaire pour la lecture du contenu des fichiers de type catalogue:

- Le type *DIR* : type correspondant au fichier de type catalogue.
- La structure *dirent* : correspond à une entrée dans un catalogue et contient au moins deux champs *d_name* (référence du fichier) et *d_ino* (i-nombre du fichier).
- Les prototypes des fonctions POSIX utilisées.

4.7.2 - Ouverture d'un fichier catalogue

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

Cette fonction réalise l'ouverture du fichier répertoire, alloue un objet de type DIR et renvoie l'adresse de cette objet. En cas d'échec, la fonction renvoie le pointeur NULL.

4.7.3 - Lecture d'une entrée

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

Cette fonction lit l'entrée suivante du catalogue et renvoie l'adresse d'une structure *dirent*.
En fin de catalogue ou en cas d'erreur, la fonction renvoie le pointeur NULL.

4.7.4 - Fermeture d'un fichier catalogue

```
#include<dirent.h>
int closedir(DIR *ptr);
```

4.8 - La bibliothèque d'entrée-sortie standard

Cet ensemble de fonction d'E/S, très portable, constitue une couche au dessus des primitives POSIX qui ont déjà été étudiées. Il permet de diminuer le nombre d'appels système.

4.8.1 - Le fichier <stdio.h>

- Macro définition de constante. En particulier NULL (pointeur nul) et EOF (indication de fin de fichier).
- Définition du type FILE. A chaque fichier ouvert au niveau de la bibliothèque standard est associé un objet de ce type. Parmi les champs de la structure, on trouvera notamment un descripteur.
- Prototypes de fonctions.

4.8.2 - Ouverture d'un fichier

```
#include<stdio.h>
FILE *fopen(const char *ref, const char *mode);
```

Cause l'ouverture du fichier dont la référence est pointée par *ref* et le renvoie d'un pointeur sur l'objet de type FILE associé à ce fichier.
En cas d'échec, la fonction renvoie NULL.
Le paramètre *mode* précise le mode d'ouverture dont les trois principaux sont:

- r lecture
- w écriture avec troncature ou création
- a écriture en fin de fichier ou création

4.8.3 - Fermeture d'un fichier

```
#include<stdio.h>
int fclose(FILE *ptr)
```

4.8.4 - Ouverture bibliothèque et système

Il est possible de récupérer le descripteur (niveau système) associé à un fichier ouvert au niveau de la bibliothèque

```
#include<stdio.h>
int fileno(FILE *ptr)
```

4.8.5 - Lecture dans un fichier

```
#include<stdio.h>
int fgetc(FILE *ptr);
char *fgets(char *buf, int taille, FILE *ptr);
int fread(void *buf, size_t taille, size_t nombre, FILE *ptr);
```

La fonction *fgetc* lit un caractère. Une fin de fichier ou une erreur est signalée par la valeur EOF.
La fonction *fgets* lit une chaîne d'au plus *taille*-1 caractères, un caractère ASCII nul étant ajouté en dernière position, et renvoie *buf*, l'adresse de cette chaîne. (NULL si échec).
La fonction *fread* lit *nombre* objets de *taille* caractères qu'elle écrit à partir de l'adresse *buf* et renvoie le nombre d'objets lus.

```
#include<stdio.h>
int fscanf(FILE *ptr, const char *format,…);
```

Cette fonction réalise une lecture formatée et renvoie soit le nombre de conversion réalisées, soit EOF en cas de fin de fichier ou d'erreur.
Il faut se rappeler qu'à partir du troisième paramètre, on passe des **adresses** d'objets !
La fonction *scanf* est un cas particulier de cette fonction avec un premier paramètre égal à *stdin*.

4.8.6 - Ecriture dans un fichier

```
#include<stdio.h>
int fputc(int c, FILE *ptr);
int fputs(char *buf, FILE *ptr);
int fwrite(void *buf, size_t taille, size_t nombre, FILE *ptr);
```

La fonction *fputc* écrit le caractère (unsigned char)*c*.
La fonction *fputs* écrit la chaîne pointée par *buf*, sans recopier le caractère ASCII nul de fin de chaîne.
La fonction *fwrite* écrit *nombre* objets de *taille* caractères qu'elle a trouvés à partir de l'adresse *buf* et renvoie le nombre d'objets écrits.
Toutes ces fonction renvoient une valeur négative en cas de problème.

```
#include<stdio.h>
int fprintf(FILE *ptr, const char *format,…);
```

Cette fonction réalise une écriture formatée et renvoie le nombre de conversion réalisées, ou une valeur négative en cas d'erreur.
Contrairement à *fscanf*, à partir du troisième paramètre, on passe des **noms** d'objets !
La fonction *printf* est un cas particulier de cette fonction avec un premier paramètre égal à *stdout*.

4.8.7 - Autre fonctions de la bibliothèque standard d'E/S

Déplacement de la position courante : *fseek*.
Test de la fin de fichier : *feof*.

5 - Les verrous

5.1 - Introduction : Les accès concurrents

Dans un système d'exploitation multitâches, les différents processus sont concurrents et par conséquent sont susceptibles d'accéder à un fichier régulier en « même temps ». La nécessité de régler ce type de conflit est particulièrement fondamentale dans le développement d'applications de gestion.

5.2 - Les verrous externes

5.2.1 - Le principe

Un verrou externe est tout simplement un fichier régulier dont la seule présence constitue le verrouillage de la ressource à protéger.

5.2.2 - Création d'un fichier verrou

On utilise la primitive `open` habituelle avec deux drapeaux particuliers `O_CREAT` (création de fichier) et `O_EXCL` (en mode exclusif) .
Si le fichier verrou n'existe pas, la primitive s'exécutera normalement et renverra un descripteur associé à ce fichier verrou.
Si le fichier verrou existe déjà, la primitive échouera et renverra -1 avec *errno=EEXIST* .

5.2.3 - Suppression d'un fichier verrou

On utilise la primitive `unlink` habituelle.

5.2.4 - Première utilisation :assurer l'unicité d'un processus

On veut se garder ici de lancer plusieurs fois la même commande (typiquement un processus démon assurant un service) : le verrouillage externe est parfaitement adapté à ce type de fonctionnalité.

```
#include <fcntl.h>
#include <sys/errno.h>
#include <stdio.h>

extern int errno;
int main(void)
{ int desc;
  if((desc=open("cad",O_CREAT|O_EXCL,0))==-1 && errno== EEXIST)
    exit(1) ;
  else
  {
    close(desc);
    /* Action du processus */
    unlink("cad");
    exit(0);
  }
}
```

5.2.5 - Seconde utilisation : accès exclusif à un fichier

On veut s'assurer ici que le processus accède seul à une ressource pendant une partie de son existence. Avant de rentrer dans cette section « critique », le processus essaie de créer un fichier verrou externe et se met en attente si nécessaire.

```
#include <fcntl.h>
#include <sys/errno.h>

extern int errno;

int main(void)
{
  int descV;

  while((descV=open("cad",O_CREAT|O_EXCL,0))==-1&&errno==EEXIST)
    sleep(10);
  /* Section critique */
  unlink("cad ");
  /* Suite programme*/
  exit(0);
}
```

5.2.6 - Critique du mécanisme

- Accès disque trop nombreux.
- Attente active coûteuse en temps CPU.
- Fiabilité contestable.
- Risque d'interblocage (deadlock).

5.3 - Les verrous internes

5.3.1 - Introduction

Les verrous internes sont davantage associés aux fichiers à protéger en ce sens que leur présence va être détectée dans les tables gérées par le système. Les mécanismes mis en place dans System V (repris par POSIX) et BSD diffèrent aussi bien au niveau des fonctionnalités qu'au niveau de l'implémentation d'où problèmes de portabilité des applications.

5.3.2 - Caractéristiques générales

- Propriétaire : un processus.
- Portée : ensemble des positions du fichier qui sont protégées.
- Mode opératoire : soit impératif (*mandatory*, empêche les E/S), soit consultatif (*advisory*, n'empêche rien) selon que les processus sont obligés ou pas de le consulter.
- Type : soit partagé (shared) soit exclusif (exclusive) selon que plusieurs verrous peuvent cohabiter sur un même fichier ou non.
  verrous partagés pour la lecture
  verrous exclusifs pour l'écriture

5.3.3 - L'implémentation System V

Le noyau gère une table de verrous dont chaque entrée a une structure `flock` qui peut être consultée ou modifiée avec la primitive `fcntl`.

5.3.3.1 - Champs de la structure flock

`l_type` type du verrou

`F_RDLCK`	partagé (en lecture)
`F_WRLCK`	exclusif (en écriture)
`F_UNLCK`	absence de verrou

`l_whence` type de repérage dans le fichier à verrouiller

0	par rapport au début
1	par rapport à la position courante
2	par rapport à la fin

`l_start` position du début de la zone à verrouiller relativement à `l_whence`

`l_len` longueur de la zone à verrouiller (longueur nulle = tout le fichier)

`l_pid` PID du processus propriétaire (fournie au retour dans certains cas)

5.3.3.2 - Manipulation des verrous avec la primitive fcntl

```
#include  <fcntl.h>
int  fcntl(int desc,int oper,struct flock *buf_adr) ;
```

Le paramètre *desc* est le descripteur du fichier à verrouiller qui aura été ouvert auparavant.

Le paramètre *oper* désigne l'opération à réaliser sur le verrou :

`F_SETLK`	pose non bloquante : en cas d'échec, c.a.d. si un verrou incompatible existe déjà, la primitive revient en erreur
`F_SETLKW`	pose bloquante (attend de pouvoir mettre le verrou)
`F_GETLCK`	test d'existence et information sur le verrou

Le dernier paramètre est l'adresse d'un objet de structure `flock` qui servira à décrire le verrou à mettre en place (`F_SETLK` ou `F_SETLKW`) ou à récupérer la description d'un verrou existant (`F_GETLCK`).

5.3.3.3 - Mode opératoire

Par défaut, les verrous System V sont consultatifs : rien n'empêche un processus ne jouant pas le jeu d'éditer un fichier verrouillé ! ! !

Certaines implémentations autorisent les verrous impératifs : d'abord on positionne le set-gid bit sur le fichier à protéger, sans que le bit d'exécution soit levé pour le groupe, puis on pose le verrou interne sur le fichier. Le fichier sera déverrouillé par le système si le processus propriétaire du verrou est tué ou meurt avant d'avoir levé le verrou.

6 - Les IPC

6.1 - Introduction

Mécanismes « Inter Processus Communication System V » :

- *Files de messages*

  implantation Unix du concept de boîte à lettres
- *Segments de mémoire partagée*

  zones mémoire accessibles à plusieurs processus
- *Sémaphores*

  mécanisme de synchronisation de processus

6.2 - Caractéristiques communes

Le système gère une table spécifique pour chaque type d'I.P.C. System V. Chaque objet possède une **identification interne** (nombre entier naturel) dont la connaissance est indispensable pour accéder à l'objet ; cette identification est obtenue soit par héritage soit par interrogation du système à l'aide de primitives :

	Primitive	Identification interne
Files de messages	msgget	msqid
Segments de mémoire partagée	shmget	shmid
Ensemble de sémaphores	semget	semid

Chaque objet possède **une clé** pour l'identification externe : il s'agit d'un entier de type `key_t` défini dans `<sys/types.h>`. Cette clé joue le même rôle qu'une référence pour un fichier. Une clé particulière `IPC_PRIVATE` réserve l'utilisation de l'objet au processus créateur et à sa descendance.

6.3 - Le fichier standard <sys/ipc.h>

6.3.1 - Les constantes macro-définies.

Constante	Rôle et interprétation	Primitives
IPC_PRIVATE	un nouvel objet est créé dont l'identification ne pourra plus être demandé au système	msgget, shmget, semget
IPC_CREAT	un nouvel objet est créé s'il n'existe pas	msgget, shmget, semget
IPC_EXCL	utilisé conjointement à IPC_CREAT : retour en erreur si l'objet existe déjà	msgget,shmget, semget
IPC_NOWAIT	opération non bloquante	msgrcv,msgsnd, semop
IPC_RMID	suppression de l'objet	msgctl, shmctl, semctl
IPC_STAT	consultation des caractéristiques	msgctl, shmctl, semctl
IPC_SET	modification des caractéristiques	msgctl, shmctl, semctl

6.3.2 - La structure ipc_perm

Les droits d'accès aux objets sont construits sur le même principe que ceux des fichiers : trois groupes d'utilisateurs (propriétaire/groupe propriétaire/autres) et deux types d'accès (lecture/écriture).

```
struct ipc_perm
{
  uid_t           uid ; /* identification du propiétaire    */
  gid_t           gid ; /* identification du groupe */
  unsigned short  mode ;      /* droits d'accès */
  unsigned short  seq ; /* nb utilisations entrée */
  key_t                 key ; /* clé */
}
```

6.4 - La gestion des clés

La primitive `ftok` construit une clé à partir d'une référence de fichier existant et d'un nombre entier : attention si le fichier est déplacé, la clé sera modifiée ! ! !

```
#include <sys/ipc.h>
key_t ftok(const char *référence,int nombre) ;
```

<u>Exemple</u>
La référence de fichier et le nombre entier sont passés en arguments, sur la ligne de commande.

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>

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

if(argc != 3) {
   fprintf(stderr,"%s: syntaxe:%s <ref_cle>
   <nbr_cle>\n",argv[0],argv[0]);
   exit(1);
}
if((cle = ftok(argv[1],atoi(argv[2]))) == -1) {
   fprintf(stderr,"Probleme de cle\n");
   exit(2);
}}
```

6.5 - Les commandes IPC System V

En principe, les applications doivent supprimer les objets qu'elles n'utilisent plus en appelant les primitives appropriées (`msgctl/shmctl/semctl`) avec l'option `IPC_RMID` mais il est néanmoins prévu de supprimer ces objets sur la ligne de commandes avec `ipcrm -q <msqid> -m <shmid> -s <semid>`.
La commande `ipcs` permet la consultation des trois tables d'I.P.C. du système.

6.6 - Les sémaphores

6.6.1 - Principe initial

Un sémaphore est un entier, prenant les deux valeurs 0 et 1, auquel est associé deux opérations atomiques :

- **Opération P** : abaisser le sémaphore
  Si l'entier est à 1, il est décrémenté.
  Si l'entier est à 0, le processus est mis en sommeil.
- **Opération V** : lever le sémaphore
  L'entier est mis à 1.
  Si l'entier était à 0 avant l'opération V, les processus endormis, en attente de son incrémentation, sont réveillés.

Les sémaphores constituent la solution proposée par Dijkstra pour résoudre le problème de **l'exclusion mutuelle** : plusieurs processus jouant le jeu pourront ainsi s'assurer l'usage exclusif d'une ressource particulière en utilisant un sémaphore associé à celle-ci.

Exemple
Deux processus (père/fils) s'exécutent simultanément et ne doivent pas entrer en section critique en même temps (exclusion mutuelle).
Le processus père crée un sémaphore qu'il initialise à 1 (sémaphore levé) puis crée le processus fils. Les deux processus -père et fils- qui sont des processus concurrents, entreront alors en section critique en respectant tous deux l'algorithme suivant :

1. Opération P sur le sémaphore
2. Entrée en section critique
3. Opération V sur le sémaphore

6.6.2 - Principe des sémaphores Unix

Le principe des sémaphores élaboré par Dijkstra a été repris et amélioré : un sémaphore est un entier naturel **S** auquel est associé trois opérations particulières :

- **Opération P_n(S)**
  Si S≥n, S <- S-n
  Si S<n, mise en attente du processus
- **Opération V_m(S)**
  S <- S+m et réveil des processus en attente de l'augmentation de la valeur de ce sémaphore
- **Opération Z(S)**
  Mise en attente du processus jusqu'à ce que S=0

L'implémentation System V des sémaphores est plus complète encore puisque les différentes primitives utilisées manipulent non pas un sémaphore unique mais des ensembles de sémaphores.

On peut donc résoudre le problème de l'acquisition simultanée d'exemplaires multiples de plusieurs ressources différentes ! ! !

6.6.3 - La table des sémaphores

Chaque ensemble de sémaphores possède un **semid** (semaphore identification) qui référence une entrée dans cette table.
Chaque entrée a une structure *semid_ds* (définie dans le fichier `<sys/sem.h>`) :

```
struct semid_ds
{
struct ipc_perm sem_perm ;    /* structure décrivant les droits   */
struct __sem *sem_base ;      /* pointeur sur le premier sémaphore      */
time_t sem_otime ;            /* date de la dernière opération PVZ      */
time_t sem_ctime ;            /* date de la dernière opération semctl*/
unsigned short sem_nsems ;    /* nombre de sémaphores de l'ens.   */
}
```

La structure *ipc_perm* (définie dans le fichier `<sys/ipc.h>`) est commune à tous les mécanismes I.P.C. System V .

6.6.4 - Les commandes

Information sur les sémaphores : `ipcs -s`
Destruction d'un ensemble de sémaphores : `ipcrm -s <semid>`

6.6.5 - La primitive semget

Elle permet, soit de créer un nouvel ensemble de sémaphores, soit d'accéder à un ensemble déjà existant. Dans les deux cas, la primitive renvoie un nombre entier qui est le **semid** de l'ensemble de sémaphores ou –1 en cas d'échec.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semget(key_t clef, int nb, int drap) ;
```

La clef d'accès à l'ensemble de sémaphores est passée en premier paramètre.
Cas général :
Elle est calculée avec la fonction **ftok** .
Cas particulier :
Elle est égale à la macro-constante **IPC_PRIVATE** : le **semid** du nouvel ensemble ne pourra pas être demandé ultérieurement au système et par conséquent, seule la filiation du processus créateur aura connaissance (par héritage) de ce **semid** et sera en mesure d'accéder à l'ensemble.
Néanmoins un processus connaissant cette identification pourra accéder quand même à l'ensemble si les droits d'accès sont suffisants.
Dans ce cas le troisième paramètre ne sert qu'à préciser les droits d'accès.

Le second paramètre est le nombre de sémaphores de l'ensemble : ces sémaphores seront numérotés de 0 à *nb*-1.

Le troisième paramètre règle le comportement de la primitive :

création d'un ensemble :	**IPC_CREAT| *0xyz***
création exclusive :	**IPC_CREAT|IPC_EXCL|*0xyz***
recherche d'un ensemble	**IPC_ALLOC**

6.6.6 - La primitive semctl

Cette primitive permet de consulter/modifier la structure **semid_ds** mais aussi d'initialiser les sémaphores, de consulter leurs valeurs et d'autres données associées à un ensemble de sémaphores.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semctl(int semid, int semnum, int option,. ? ? ? arg.) ;
```

Le second champ est le numéro du sémaphore dans l'ensemble, le troisième précise le type de l'opération de contrôle réalisée et le quatrième est constitué de données supplémentaires adaptées à l'option choisie.

Valeur de option	Interprét. de semnum	Interprétation de arg	Valeur de retour et effet
GETNCNT	numéro de sém.		nombre de processus en attente d'augmentation du sémaphore
GETZCNT	numéro de sém.		nombre de processus en attente de nullité du sémaphore
GETVAL	numéro de sém.		valeur du sémaphore
GETALL	nb de sém.	tableau d'entiers courts non signés	le tableau contient alors les valeurs des premiers sémaphores
GETPID	numéro de sém.		PID du dernier processus ayant réalisé une opération PVZ
SETVAL	numéro de sém.	entier	initialisation du sémaphore
SETALL	nb de sém.	tableau d'entiers courts non signés	le tableau va servir à initialiser les premiers sémaphores
IPC_RMID			suppression de l'entrée dans la table des sémaphores
IPC_STAT		pointeur sur struct semid_ds	extraction de l'entrée dans la table des sémaphores
IPC_SET		pointeur sur struct semid_ds	modification de l'entrée dans la table des sémaphores

6.6.7 - La primitive semop

La structure **sembuf** correspond à une opération (P/V/Z) sur un sémaphore.

```
struct sembuf
{
 unsigned short int     sem_num ; /* numéro de sémaph. ds l'ens   */
 short                  sem_op ;  /* opération sur le sémaphore   */
 short                  sem_flg ; /* option                       */
}
```

Le signe de **sem_op** détermine l'opération :

- Signe négatif : opération Pn
- Signe positif : opération Vn
- Valeur nulle : opération Z.

La primitive **semop** permet la réalisation atomique d'un ensemble d'opérations P, V et Z sur un ensemble de sémaphores : si une des opérations ne peut être réalisée sur le champ, toutes les opérations déjà effectuées par la primitive sont annulées. Chaque opération peut être rendue individuellement non bloquante en positionnant l'indicateur **IPC_NOWAIT** dans le champ **sem_flg .** En positionnant l'autre indicateur **SEM_UNDO**, on annulera l'effet de cette opération sur le sémaphore concerné à la mort du processus ayant réalisé l'opération. *semop* renvoie –1 en cas d'échec.

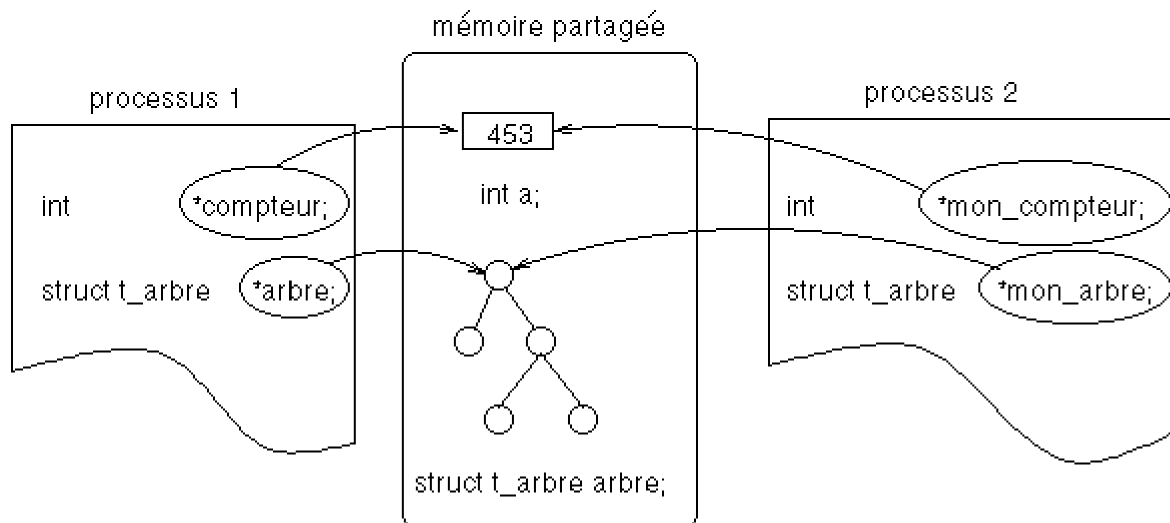
```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *tab_op, int nb_op) ;
```

Le troisième paramètre précise le nombre d'opérations à réaliser.
Le second est l'adresse de la première opération.

6.7 - Segments de mémoire partagée

6.7.1 - Le principe

RENDRE COMMUN À DES PROCESSUS DISTINCTS UNE ZONE MEMOIRE DÉFINIE DYNAMIQUEMENT PAR L'UN D'ENTRE EUX.



6.7.2 - La table des segments

Chaque segment possède un *shmid* (Shared Memory Identification) qui référence une entrée dans cette table. Chaque entrée a une structure *shmid_ds* définie dans <*sys/shm.h*> qui contient notamment les champs :

- *Shm_perm* : structure ipc_perm décrivant les permissions.
- *Shm_segsz* : taille du segment.
- *Shm_lpid* : pid du dernier processus utilisateur.
- *Shm_cpid* : pid du processus créateur du segment.
- *Shm_natch* : nombre d'attachements.
- *Shm_atime* : date du dernier attachement.
- *Shm_dtime* : date du dernier détachement.
- *Shm_ctime* : date de la dernière opération de contrôle.

La structure *ipc_perm* définie dans <*sys/ipc.h*>, commune à tous les IPC System V, comporte en particulier un champ mode dont certains bits jouent un rôle particulier :

- 6 bits règlent les droits d'accès au segment, exactement comme pour un fichier.
- Le bit SHM_DEST précise si le segment doit être supprimé lors du dernier détachement. Ce bit sera automatiquement levé lors d'une demande de suppression du segment par la commande *ipcrm* par la primitive *shmtl* utilisée avec la constante IPC_RMID.

6.7.3 - Les commandes

Informations sur les segments : `ipcs -m`.
Suppression d'un segment donné : `ipcrm -m <shmid>.`

6.7.4 - Création d'un segment et obtention d'un shmid

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmget(key_t key, int size, int shm_flag) ;
```

Créer ou retrouver un segment de mémoire partagée. En cas de succès, renvoie d'un *shmid* (entier > 0), sinon renvoi de –1 et *errno*…

Le paramètre *key* est une clé d'accès au segment. Pour réserver le segment au processus créateur et à sa descendance, on lui donne la valeur IPC_PRIVATE.

Le paramètre *size* est la taille du segment.

Le paramètre *shm_flag* règle les droits d'accès et certains indicateurs de création et de recherche :

- IPC_CREAT : création d'un IPC s'il n'existe pas.
- IPC_ALLOC : recherche d'un IPC supposé exister.
- IPC_EXCL : erreur si l'IPC existe déjà.
- 0xyz : codage octal des droits d'accès.

Ex1 : IPC_CREAT|IPC_EXCL|0666 : création d'un s.m.p. avec *rw*, avec échec si le segment existe déjà.

Ex2 : IPC_ALLOC : recherche de segment existant avec retour du *shmid* si existence et retour de –1 sinon.

6.7.5 - Opérations de contrôle d'un segment

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmctl(int shmid, int cmd, struct shmid_ds *buf) ;
```

Consulter, modifier les caractéristiques d'un segment, ou même le supprimer. Retourne 0 si succès, -1 sinon.

Le paramètre *shmid* est un descripteur de segment.

Le paramètre *cmd* précise l'opération de contrôle à réaliser :

- IPC_STAT : Consultation.
- IPC_SET : Modification.
- IPC_RMID : Suppression.

Le paramètre *buf* est l'adresse d'un objet de structure *shmid_ds* déclaré par l'utilisateur.

Ex :

```
struct shmid_ds brou
  …
  shmctl(id, IPC_STAT, &brou) ;
  brou.shm_perm.mode=0420 ;
  shmctl(id, IPC_SET, &brou) ;
  sleep(1000) ;
  shmctl(id, IPC_RMID, (struct shmid_ds *)0) ;
```

6.7.6 - Attachement à un segment

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
void* shmat(int shmid, void *buf, int shmflag) ;
```

Rendre visible un segment de mémoire partagée dans l'espace d'adressage d'un processus. Retourne de l'adresse d'attachement, sinon –1.

Le paramètre *shmid* est le descripteur du segment.
Le paramètre *buf* est l'adresse d'attachement ; il est conseillé de laisser le système de choisir le point de greffe en passant zéro comme argument.
Le paramètre *shmflag* est en général égal à zéro, ce qui signifie que le segment est attaché en lecture/écriture. On peut interdire toute écriture en passant SHM_RDONLY comme argument.

6.7.7 - Le détachement d'un segment

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>
int shmdt(void *buf) ;
```

Supprimer la visibilité d'un processus sur un segment. Retourne zéro ou -1. Le paramètre *buf* est l'adresse d'attachement du segment, c'est-à-dire la valeur renvoyée par la primitive *shmat*.

6.7.8 - Détachement et suppression

Le détachement n'affecte que la visibilité du segment depuis un processus et pas du tout son existence. Un segment pouvant être attaché à plusieurs processus, une suppression n'aura vraiment lieu que lorsque le champ *shm_nattch* de la structure *shmid_ds* décrivant le segment sera à 0. L'opération de contrôle SHM_RMID réalisé avec la primitive *shmctl* consiste donc à lever le drapeau SHM_DEST qui signifie au noyau de supprimer le segment dès que possible.

Exemple.
Le listing ci-dessous est celui d'un programme écrivain. Les commentaires donnent les transformations à effectuer pour obtenir un programme lecteur.

```
#include<sys/types.h>
#include<sys/ipc.h>
#include<sys/shm.h>

int  main()
{ key_t key ;
  int flag, id ;
  char *buf ;

flag=IPC_CREAT|0600 ;              /* flag=IPC_ALLOC ;*/
key=52 ;
if ((id=shmget(key,512,flag))<0) exit(1) ;
if((buf=shmat(id,0,0))<0) exit(2) ;
strcpy(buf,"croa ") ;              /*printf(" Contenu : %s ", buf);*/
shmdt(buf) ;                       /*shmdt(buf); exit(0);*/
sleep(100) ;
shmctl(id,IPC_RMID,0) ;
exit(0) ;}
```

7 - Les threads

7.1 - Introduction

Dans les versions classiques du système UNIX, l'un des concepts fondamentaux est celui de processus. Un processus recouvre à la fois un ensemble de ressources (essentiellement un espace d'adressage) et une unité d'exécution unique utilisant ces ressources : on y trouve le programme que le processus exécute, des données systèmes (priorité, propriétaires, fichiers ouverts, ...) et l'ensemble des données que le programme manipule, cet ensemble pouvant être lui-même divisé entre les données proprement dites (correspondant aux données statiques et au tas d'un programme C) et les données de la pile d’exécution (correspondant aux variables automatiques). Si plusieurs processus peuvent partager physiquement le même programme, ce que l'on désigne par le terme réentrance, chaque processus dispose de ses propres données physiques, même si par le mécanisme de mémoire partagée, il est possible à plusieurs processus de partager physiquement certaines pages de données après qu'ils en aient fait la demande effective.
Avec l'émergence des machines multiprocesseurs et le développement d'applications selon le modèle serveur/client dans lequel le serveur se démultiplie pour mieux servir ses clients, le concept de processus tel qu'il était ainsi défini a rapidement montré ses limites et son inéquation à exploiter cet aspect multiprocesseurs, essentiellement en raison du coût élevé de la création d'un nouveau processus. Cela a tout d'abord conduit à l'intégration, par certains constructeurs, dans des versions UNIX existantes, de mécanismes permettant de séparer les deux aspects des processus, a savoir d'un coté le processus comme unité d'encapsulation des ressources utilisées et d'un autre les unités d’exécution sur cet ensemble : cela a été en particulier le cas de SUN avec le concept de processus léger (lightprocess). Parallèlement, cette idée a fait son chemin puisqu'elle constitue l'une des abstractions de base que l'on retrouve dans les différents exemples de micro-noyaux qui ont été développés.
Comme nous nous placerons dans ce chapitre dans un contexte UNIX où les activités seront créées dans le cadre d'un processus UNIX, nous adopterons les termes de processus pour désigner l'ensemble des ressources et celui d'activités pour désigner les unités d’exécution(les threads).
Même si fondamentalement, l'utilisation de ce mécanisme est particulièrement adaptée d'une part à un contexte temps réel et d'autre part à un environnement multiprocesseurs, son utilisation peut s'avérer bénéfique dans le cas de systèmes monoprocesseurs par l’amélioration qu'elle apporte du point de vue des performances en matière de création de processus (ou plutôt d'activités).
L’intérêt d’intégrer un tel mécanisme dans les systèmes ouverts a conduit à l'élaboration d'une interface standard dans le cadre de POSIX (standard POSIX 1003.4a). Cependant, si dans les micro-noyaux, les activités constituent des objets de base au même titre que les processus, dans la plupart des systèmes ouverts qui proposent actuellement l'interface correspondante, les différentes fonctions constituent une bibliothèque.
Il doit être clair que l'introduction de tels mécanismes, par lesquels des données physiques sont massivement partagées par différentes entités, suppose la disponibilité de mécanismes de synchronisation permettant de régler les conflits provoqués par les accès concurrentiels à ces données par différentes activités à l'intérieur d'une même tâche.

7.1.1 - Processus UNIX et threads

A un processus UNIX est associé un ensemble d'informations constituant différentes structures de données permettant au système d'en assurer le contrôle. Ces différentes informations peuvent être placées dans deux catégories différentes selon qu'elles contribuent au contrôle des ressources du processus ou a celui de son exécution.
Dans la première catégorie, on trouve en particulier : le masque de création des fichiers (umask), les propriétaires et groupes propriétaires réel et effectif, les descripteurs vers les fichiers ouverts, le répertoire de travail et les informations sur l'implantation en mémoire des données du programme.
Dans la seconde catégorie, on trouve : les informations nécessaires à l'ordonnanceur du système (scheduler), la valeur des registres (et en particulier le compteur ordinal), la pile d'exécution et les informations relatives aux signaux. Ainsi les informations de la première catégorie seront partagées par toutes les activités s'exécutant dans le même contexte et chacune disposera de caractéristiques de la seconde catégorie qui lui seront propres.
Dans ce contexte, un processus UNIX correspond à un processus (ou une tache) où ne s'exécute qu'une seule activité.

7.1.2 - Remarque

Nous nous proposons de décrire le fonctionnement général du mécanisme des Posix threads (P-threads) afin de rester le plus portable possible. Pour cette raison, les types et les fonctions propres à l'utilisation des threads commencent par pthread.

7.1.3 - Compilation pour les threads

Sous Linux : cc -D_REENTRANT -lpthread -o <pgr> <pgr.c>

7.2 - Les attributs d'un activité

7.2.1 - Identification d'une activité

Toute activité possède une identification (TID) de type **pthread_t** (qui correspond généralement au type entier) dans le cadre du processus dans le contexte duquel elle s'exécute. Un appel à la primitive :

```
#include <unistd.h>
pid_t getpid(void);
```

permet à une activité d'obtenir l'identité du processus UNIX auquel elle appartient. Un appel à la fonction :

```
#include <pthread.h>
pthread_t pthread_self(void);
```

renvoie l'identité de l'activité. Dans un contexte où l'identification d'une activité est un nombre entier, un processus UNIX correspond à une activité initiale dont le numéro est 1 et les nouvelles activités (ou activités annexes) créées dans le cadre d'un processus auront comme identités successives 2,3,… Dans ce même

contexte, une activité de numéro *tid* d'un processus de numéro *pid* pourra être désignée sous la forme *pid.tid* ce qui permettra, si le système le permet, d'adresser un signal à une activité particulière d'un processus. Ainsi par exemple, la commande : **kill –9 12345.3** enverrait le signal 9 à l'activité 3 du processus UNIX de numéro 12345.

7.2.2 - Terminaison d'un activité

Disons dès maintenant que la terminaison de l'activité initiale d'un processus correspond à celle du processus et entraîne donc la libération des ressources correspondant au processus : ceci a pour conséquence d'empêcher la poursuite de l'exécution des autres activités encore actives dans le cadre du processus.
Par ailleurs, un appel à la fonction :

```
#include <pthread.h>
int pthread_equal(pthread_t tid_1, pthread_t tid_2);
```

permet de tester de manière portable (c'est à dire sans hypothèse sur la nature du type thread_t l'égalité des deux identités tid_1 et tid_2 : la valeur renvoyée par la fonction est non nulle si les identités sont égales et nulle sinon.

7.3 - Création et terminaison des activités

7.3.1 - Création d'une activité : pthread_create

Un appel à la fonction :

```
#include <pthread.h>
int pthread_create(     pthread_t *p_tid,
                              pthread_attr_t attr,
                              void *(*fonction)(void *arg),
                              void *arg);
```

correspond à la demande de création et d'activation d'une nouvelle activité.
La valeur 0 est renvoyée en cas de réussite et la valeur -1 en cas d'échec. Dans ce cas, la variable **errno** permet de connaître la nature de l'erreur rencontrée (EAGAIN si le nombre maximal d'activités est atteint et ENOMEM si l'espace mémoire est insuffisant).

Le rôle et l'interprétation des différents paramètres sont les suivants :
- Au retour d'un appel réussi, ***p_tid** a pour valeur l'identité de l'activité nouvellement créée ;
- le paramètre **attr** définit la valeur des attributs de l'activité. la valeur par défaut de nom symbolique **pthread_attr_default** était généralement utilisée, dans les dernières implémentation des thread de POSIX cette variable n'est plus utilisée, on la positionnera donc à la valeur **NULL**;
- le paramètre **fonction** correspond à la fonction exécutée par l'activité après sa création : il s'agit donc de son point d'entrée (comme la fonction main est la fonction d'un programme appelée au lancement d'une commande). Ce paramètre est un pointeur sur une fonction ayant un argument de type pointeur (le prototype spécifie le

pointeur générique void *) et ayant comme valeur un pointeur (le prototype spécifie également le pointeur générique). Un retour de cette fonction correspondra à la terminaison de l'activité correspondante;
- le paramètre **arg** correspond au paramètre transmis à la fonction précédente appelée au lancement de l'activité.

7.3.2 - Terminaison d'une activité

7.3.2.1 - Terminaison par exit ou _exit

Si une activité quelconque (c'est à dire initiale ou annexe) d'un processus exécute un appel à l'une de ces deux fonctions, toutes les activités dans le processus se terminent.

7.3.2.2 - La fonction pthread_exit

Un appel à la fonction :

```
#include <pthread.h>
int pthread_exit(int *p_status);
```

termine l'activité appelante avec une valeur de retour égale à ***p_status**. Ce code de retour est accessible aux autres activités dans le même processus au travers de la fonction **pthread_join**. Rappelons que la sortie de la fonction appelée au lancement d'une activité entraîne automatiquement un appel à la fonction **pthread_exit**.
Dans le cas où c'est l'activité initiale d'un processus qui réalise un appel à cette fonction, les ressources du processus sont libérées, ce qui entraîne des erreurs à l'exécution des autres activités du processus et donc leur terminaison.
Il est important de noter que lorsqu'une activité se termine, elle ne disparaît pas, contrairement à ce qui se passe avec les processus UNIX où un processus zombi (c'est à dire terminé) disparaît des que son père a réalisé un appel wait ou waitpid pour acquérir le code de retour correspondant à sa terminaison. Une activité peut ainsi être attendue (par un appel à pthread_join) par plusieurs de ses congénères dans le même processus. Les ressources qui ont été allouées à l'activité ne sont pas restituées. En particulier, son numéro et le segment d'adresses correspondant à sa pile d'exécution sont pas récupérés. La destruction (entraînant la restitution des différentes ressources) doit être explicitement demandée au moyen d'un appel à la fonction :

```
#include <pthread.h>
int pthread_detach(pthread_t p_tid);
```

Un tel appel n'interrompt pas une activité en cours d'exécution : elle indique qu'à la terminaison de l'activité ses ressources devront être restituées au processus auquel elle appartient.

7.3.2.3 - Demande d'abandon d'une activité

Une activité d'un processus peut demander l'abandon d'une autre activité du même processus par une appel à la fonction :

```
#include <pthread.h>
int pthread_cancel(pthread_t tid);
```

Le traitement d'une requête d'abandon par une activité dépend du type de comportement qu'elle a sélectionné. Ce comportement est déterminé par la valeur de deux indicateurs particuliers qui peuvent être positionnés ou non :
- Un indicateur général permet, lorsqu'il n'est pas positionné, de différer la prise en compte des requêtes qui deviennent pendantes jusqu'à ce qu'il soit de nouveau positionné (il s'agit d'un mécanisme analogue à celui qui permet de bloquer la délivrance d'un signal). Le changement d'état de l'indicateur général est réalisé par un appel à la fonction :

```
#include <pthread.h>
int pthread_setcancel(pthread etat);
```

Les valeurs reconnues du paramètre état sont **CANCEL_ON** et **CANCEL_OFF** et la valeur retournée par la fonction est celle de l'état antérieur. Par défaut, c'est à dire à la création d'une activité, l'indicateur général est positionné;

- Un indicateur d'asynchronisme, qui par défaut est non positionné et qui peut être modifié par un appel à la fonction :

```
#include <pthread.h>
int pthread_setasynccancel(pthread_t etat);
```

Le positionnement de cet indicateur n'est significatif que si l'indicateur général l'est également. Son effet est alors de provoquer la prise en compte immédiate d'une requête d'abandon formulée par une autre activité. S'il n'est pas positionné, la prise en compte d'une telle requête est réalisée au premier point de contrôle rencontré. Un tel point peut tout d'abord être explicitement placé dans le corps de l'activité et correspond à un appel à la fonction :

```
#include <pthread.h>
void pthread_test_cancel(void);
```

Par ailleurs un point de contrôle est implicitement associé à un appel de chacune des fonctions **pthread_join**, **pthread_setcancel**, **pthread_setasynccancel**, **pthread_cond_wait** et **pthread_cond_timedwait** (pour ces deux dernières fonctions, après la libération du sémaphore d'exclusion mutuelle correspondant à l'appel).
Dans tous les cas, si aucune requête d'abandon de l'activité n'a été formulée ou si l'indicateur général n'est pas positionné, l'activité se poursuit normalement.

7.3.2.4 - La pile de nettoyage d'une activité

A chaque activité est associée une pile d'appels de fonctions à réaliser à sa terminaison provoquée par un appel à **pthread_exit** ou suite à une requête d'abandon. Cela peut permettre par exemple la libération d'espace mémoire alloué dynamiquement lors de l'exécution de l'activité ou l'impression d'un message spécifique.
La gestion de cette pile est tout d'abord réalisée par la fonction d'empilement:

```
#include <pthread.h>
int pthread_cleanup_push(     void (*fonction)(void *arg),
      void *arg);
```

qui permet d'empiler un appel à la fonction ***fonction** avec la valeur du paramètre spécifié. Par ailleurs, la fonction de dépilement :

```
#include <pthread.h>
void pthread_cleanup_pop(int executer);
```

permet de dépiler l'appel qui se trouve au sommet de la pile. Si l'argument à exécuter est non nul, l'appel qui est dépilé est effectivement réalisé et si sa valeur est nulle, il ne l'est pas.

7.4 - Synchronisation des activités

7.4.1 - Introduction

Le concept d'activités, s'il facilite la communication des entités contribuant à une même tâche puisque les différentes activités d'une même tâche partagent par définition leurs données, nécessite de prendre certaines précaution afin d'assurer une utilisation cohérente de ces données. On se trouve confronté ici au même type de problèmes que lors de l'utilisation d'un même segment de mémoire partagée par différents processus. L'implantation des P-Threads intègrent tout d'abord un mécanisme (mutex ou sémaphore d'exclusion mutuelle) permettant de résoudre le problème de l'exclusion mutuelle d'activités. Un certain nombre d'autres mécanismes sont généralement fournis pour permettre la synchronisation de différentes activités à l’intérieur d'une même tache. On retrouve tout d'abord la possibilité de permettre à une activité d'attendre la terminaison d'une autre. Le second mécanisme qui est fourni permet à des activités d'une même tache de se synchroniser sur l’occurrence d’événements : les objets intervenant dans ce mécanisme sont appelés variables de conditions.

7.4.2 - Synchronisation sur terminaison d'une activité : pthread_join

Il est tout d'abord possible à une ou plusieurs activités d'attendre la terminaison d'une activité particulière et de récupérer une valeur de retour correspondant à sa terminaison. Un telle attente est réalisée par un appel à la fonction :

```
#include <pthread.h>
int pthread_join(pthread_t tid, int **status);
```

L'activité appelante suspend alors son exécution jusqu'à ce que l'activité d'identité **tid** dans le même processus exécute un appel à la fonction **pthread_exit** ou qu'une autre activité en ait demandé l'abandon par un appel à la fonction **pthread_cancel**. Si l'activité est déjà terminée, le retour est immédiat. Au retour de la fonction, ****status** est égale à la valeur de retour de l'activité.
La valeur de retour est 0 en cas de réussite de l'appel et à -1 en cas d'erreur :

EINVAL : identité d'activité incorrecte;
ESRCH : activité inexistante;
EDEADLK : l'attente de l'activité spécifiée conduit à un deadlock.

7.4.3 - LES MUTEX

Un mutex permet l'exclusion mutuelle entre threads.
Un mutex est de type pthread_mutex_t et a un ensemble d'attributs associé de type pthread_mutexattr_t, que nous ne détaillerons pas ici (ils ne sont pas portables). Nous utiliserons l'ensemble par défaut qui est obtenu en passant NULL aux fonctions qui demandent un pointeur sur cet ensemble .

7.4.3.1 - La primitive de création d'un mutex

```
#include <pthread.h>
int pthread_mutex_init(pthread_mutex_t *mutex_pt, pthread_mutexattr_t *attr) ;
```

Permet de créer un nouveau mutex avec un certain nombre d'attributs.
Paramètres :
- Avant appel : mutex_pt : pointe sur une zone réservée pour contenir le mutex
attr : ensemble d'attributs à affecter au mutex
- Après appel : mutex_pt : pointe sur le mutex nouvellement créé.
Retour : 0 : en cas de succès
!=0 : en cas d'échec et positionnement de la variable errno (EINVAL, ENOMEM, EAGAIN)

7.4.3.2 - La primitive P pour un mutex

L'appel bloquant

```
#include <pthread.h>
int pthread_mutex_lock(pthread_mutex_t *mutex_pt);
```

Permet, de façon atomique, de réserver un mutex ou d'attendre tant que le mutex est réservé par une autre activité.

Paramètre : mutex_pt : pointe sur le mutex à réserver
Retour : 0 : en cas de succès
!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL, EDEADLCK)

L'appel non bloquant

```
#include <pthread.h>
int pthread_mutex_trylock(pthread_mutex_t *mutex_pt);
```

Permet, de façon atomique, de réserver un mutex ou de renvoyer une valeur particulière si le mutex est réservé par une autre activité.

Paramètre : mutex_pt : pointe sur le mutex à réserver

Retour : 1: en cas de réservation

0 : en cas d'échec de la réservation

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL)

7.4.3.3 - La primitive V pour un mutex

```
#include <pthread.h>
int pthread_mutex_unlock(pthread_mutex_t *mutex_pt);
```

Permet de libérer un mutex et de débloquer les activités en attente sur ce mutex.

Paramètre : mutex_pt : pointe sur le mutex à libérer

Retour : 0 : en cas de succès

!=0 et errno = EBUSY : mutex déjà pris

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL)

7.4.3.4 - La primitive de destruction d'un mutex

```
#include <pthread.h>
int pthread_mutex_destroy(pthread_mutex_t *mutex_pt);
```

Permet de détruire un mutex.

Paramètre : mutex_pt : pointe sur le mutex à détruire

Retour : 0 : en cas de succès

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL, EBUSY)

7.4.4 - Les conditions

Une condition est un mécanisme permettant de synchroniser plusieurs activités à l'intérieur d'une section critique. En effet lors de l'attente
sur une condition un mutex est libéré de façon automatique et réservé lors de la sortie d'attente.

Une condition est de type pthread_cond_t, et à un certain nombre d'attributs que nous ne détaillerons pas ici (ils ne sont portables).
Nous utiliserons la valeur par défaut qui est obtenu en passant NULL aux fonctions qui attendent un pointeur sur ce type.

7.4.4.1 - La primitive de création d'une condition

```
#include <pthread.h>
int pthread_cond_init(pthread_cond_t * cond_pt, pthread_condattr_t *attr);
```

Permet de créer une nouvelle condition.
Paramètres :

- Avant appel cond_pt : pointeur sur la zone réservée pour recevoir la condition
  attr : attributs à donner à la condition lors de sa création
- Après appel cond_pt : pointe sur la nouvelle condition

Retour : 0 : en cas de succès
!=0 : en cas d'erreur et positionnement de la variable errno (EAGAIN, EINVAL, ENOMEM)

7.4.4.2 - Les primitives d'attente sur une condition

L'attente non bornée

```
#include <pthread.h>
int pthread_cond_wait(pthread_cond_t *cond_pt, pthread_mutex_t * mutex_pt);
```

Appel bloquant qui attend une condition (envoyée par exemple par la primitive pthread_cond_signal). Pendant l'attente le mutex mutex_pt est libéré et une fois la condition reçue le thread courante essaye de reprendre le mutex (elle peut donc rester bloquée en essayant de le reprendre).

Paramètres :
cond_pt : pointeur sur la condition à attendre
mutex_pt : pointeur sur le mutex à libérer pendant l'attente (ceci suppose que ce mutex est réservé)
Retour : 0 : en cas de succès
!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL, EDEADLK)

L'attente bornée

```
#include <pthread.h>
int pthread_cond_timewait(pthread_cond_t *cond_pt, pthread_mutex_t * mutex_pt, struct
timespec *abstime);
```

Permet d'attendre une condition en libérant le mutex de la section critique pendant un temps borné.

Paramètres :
cond_pt : pointeur sur la condition à attendre
mutex_pt : pointeur sur le mutex à libérer pendant l'attente (ceci suppose que ce mutex est réservé)
abstime : pointeur sur une structure contenant le temps d'attente :

```
struct timespec {
      unsigned long tv_sec; /* secondes */
      long tv_nsec; /* nanosecondes */
};
```

Retour :

0 : en cas de succès

!=0 : en cas d'expiration avec errno==ETIMEDOUT

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL, EDEADLK)

7.4.4.3 - Les primitives de signal d'une condition

Réveil d'un thread

```
#include <pthread.h>
int pthread_cond_signal(pthread_cond_t *cond_pt);
```

Permet de réveiller un thread en attente sur une condition.

Paramètre : cond_pt : pointeur sur la condition à signaler

Retour : 0 : en cas de succès

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL)

Réveil de toutes les threads

```
#include <pthread.h>
int pthread_cond_broadcast(pthread_cond_t *cond_pt);
```

Permet de réveiller toutes les threads en attente sur une condition.

Paramètre : cond_pt : pointeur sur la condition à signaler

Retour : 0 : en cas de succès

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL)

Remarque :

Contrairement aux signaux, si une condition est signalée alors qu'aucun thread n'est en attente dessus, l'information "la condition a été signalée" est perdue.

7.4.4.4 - La primitive de destruction d'une condition

```
#include <pthread.h>
int pthread_cond_destroy(pthread_cond_t *cond_pt);
```

Permet de détruire les ressources associées à une condition

Paramètre : cond_pt : pointeur sur la condition à détruire

Retour : 0 : en cas de succès

!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL, EBUSY)

7.4.5 - Autres mécanisme associés aux threads

7.4.5.1 - Demande explicite de libération du processeur

Un thread peut demander explicitement au cours de son exécution de rendre la main grâce à la primitive suivante :

```
#include <pthread.h>
void sched_yield();
```

7.4.5.2 - Appel unique à une fonction

Avec les threads la norme POSIX propose un mécanisme assurant qu'une fonction est exécutée seulement une fois. Pour cela on utilise une variable de type pthread_once_t, initialisée à la valeur PTHREAD_ONCE_INIT et la primitive est la suivante :

```
#include <pthread.h>
int pthread_once(pthread_once_t *once_block, void (*init_routine)());
```

Paramètres :	once_block : pointeur sur variable initialisée à la valeur pthread_once_init
	init_routine : pointeur sur la fonction à n'appeler qu'une fois
Retour :	0 : en cas de succès
	!=0 : en cas d'erreur et positionnement de la variable errno (EINVAL)

Exemple :

```
#include <pthread.h>

static pthread_once_t make_mutex = PTHREAD_ONCE_INIT;
static pthread_mutex_t mutex;

void mutex_init ()
{
…
      pthread_mutex_init(&mutex,pthread_mutexattr_default);
…
}
...
pthread_once (&make_mutex, mutex_init);
...
```

7.5 - Exemple d'utilisation des mutex et des conditions

Dans cet exemple nous proposons une version du problème des trois fumeurs où chaque fumeur attend un produit particulier parmi : papier, tabac, feu (deux fumeurs n'attendant pas le même produit). La seita dépose un produit chaque fois qu'il n'y a plus rien sur la table (condition).

```
/* File : pthread_fumeurs.c */

#include <stdio.h>
#include <pthread.h>

pthread_t pthread_id[3];
pthread_mutex_t mutex[3];
pthread_cond_t cond;
int i;

void * fumeur(void *);

int main(int c, char *v[])
{
  /* initialisation des trois mutex et de la condition */
  for(i=0;i<3;i++)
    pthread_mutex_init(mutex+i,NULL);
  pthread_cond_init(&cond,NULL);

  /* Prise des trois mutex */
  for(i=0;i<3;i++)
    pthread_mutex_lock(mutex+i);

  /* creation des 3 threads fumeurs */
  for(i=0;i<3;i++)
    pthread_create(pthread_id+i,NULL,fumeur,(int *)i);

  /* Liberer 100 fois un mutex particulier
   et attendre sur la condition */
  for(i=0;i<100;i++)
    pthread_cond_wait(&cond,mutex+(i%3));

  for(i=0;i<3;i++)
      pthread_mutex_destroy(mutex+(i%3));

  pthread_cond_destroy(&cond);
  exit(0);
}

void *fumeur(void *num)
{
  for(;;)
    {
      /* attendre sur un mutex particulier avant de fumer */
      pthread_mutex_lock(mutex+(int)num);
      printf("%d fume\n",(int)num);

      /* liberer le mutex */
      pthread_mutex_unlock(mutex+(int)num);

      /* prevenir qui j'ai fini de fumer par une condition */
      pthread_cond_signal(&cond);

     /* Rendre le processeur pour etre sur que c'est le buraliste
         qui a la main et donc qui reprend le semaphore */
      sched_yield();
    }
}
```