Chapitre 14 LES FILES DE MESSAGES

1. GENERALITES

Il s'agit d'une implémentation du concept de boite à lettres. Le but est d'échanger ,de façon asynchrone, des données stockées dans le noyau du système, sous forme de paquets d'octets identifiables. C'est un mode de communication opposé à celui des tubes parce qu'une opération de lecture ne peut extraire que le produit d'une opération d'écriture.

Avec une seule file de message, on peut envoyer des paquets à plusieurs processus (notion de multiplexage).

La structure d'un message est donnée dans **/usr/include/sys/msg.h** par :

```
struct msg_buf
 {
        long mtype;          /* type du message       */
        char mtext [1];      /* contenu du message */
};
```

Elle est inutilisable en l'état (taille 1 !!!). La taille maximale d'un texte de message est définie à la configuration du système.

Les messages sont rangés dans une file de messages, zone de mémoire centrale gérée par le système. Le processus qui a créé une file de messages en est le propriétaire.

Une file de messages est associée à une clé (l'équivalent numérique d'un nom) utilisée pour obtenir l'identificateur de la file de messages **msgid**, fourni par UNIX au processus qui donne la clé.

Tout processus qui a connaissance de l'existence de la file de messages et qui possède des droits d'accès peut réaliser des opérations sur la file :

- création d'une file,
- consultation des caractéristiques d'une file :
    * nombre de messages dans la file,
    * taille de la file,
    * pid du dernier processus ayant écrit dans la file,
    * pid du dernier processus ayant lu (extrait) dans la file,
- envoi d'un message dans la file,
- lecture (extraction) d'un message, en liaison avec son type.

Le fichier **/usr/include/sys/msg.h** contient la définition de la structure **msqid_ds** consultable par la fonction système **ipcs**, et de diverses constantes.

2. CREATION D'UNE FILE DE MESSAGES

La fonction prototypée par :

int msgget (key_t cle, int option)

retourne l'indicateur de la file (ou -1 en cas d'erreur).

key_t : type défini dans /usr/include/sys/ipc.h, équivalent à long

option : combinaison, avec l'opérateur | de droits d'accès et de constantes définies dans /usr/include/sys/msg.h et /usr/include/sys/ipc.h :

MSG_R (0400), MSG_W (0200): permission en lecture, en écriture
IPC_CREAT (0001000) : création d'une file de messages
IPC_EXCL (0002000) : échec si la file existe déjà

cle: si **cle** = **IPC_PRIVATE**, une nouvelle file est créée

sinon si **cle** ne correspond pas à une file existante
si **IPC_CREAT** n'est pas positionné : erreur, retour de -1
sinon, une nouvelle file est créée associée à cette clé avec les droits d'accès spécifiés. Le propriétaire effectif est le propriétaire effectif du processus. Le groupe propriétaire et créateur est le groupe propriétaire effectif du processus.

sinon si IPC_CREAT **et** IPC_EXCL sont positionnés :erreur,retour de -1
sinon, la fonction retourne l'identificateur de la file

La fonction prototypée par :

key_t ftok (char *nom, int id)

dans /usr/include/sys/types.h calcule une clé unique dans tout le système à partir du fichier de chemin **nom** et de **id**.

Exemple : récupération de l'identificateur d'une file de messages
num = msgget (ftok (CHEMIN, cle), 0)

Exemple : création d'une file de messages

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
main ()
{
        key_t cle;
        int flag, num, i;
        printf ("donnez la cle entière associée à la file à créer : ");
        scanf ("%ld", &cle);
        flag = MSG_W | MSG_R | IPC_CREAT;
        /* autre solution : flag = IPC_CREAT | IPC_EXCL | 0666 ; */
        if ((num = msgget (cle, flag)) == -1)
        /* autre solution : if (((num = msgget (ftok (CHEMIN, cle), flag)) == -1)  */
                {
                        fprintf (stderr, "création impossible\n");
                        exit (1);
                }
        printf ("file créée avec l'identificateur  %d\n", num);
}
```

3. CONSULTATION, MODIFICATION DES CARACTERISTIQUES D'UNE FILE DE MESSAGES.

On utilise à cet effet la fonction prototypée par :

int msgctl (int msgid, int cmd, struct msqid_ds *buf)

retourne 0 (ou -1 en cas d'échec)

msgid : identificateur de la file

cmd : définit les opérations à effectuer, selon sa valeur :

IPC_RMID : la file est supprimée
IPC_SET : mise à jour de quelques valeurs de permission (champs de la structure msg_perm associée à msgid : msg_perm.uid, msg_perm.gid et msg_perm.mode)
IPC_STAT : recopie dans buf les informations relatives à la file contenues dans l'objet de type struct msqid_ds, associé à la file.

Exemple :

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

main ()
{
        int num;
        struct msqid_ds buf;
        printf ("numéro de la file à étudier ? ");
        scanf ("%d", &num);
        if (msgctl (num, IPC_STAT, &buf) == -1)
                {  fprintf (stderr, "consultation impossible\n");
                   exit (1);
                }
        printf ("caractéristiques de la file : %d\n", num);
        printf ("nombre de messages : %d\n", buf.msg_qnum);
        printf ("droits d'accès     : %o\n", buf.msg_perm.mode);
        if (msgctl (num, IPC_RMID, &buf) == -1)
        /* ou bien : if (msgctl (num, IPC_RMID, NULL ) == -1)  */
                {  fprintf (stderr, "suppression impossible\n");
                   exit (2);
                }
        printf ("suppression de la file %d réussie\n", num);
}
```

Un exemple intéressant est donné dans A.B. FONTAINE et Ph. HAMMES, p. 359 à 361.

4. ENVOI DE MESSAGE A UNE FILE

La fonction prototypée par :

int msgsnd (int msgid, struct msgtxt *msgp, int msgz, int msgflg)

retourne 0 (ou -1 en cas d'erreur), avec :

msgid : identificateur de la file

La structure msgbuf (inutilisable, cf. p. 1) est remplacée par exemple par :

```
struct msgtxt
{
        long mtype;            /* type du message         */
        char mtext [256];      /* contenu du message      */
};
```

msgz : taille réelle du message

msgflg peut prendre les valeurs :

0 : blocage de la fonction si la file est pleine

IPC_NOWAIT : si la file est pleine, l'appel à la fonction n'est pas bloquant

Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define TAILLE 100        /* taille maximale d'un message */
struct msgtxt
        {
                long mtype;            /* type du message         */
                char mtext [256];  /* contenu du message */
        };

main ()
{
        int i, num;
        long t;
        char ch [TAILLE];
        struct msqid_ds buf;
        struct msgtxt message;
        printf ("numéro de la file à étudier ? ");
        scanf ("%d", &num);
        if (msgctl (num, IPC_STAT, &buf) == -1)
                {  fprintf (stderr, "erreur\n");
                   exit (1);
                }
        printf ("caractéristiques de la file : %d\n", num);
        printf ("nombre de messages avant écriture : %d\n", buf.msg_qnum);
        printf ("pid du dernier processus écrivain : %d\n",buf.msg_lspid);
        printf ("type du message : ");
        scanf ("%ld", &t);
        message.mtype = t;
        printf ("texte du message (%d car. maxi) : ", TAILLE-1);
        gets (ch);
        strcpy (message.mtext, ch);
```

```
        i = msgsnd (num, &message, strlen (ch), ~ IPC_NOWAIT);
        if (i != 0) fprintf (stderr,"message non envoyé\n");
        else    {       printf ("numéro du processus %d\n", getpid ());
                        msgctl (num, IPC_STAT, &buf);
                        printf ("nombre de messages après écriture : %d\n", buf.msg_qnum);
                        printf ("pid du dernier processus écrivain : %d\n", buf.msg_lspid);
                }
}
```

5 . EXTRACTION D'UN MESSAGE D'UNE FILE.

La fonction prototypée par :

int msgrcv (int msgid, struct msgtxt *buf, int msgz, long msgtyp, int msgflg)

retourne la longieur du message extrait (-1 en cas d'erreur), avec :

msgid : identificateur de la file

la structure msgbuf (inutilisable, cf. p.1) est remplacée par :

```
struct msgtxt
{
        long mtype;             /* type du message         */
        char mtext [256];  /* contenu du message */
};
```

msgz : taille maximale d'un message recevable

msgtyp : code du type du message à extraire :
- 0 : le premier message dans la file, quelque soit son type
- > 0 : le premier message de type msgtyp dans la file
- < 0 : le premier message de type 0 ... |msgtyp|

msgflg précise l'action à réaliser si aucun message du type attendu n'est présent dans la file :

si IPC_NOWAIT est positionné, on retourne -1 et l'appel est non bloquant

sinon, la fonction se bloque jusqu'à l'arrivée d'un message satisfaisant ou jusqu'à réception d'un signal par le processus appelant ou jusqu'à disparition de la file.

Exemple :

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#define TAILLE 100          /* taille maximale d'un message */
struct msgtxt
{
        long mtype;             /* type du message                 */
        char mtext [256];       /* contenu du message              */
};

main ()
{
        int i, num;
        long type;
```

```
        struct msqid_ds buf;
        struct msgtxt message;
        printf ("numéro de la file à étudier ? ");
        scanf ("%d", &num);
        if (msgctl (num, IPC_STAT, &buf) == -1)
                {  fprintf (stderr, "erreur\n");
                   exit (1);
                }
        printf ("caractéristiques de la file : %d\n", num);
        printf ("nombre de messages avant lecture : %d\n", buf.msg_qnum);
        printf ("pid du dernier processus lecteur : %d\n", buf.msg_lrpid);
        printf ("type du message : ");
        scanf ("%ld", &type);
        i = msgrcv (num, &message, TAILLE, type, ~ IPC_NOWAIT);
        if (i == -1) fprintf (stderr, "pas de message\n");
        else {          printf ("pid du processus courant %d\n", getpid ());
                        msgctl (num, IPC_STAT, &buf);
                        printf ("nombre de messages après lecture : %d\n", buf.msg_qnum);
                        printf ("pid du dernier processus lecteur : %d\n",buf.msg_lrpid);
                }
}
```

Chapitre 15

LES SEGMENTS DE MEMOIRE PARTAGEE

Un segment de mémoire partagée est une zone d'octets désignée par un pointeur (adresse de base). La communication entre processus par fichiers, tubes ou files de messages suppose d'abord la recopie des données de l'espace d'adressage de l'émetteur vers le noyau, puis la recopie des données du noyau vers l'espace d'adressage du processus destinataire. Pour ces opérations, le processus bascule nécessairement du mode utilisateur en mode noyau.

Au contraire, un segment de mémoire partagée est utilisable par plusieurs processus sans avoir à le recopier dans leurs environnements respectifs.

Un segment de mémoire partagée est identifié par le système grâce à une clé entière et accessible par d'autres processus grâce à cette clé. L'adresse de base de la mémoire partagée, vue de deux processus, n'a aucune obligation d'être identique : les processus ne sont pas obligés d'utiliser la zone d'octets à partir de son "début".

A tout segment de mémoire partagée, est associée une structure d'informations de *type struct shmid_ds*, définie dans **/usr/include/sys/shm.h**, et consultable par la fonction système **ipcs**. Ce fichier définit également un certain nombre de constantes.

Une action **exit** sur un processus ayant créé des segments libère ceux-ci.

1. CREATION D'UN SEGMENT DE MEMOIRE PARTAGEE

La fonction prototypée par :

int shmget (key_t cle, int taille, int option)

retourne l'identificateur d'un segment de mémoire partagée associée à une clé donnée (-1 en cas d'erreur), avec :

taille : la taille du segment en octets (comprise entre un minimum et un maximum définis lors de la configuration du système)

option : combinaison, avec l'opérateur | , de droits d'accès et de constantes prédéfinies:
SHM_R : permission en lecture (ou 0400)
SHM_W : permission en écriture (ou 0200)
IPC_CREAT, IPC_EXCL

cle : clé de type key_t (défini dans /usr/include/sys/ipc.h, équivalent à long)

si **cle** = **IPC_PRIVATE**, un nouveau segment est créé

si non si **cle** ne correspond pas à un segment existant,
si **IPC_CREAT** n'est pas positionné : erreur et retour de -1
sinon, un nouveau segment est créé et associé à cette clé avec les droits d'accès mentionnés. Le propriétaire et le créateur sont le

propriétaire effectif du processus. Le groupe propriétaire et le groupe créateur sont le groupe propriétaire effectif du processus

sinon si IPC_CREAT **et** IPC_EXCL sont positionnés : erreur,retour de -1
sinon, la fonction retourne l'identificateur du segment

Exemple :récupération de l'identificateur d'un segment

```
num = shmget (ftok (CHEMIN, cle) , taille, 0)
/* ou bien : num = shmget (ftok(CHEMIN, cle) , 0 , 0)  */
```

Exemple de création d'un segment :

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

main ()
{
        int num;
        if ((num = shmget (IPC_PRIVATE, 1024, IPC_CREAT | IPC_EXCL | 0666)) == -1)
                fprintf (stderr, "erreur\n");
        else printf ("segment d'identificateur %d créé\n", num);
}
```

2. OPERATIONS DE CONTROLE

La fonction prototypée par :

int shmctl (int id, int cmd, struct shmid_ds *buf)

retourne 0 (- 1 en cas d'erreur), avec :

id : identificateur du segment (le retour de shmget)

buf : pointeur sur la structure shmid_ds, contenant une copie des informations sur le segment

cmd : IPC_STAT : la structure shmid_ds associée au segment est copiée dans la structure pointée par buf
IPC_SET : la structure shmid_ds associée au segment est modifiée par les valeurs des champs de la structure pointée par buf
IPC_RMID : suppression du segment de la mémoire. La suppression sera différée si le segment est encore attaché à un processus.

3. ATTACHEMENT D'UN SEGMENT A UN PROCESSUS

La fonction prototypée par :

void * shmat (int id, const void *adr, int flag)

retourne adr ou NULL en cas d'erreur, avec :

id : identificateur du segment (le retour de shmget)
adr : si adr = NULL, le système choisit la première adresse disponible et la fonction la retourne.
Alors, adr pointe sur le 1er octet du segment.
Avantage : l'adresse sera correctement construite

si adr != NULL, et si SHM_RND est positionné dans flag, le segment est attaché à l'adresse adr - (adr modulo SHMLBA, *segment low boundary address*, adresse de base de la mémoire partagée)

si adr != NULL, et si SHM_RND n'est pas positionné dans flag, le segment est attaché à l'adresse adr

si SHM_RDONLY (010000) est positionné dans flag, le segment n'est attaché qu'en lecture. Toute tentative d'écriture génère le signal SIGSEGV

Attention, les segments commencent à des limites de pages.

4. DETACHEMENT D'UN SEGMENT

La fonction prototypée par :

int shmdt (const void *ptr)

retourne 0 (- 1 en cas d'échec)

ptr : adresse retournée par shmat.

Cette fonction **détache** le segment du processus qui l'utilise.

Attention, ptr garde sa valeur et on évitera de l'utiliser après un appel à shmdt. La fonction shmdt **ne détruit pas** le segment. Seul un appel à shmctl peut le faire.

La terminaison d'un processus entraîne le détachement de tous les segments qu'il a préalablement créés.

Chapitre 16 PROTECTION - SECURITE

Le coût des déficiences et des défectuosités en informatique est élevé :

- erreurs propagées par un programme utilisateur,
- erreurs dans le SE,
- pannes matérielles,
- malveillances, etc...

Le coût de toute protection est également élevé. D'où la recherche d'un compromis privilégiant des objectifs stratégiques.

Le degré de protection dépend de la protection des informations manipulées, du degré de confiance que l'on peut accorder aux logiciels, dont le SE.

Un logiciel fiable :
- satisfait à ses spécifications,
- résiste à un environnement imprévu :
  - soit en le signalant,
  - soit en le corrigeant

en évitant la propagation des erreurs.

1. CONTROLE D'ACCES AUX RESSOURCES PAR UN PROCESSUS

Il s'effectue à deux niveaux :

- niveau 1 (logique) : en développant un modèle de protection : des règles spécifiant quels accès sont autorisés et quels accès sont interdits

- niveau 2 (physique) : en mettant en oeuvre des mécanismes de protection pour appliquer le modèle

2. MODELE DE PROTECTION PAR MATRICE D'ACCES

Le SE comporte :

- des entités actives ou **sujets** : les processus
- des entités accessibles ou **objets**, représentants de *types abstraits*. Ce sont :

  - les ressources matérielles : UC, mémoire centrale, entrées/sorties, etc...
  - les ressources logicielles : fichiers, programmes, sémaphores, etc...

Un modèle de protection doit répondre à la question : *Quels sujets ont accès à quels objets ? De quelle façon (modalités d'accès) ?* Etant entendu qu'il convient de prévoir une modalité d'accès par type abstrait.

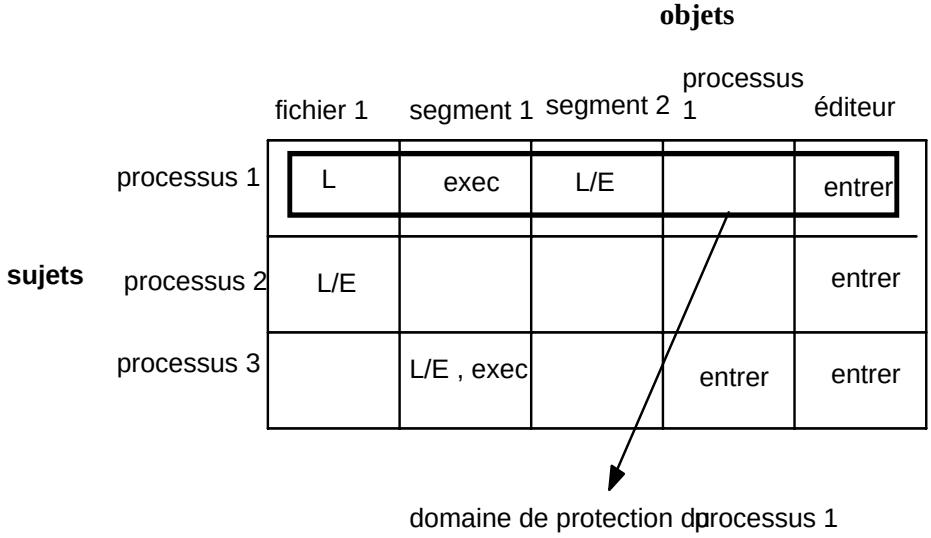
Définition : On appelle **droit d'accès** tout couple (nom d'objet, modalités d'accès à cet objet). Par exemple (fichier f1 , lire) ou (processus P0 , appeler)

Le modèle de protection fixe le droit d'accès de chaque processus à chaque instant. L'ensemble de ces droits pour un processus est appelé **domaine de protection du processus**.

Exemple : D = { (fichier f1 , ouvrir) , (fichier f1 , fermer) , (fichier f1 , lire) , (fichier f1 , écrire) , (éditeur , appeler) , (console 5 , lire) , (imprimante , afficher) }

Un domaine de protection peut se représenter par une **matrice d'accès**.

Exemple :



2.1 Domaine de protection restreint

Un processus peut être décomposé en tâches, avec des droits d'accès différents selon les tâches. Donc, la matrice doit pouvoir évoluer dynamiquement :

- pour ajouter un droit d'accès à la case (i , j)
- pour retrancher un droit d'accès à la case (i , j)

Par exemple, si on ajoute **(domaine j , entrer)** en ligne i, le processus qui s'exécute dans le domaine de protection i passe au domaine de protection j.

Exemple :

Le degré de protection d'un SE est celui de son maillon le plus faible ("principe de la porte dérobée"). Un procédé de contrôle simple, mais permanent est préférable à un procédé complexe, mais pas systématiquement utilisé. De plus, il faut moduler les contrôles en fonction des utilisateurs. Pour ces raisons, tout type abstrait de domaine de protection doit être modulaire.

2. 2 Problème du cheval de Troie

Un programme peut profiter du domaine de protection de l'utilisateur pour consulter ou copier ou modifier des données auxquelles il ne devrait pas avoir accès. **Il est donc important d'exécuter tout programme dans le domaine de protection le plus réduit**.

2.3 Problème du confinement

Un programme, exécuté par un utilisateur qui n'est pas son propriétaire, peut retenir une information confidentielle : **il faut donc confiner l'information**.

Les mécanismes de contrôle rassemblés dans la matrice d'accès peuvent être classés en trois catégories :

- accès hiérarchiques
- listes d'accès
- capacités

3. ACCES HIERARCHIQUES

On peut hiérarchiser les accès en deux modes :

- le mode système ou **maître** : par exemple tout ce qui concerne la création ou la destruction des processus, l'accès aux tables du SE, les instructions d'entrée-sortie. Le domaine de protection est vaste : tous les droits du SE (processus système, super utilisateur(s))

- le mode utilisateur ou **esclave** : le domaine de protection est celui d'un sous-ensemble d'instructions autorisées et utilisées

Souvent, un bit du mot d'état de la machine indique quel est le mode de protection pour la mise en œuvre des contrôles.

Toutefois, il n'est pas facile de mettre en œuvre le ***principe du moindre privilège*** énoncé ci-dessus. Une solution peut être d'organiser des domaines de protection en anneaux concentriques. L'anneau central n° 0 correspond au maximum de privilèges. La mémoire centrale est segmentée avec une partition par anneau (par domaine de protection).

Le cas de 2 anneaux correspond à la situation simple maître/esclave.

Lorsqu'un processus s'exécute dans le domaine D_i (i = 0, 1, 2 , ...), il peut accéder à tout segment de mémoire centrale de niveau n° k (k # i, mais jamais k < i). C'est en contradiction avec le principe de moindre privilège.

Une solution élégante à ce problème a été proposée pour la 1ère fois pour la gamme des microprocesseurs INTEL 80286 en 1984 : il s'agit de la technique des guichets (**gates)**. Un processus n'accède à un module de niveau inférieur que par un guichet facilement contrôlable.

Définition : un guichet est un descripteur, associé à un numéro d'anneau, qui pointe vers un segment de mémoire de n° d'anneau inférieur.

Règles : - tout processus qui peut accéder à un guichet accède au segment pointé par le guichet
- il y a 2 solutions, selon les systèmes, pour répondre à la question : "*dans quel domaine de protection travaille-t-on dans l'anneau appelé ?* "

solution 1 ou principe de conformité : un processus de niveau k (k < i) qui accède à un guichet pointant sur le niveau i s'exécute dans l'anneau i. Il n'a pas accès à la mémoire de niveau k, son propre niveau !

solution 2 ou restriction de privilèges : on associe temporairement au segment de mémoire de niveau k le numéro d'anneau i pour qu'un processus de niveau k puisse accéder à cet anneau quand il est exécuté au niveau i

4. LISTES D'ACCES (authorized list)

Chaque colonne de la matrice d'accès est structurée en une liste chaînée de protections associée à un objet. C'est le propriétaire de l'objet qui définit la liste.

Lorsqu'un processus veut accéder à un objet, le SE vérifie dans la liste que l'accès est permis.

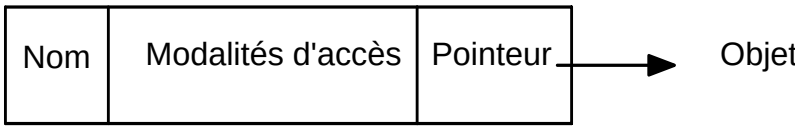
La technique UNIX des bits de protection rwx .... s'apparente à cette solution.

Toutefois, un problème se pose : il n'est pas immédiat de modifier un domaine de protection, puisqu'il faut pour cela explorer ***toutes*** les listes. Cette solution n'est donc pas bien adaptée à la mise en œuvre du principe de moindre privilège.

5. CAPACITES (capability)

Une capacité est une structure à 3 champs :

- nom d'un objet
- modalités d'accès à cet objet
- pointeur sur l'objet



Règles : - une capacité est créée par un processus particulier du SE

- les modalités d'accès sont le seul des 3 champs d'une capacité dont la modification est autorisée

- un sujet peut transmettre la copie d'une capacité à un autre sujet, éventuellement avec des modalités d'accès réduites

- lorsqu'un sujet crée un objet, le SE lui alloue une capacité associée à l'objet

Une capacité correspond à un **découpage en lignes** de la matrice d'accès. Le domaine de protection d'un processus est formé de la liste des capacités (***C-liste***) qu'il possède à un instant donné.

Exemple :

Pour un processus, changer de domaine de protection peut consister à changer de C-liste.

5.1 Domaines de protection restreints

Une solution élégante consiste à faire de chaque C-liste utilisée un objet n'admettant que la seule modalité d'accès ***entrer*** . Pour accéder à cet objet, donc pour entrer dans le domaine de protection décrit par la C-liste, un sujet doit posséder une capacité d'entrée pour cet objet.

Cette méthode permet de bien réaliser le principe du moindre privilège. Ainsi, dans la figure ci-dessous, lorsqu'un processus appelle une procédure, il doit posséder une capacité lui permettant d'accéder à la C-liste de la procédure.

Règles : - après l'appel à une procédure, le domaine de protection du processus devient celui de la C-liste de la procédure

- une fois que le processus est sorti de la procédure, son ancien domaine de protection est restauré

5.2 Protection des capacités

Elle est normalement effectuée au niveau de l'objet, c'est à dire du matériel. Deux solutions sont envisageables :

- **solution 1 : étiquetage**
Dans chaque mot mémoire, un bit indique si le mot est une capacité. Ainsi, il n'y aura pas de manipulation illicite des droits d'accès d'une capacité.

- solution 2 :partition de la mémoire

Chaque segment de mémoire contient soit des capacités, soit des données. Il existe des registres spécifiques pour la manipulation des capacités, d'autres pour les données.

Règles : les processus système accèdent aux registres capacité et aux segments capacité. Les données ne peuvent être copiées que dans des segments ou des registres de données.

Un problème se pose : la **révocation d'une capacité**.

Les capacités transmises à d'autres sujets sont disséminées. C'est extrêmement incommode pour supprimer une capacité.

La solution généralement adoptée est la suivante : le créateur d'un objet de nom O ne transmet pas à un autre sujet une copie de la capacité de O, mais il transmet une capacité pour l'objet O' qui pointe sur O. Comme la capacité de O n'est pas dupliquée, son éventuelle suppression est simple : pour révoquer, on supprime le pointeur de O' sur O.

Bibliographie

J. BEAUQUIER, B. BERARD, Systèmes d'exploitation, Edisciences
(plusieurs figures du chapitre sont issues de cet ouvrage).

Chapitre 1 GENERALITES

Les logiciels peuvent être classés en deux catégories :

- les programmes d'application des utilisateurs
- les programmes système qui permettent le fonctionnement de l'ordinateur. Parmi ceux-ci, le système d'exploitation (SE dans la suite).

Le SE soustrait le matériel au regard du programmeur et offre une présentation agréable des fichiers. Un SE a ainsi deux objectifs principaux :

- **présentation** : Il propose à l'utilisateur une abstraction plus simple et plus agréable que le matériel : une **machine virtuelle**

- **gestion** : il ordonne et contrôle l'allocation des processeurs, des mémoires, des icônes et fenêtres, des périphériques, des réseaux entre les programmes qui les utilisent. Il assiste les programmes utilisateurs. Il protège les utilisateurs dans le cas d'usage partagé.

1. HISTORIQUE

Les premiers ordinateurs étaient mis à la disposition d'un programmeur selon un calendrier de réservation : un usager avec un travail unique utilisait seul la machine à un moment donné. Puis vint l'époque du **traitement par lots** (batch) : enchaînement, sous le contrôle d'un moniteur, d'une suite de travaux avec leurs données, confiés à l'équipe d'exploitation de la machine (inconvénient : temps d'attente des résultats pour chaque utilisateur).

Cette pratique a nécessité trois innovations :

- le contrôle des E/S et leur protection pour éviter le blocage d'un lot
- un mécanisme de comptage de temps et de déroutement autoritaire des programmes pour éviter le blocage d'un lot à cause d'une séquence trop longue. Ce furent les première interruptions
- les premiers langages de commande (JCL) sous forme de cartes à contenu particulier introduites dans le paquet ($JOB, $LOAD, $RUN, etc...)

Historiquement, on peut dire que les SE sont vraiment nés avec les ordinateurs de la 3ème génération (ordinateurs à circuits intégrés apparus après 1965). Le premier SE digne de ce nom est l'OS/360, celui des IBM 360, famille unique de machines compatibles entre elles, de puissances et de configurations différentes. Bien que son extrême complexité (due à l'erreur de couvrir toute la gamme 360) n'ait jamais permis d'en réduire le nombre de bogues, il apportait deux concepts nouveaux :

- **la multiprogrammation** : partitionnement de la mémoire permettant au processeur d'accueillir une tâche dans chaque partie et donc d'être utilisé plus efficacement par rapport aux temps d'attente introduits par les périphériques (le processeur est ré-alloué)

- **les E/S tamponnées** : adjonction à l'UC d'un processeur autonome capable de gérer en parallèle les E/S ou canal ou unité d'échange. Cela nécessite une politique de partage du bus ou d'autres mécanismes (vol de cycle, DMA).

Au MIT, F.J. CORBATO et son équipe ont réalisé dès 1962, sur un IBM 7094 modifié, le premier SE expérimental à **temps partagé** (mode interactif entre plusieurs utilisateurs

simultanés), baptisé CTSS. Une version commerciale, nommée MULTICS (MULTiplexed Information and Computing Service), a été ensuite étudiée par cette équipe, les Bell Laboratories et General Electric. Les difficultés ont fait que MULTICS n'a jamais dépassé le stade expérimental sur une douzaine de sites entre 1965 et 1974. Mais il a permis de définir des concepts théoriques importants pour la suite.

La technologie à base de circuits intégrés de la 3ème génération d'ordinateurs a permis l'apparition des mini-ordinateurs et leur diffusion massive (précédant celle des micro-ordinateurs). En 1968, l'un des auteurs de MULTICS, Ken Thompson a effectué une adaptation de MULTICS mono-utilisateur sur un mini-ordinateur PDP-11 de DEC inutilisé dans son Laboratoire des Bell Laboratories. Son collègue Brian Kernighan la nomma UNICS (Uniplexed - *à l'opposé de Multiplexed* - Information and Computer Service), qui devint ensuite UNIX . En 1972, son collègue Dennis Ritchie traduisit UNIX en C, langage qu'il venait de mettre au point avec Kernighan..... L'ère des grands SE avait commencé.

Avec la grande diffusion des micro-ordinateurs, l'évolution des performances des réseaux de télécommunications, deux nouvelles catégories de SE sont apparus :

- **les SE en réseaux** : ils permettent à partir d'une machine de se connecter sur une machine distante, de transférer des données. Mais chaque machine dispose de son propre SE

- **les SE distribués ou répartis** : l'utilisateur ne sait pas où sont physiquement ses données, ni où s'exécute son programme. Le SE gère l'ensemble des machines connectées. Le système informatique apparaît comme un mono-processeur.

2. ELEMENTS DE BASE D'UN SYSTEME D'EXPLOITATION

Les principales fonctions assurées par un SE sont les suivantes :

- gestion de la mémoire principale et des mémoires secondaires,
- exécution des E/S à faible débit (terminaux, imprimantes) ou haut débit (disques, bandes),
- multiprogrammation, temps partagé, parallélisme : interruption, ordonnancement, répartition en mémoire, partage des données
- lancement des outils du système (compilateurs, environnement utilisateur,...) et des outils pour l'administrateur du système (création de points d'entrée, modification de privilèges,...),
- lancement des travaux,
- protection, sécurité ; facturation des services,
- réseaux

L'interface entre un SE et les programmes utilisateurs est constituée d'un ensemble d'instructions étendues, spécifiques d'un SE, ou appels système. Généralement, les appels système concernent soit les processus, soit le système de gestion de fichiers (SGF).

2.1 Les processus

Un processus est un programme qui s'exécute, ainsi que ses données, sa pile, son compteur ordinal, son pointeur de pile et les autres contenus de registres nécessaires à son exécution.

Attention : ne pas confondre un processus (aspect dynamique, exécution qui peut être suspendue, puis reprise), avec le texte d'un programme exécutable (aspect statique).

Les appels système relatifs aux processus permettent généralement d'effectuer au moins les actions suivantes :

- création d'un processus (fils) par un processus actif (d'où la structure d'arbre de processus gérée par un SE)
- destruction d'un processus
- mise en attente, réveil d'un processus
- suspension et reprise d'un processus, grâce à l'ordonnanceur de processus (scheduler)
- demande de mémoire supplémentaire ou restitution de mémoire inutilisée
- attendre la fin d'un processus fils
- remplacer son propre code par celui d'un programme différent
- échanges de messages avec d'autres processus
- spécification d'actions à entreprendre en fonction d'événements extérieurs asynchrones
- modifier la priorité d'un processus

Dans une entité logique unique, généralement un mot, le SE regroupe des informations-clés sur le fonctionnement du processeur : c'est le **mot d'état du processeur** (Processor Status Word, PSW). Il comporte généralement :

- la valeur du compteur ordinal
- des informations sur les interruptions (masquées ou non)
- le privilège du processeur (mode maître ou esclave)
- etc.... (format spécifique à un processeur)

A chaque instant, un processus est caractérisé par son **état courant** : c'est l'ensemble des informations nécessaires à la poursuite de son exécution (valeur du compteur ordinal, contenu des différents registres, informations sur l'utilisation des ressources). A cet effet, à tout processus, on associe un **bloc de contrôle de processus** (BCP). Il comprend généralement :

- une copie du PSW au moment de la dernière interruption du processus
- l'état du processus : prêt à être exécuté, en attente, suspendu, ...
- des informations sur les ressources utilisées
    - mémoire principale
    - temps d'exécution
    - périphériques d'E/S en attente
    - files d'attente dans lesquelles le processus est inclus, etc...
- et toutes les informations nécessaires pour assurer la reprise du processus en cas d'interruption

Les BCP sont rangés dans une table en mémoire centrale à cause de leur manipulation fréquente.

2.2 Les interruptions

Une interruption est une commutation du mot d'état provoquée par un signal généré par le matériel. Ce signal est la conséquence d'un événement interne au processus, résultant de son exécution, ou bien extérieur et indépendant de son exécution. Le signal va modifier la valeur d'un indicateur qui est consulté par le SE. Celui-ci est ainsi informé de l'arrivée de l'interruption et de son origine. A chaque cause d'interruption est associé un **niveau** d'interruption. On distingue au moins 3 niveaux d'interruption :

- les interruptions externes : panne, intervention de l'opérateur, ...
- les déroutements qui proviennent d'une situation exceptionnelle ou d'une erreur liée à l'instruction en cours d'exécution (division par 0, débordement, ...)
- les appels système

UNIX admet 6 niveaux d'interruption : interruption horloge, interruption disque, interruption console, interruption d'un autre périphérique, appel système, autre interruption.

Le chargement d'un nouveau mot d'état provoque l'exécution d'un autre processus, appelé le **traitant** de l'interruption. Le traitant réalise la sauvegarde du contexte du processus interrompu (compteur ordinal, registres, indicateurs,....). Puis le traitant accomplit les opérations liées à l'interruption concernée et restaure le contexte et donne un nouveau contenu au mot d'état : c'est l'**acquittement** de l'interruption.

Généralement un numéro de priorité est affecté à un niveau d'interruption pour déterminer l'ordre de traitement lorsque plusieurs interruptions sont positionnées. Il est important de pouvoir retarder, voire annuler la prise en compte d'un signal d'interruption. Les techniques que l'on utilise sont le **masquage** et le **désarmement** des niveaux d'interruption :

- le masquage d'un niveau retarde la prise en compte des interruptions de ce niveau. Pour cela, on positionne un indicateur spécifique dans le mot d'état du processeur. Puisqu'une interruption modifie le mot d'état, on peut masquer les interruptions d'autres niveaux pendant l'exécution du traitant d'un niveau. Lorsque le traitant se termine par un acquittement, on peut alors démasquer des niveaux qui avaient été précédemment masqués. Les interruptions intervenues pendant l'exécution du traitant peuvent alors être prises en compte

- le désarmement d'un niveau permet de supprimer la prise en compte de ce niveau par action sur le mot d'état. Pour réactiver la prise en compte, on réarme le niveau. Il est évident qu'un déroutement ne peut être masqué; il peut toutefois être désarmé.

2.3 Les ressources

On appelle **ressource** tout ce qui est nécessaire à l'avancement d'un processus (continuation ou progression de l'exécution) : processeur, mémoire, périphérique, bus, réseau, compilateur, fichier, message d'un autre processus, etc... Un défaut de ressource peut provoquer la mise en attente d'un processus.

Un processus demande au SE l'accès à une ressource. Certaines demandes sont implicites ou permanentes (la ressource processeur). Le SE **alloue** une ressource à un processus. Une fois une ressource allouée, le processus a le droit de l'utiliser jusqu'à ce qu'il **libère** la ressource ou jusqu'à ce que le SE reprenne la ressource (on parle en ce cas de **ressource préemptible**, de préemption).

On dit qu'une ressource est en mode d'**accès exclusif** si elle ne peut être allouée à plus d'un processus à la fois. Sinon, on parle de mode d'**accès partagé**. Un processus possédant une ressource peut dans certains cas en modifier le mode d'accès.

Exemple : un disque est une ressource à accès exclusif (un seul accès simultané), une zône mémoire peut être à accès partagé.

Le mode d'accès à une ressource dépend largement de ses caractéristiques technologiques. Deux ressources sont dites équivalentes si elles assurent les mêmes fonctions vis à vis du processus demandeur. Les ressources équivalentes sont groupées en classes afin d'en faciliter la gestion par l'ordonnanceur.

2.4 L'ordonnancement

On appelle ordonnancement la stratégie d'attribution des ressources aux processus qui en font la demande. Différents critères peuvent être pris en compte :

- temps moyen d'exécution minimal
- temps de réponse borné pour les systèmes interactifs
- taux d'utilisation élevé de l'UC
- respect de la date d'exécution au plus tard, pour le temps réel, etc...

2.5 Le système de gestion de fichiers

Une des fonctions d'un SE est de masquer les spécificités des disques et des autres périphériques d'E/S et d'offrir au programmeur un modèle de manipulation des fichiers agréable et indépendant du matériel utilisé.

Les appels système permettent de créer des fichiers, de les supprimer, de lire et d'écrire dans un fichier. Il faut également ouvrir un fichier avant de l'utiliser, le fermer ultérieurement. Les fichiers sont regroupés en répertoires arborescents; ils sont accessibles en énonçant leur chemin d'accès (chemin d'accès absolu à partir de la racine ou bien chemin d'accès relatif dans le cadre du répertoire de travail courant).

Le SE gère également la protection des fichiers.

3. STRUCTURE D'UN SYSTEME D'EXPLOITATION

On peut distinguer quatre grandes catégories de SE.

3.1 Les systèmes monolithiques

Le SE est un ensemble de procédures, chacune pouvant appeler toute autre à tout instant. Pour effectuer un appel système, on dépose dans un registre les paramètres de l'appel et on exécute une instruction spéciale appelée **appel superviseur ou appel noyau**. Son exécution commute la machine du mode utilisateur au **mode superviseur ou noyau** et transfère le contrôle au SE. Le SE analyse les paramètres déposés dans le registre mentionné plus haut et en déduit la procédure à activer pour satisfaire la requête. A la fin de l'exécution de la procédure système, le SE rend le contrôle au programme appelant.

Généralement, un tel SE est organisé en 3 couches :

- une procédure principale dans la couche supérieure, qui identifie la procédure de service requise

- des procédures de service dans la couche inférieure à la précédente qui exécutent les appels système

- des procédures utilitaires dans la couche basse qui assistent les procédures système. Une procédure utilitaire peut être appelée par plusieurs procédures systèmes.

3.2 Les systèmes en couches

On peut généraliser la conception précédente et concevoir un SE composé de plusieurs couches spécialisées, chaque couche ne pouvant être appelée que par des procédures qui lui sont immédiatement inférieures. Citons par exemple le premier SE de cette nature proposé par Dijkstra en 1968 :

- couche 0 : allocation du processeur par commutation de temps entre les processus, soit à la suite d'expiration de délais, soit à la suite d'interruption (multiprogrammation de base du processeur)

- couche 1 : gestion de la mémoire, allocation d'espace mémoire pour les processus (pagination)

- couche 2 : communication entre les processus et les terminaux

- couche 3 : gestion des E/S (échanges d'information avec des mémoires tampons, c'est à dire avec des périphériques abstraits, dégagés des spécificités matérielles)

- couche 4 : programmes utilisateurs

3.3 Les machines virtuelles

Une des premiers SE à gérer le concept de machine virtuelle a été l'adaptation temps partagé de l'OS/360 d'IBM, proposé vers 1968 sous le nom de CP/CMS, puis sous le nom de VM/370 en 1979.

Le cœur du SE, appelé moniteur de machine virtuelle ou VM/370, s'exécute à même le matériel et fournit à la couche supérieure plusieurs machines virtuelles. Ces machines virtuelles sont des copies conformes de la machine réelle avec ses interruptions, ses modes noyau/utilisateur, etc...

Chaque machine virtuelle peut exécuter son propre SE. Lorsqu'une machine virtuelle exécute en mode interactif un appel système, l'appel est analysé par le moniteur temps partagé de cette machine, CMS. Toute instruction d'E/S, toute instruction d'accès mémoire est convertie par VM/370 qui les exécute dans sa simulation du matériel. La séparation complète de la multiprogrammation et de la machine étendue rend les éléments du SE plus simples et plus souples. VM/370 a gagné en simplicité en déplaçant une grande partie du code d'un SE dans le moniteur CMS.

3.4 L'architecture client/serveur

Cette tendance s'est accentuée dans les SE contemporains en tentant de réduire le SE à un noyau minimal. Une des formes les plus accentuées de cette évolution est l'architecture client/serveur.

La plupart des fonctionnalités d'un SE sont reportées dans des processus utilisateurs. Pour demander un service comme la lecture d'un bloc de fichier, le processus utilisateur ou processus client envoie une requête à un processus serveur qui effectue le travail et envoie une réponse. Le noyau ne gère que la communication entre les clients et les serveurs. Cependant, le noyau est souvent obligé de gérer certains processus serveurs critiques comme les pilotes de périphériques qui adressent directement le matériel.

La décomposition du SE en modules très spécialisés le rend facile à modifier. Les serveurs s'exécutent comme des processus en mode utilisateur et non pas en mode noyau. Comme ils n'accèdent donc pas directement au matériel, une erreur n'affecte que le serveur et pas l'ensemble de la machine.

En outre, ce modèle est bien adapté aux systèmes distribués. Un client n'a pas besoin de savoir si le SE fait exécuter sa requête par un serveur de sa propre machine ou celui d'une machine distante.

Chapitre 2 NOTIONS SUR UNIX

Avertissement : UNIX est une marque déposée de Bell Laboratories.

1. GENERALITES

1.1 Historique

Vers 1965, Bell Laboratories, General Electric et le M.I.T. lancent le projet MULTICS qui vise à créer un système d'exploitation **multitâches et multi-utilisateurs en temps partagé**, implantable sur tout ordinateur, assurant une **portabilité** des applications. Les chefs de projets étaient Ken THOMPSON et Dennis RITCHIE de Bell Laboratories. Une première version écrite en PL/1 est proposée en 1970, avec deux originalités : un système de gestion de fichiers arborescent et un **shell** conçu comme processus utilisateur. Mais c'est l'échec, PL/1 n'étant pas adapté aux objectifs.

Après le retrait de Bell Laboratories, lassé d'avoir investi 7 millions de $ sans retour, Ken THOMPSON et Dennis RITCHIE continuent seuls et réécrivent MULTICS en Assembleur. Sur la machine hébergeant le premier MULTICS, Ken THOMPSON crée le langage B en 1971 et en 1973, Dennis RITCHIE et Brian KERNIGHAN créent le langage C issu d'améliorations de B, en vue d'une réécriture d'UNIX, nom qui a succédé à MULTICS (*Uniplexed Information and Computer Service*). Ce fut la première version (V6) commercialisée en 1974. Depuis, plus d'une vingtaine de versions d'UNIX ont vu le jour, sans compter les systèmes très proches. Les quatre versions principales sont BSD, SYSTEM V, POSIX (tentative de fédérer les deux familles précédentes) et LINUX proposé par Linus TORVALDS en 1991. Le code source, écrit pour l'essentiel en C, est largement accessible.

Le label UNIX 98 de l'Open Group prescrit les threads (tâches) standardisés (cf. chapitre 11), le temps réel, les 64 bits et les systèmes de fichiers étendus (cf. chapitre 7).

Quelles que soient les versions :
UNIX = le noyau + l'interpréteur de commandes + des utilitaires

UNIX est construit autour de deux notions essentielles :
- **fichiers** organisés en structure arborescente
- **processus** : un processus correspond à l'exécution d'un programme (système ou utilisateur) et à son environnement (descripteurs de fichiers, registres, compteur ordinal, …). La gestion de la mémoire centrale constitue un aspect important du contrôle des processus.

1.2 Les entrées-sorties

Dans UNIX, toute unité d'E/S est désignée par un fichier spécial (cf. ce que vous connaissez en C sur stdin, stdout, stderr). Il existe deux modes d'échange pour l'E/S :

- **bloc** : les mémoires de masse (disque, disquettes) échangent des blocs d'information avec la mémoire centrale en utilisant des tampons

- **caractère** : les terminaux, les imprimantes et les réseaux échangent des caractères avec la mémoire centrale, sans tampon.

On peut rediriger les E/S standard :
< pour redéfinir stdin
> pour redéfinir stdout
>> idem, mais les données sont ajoutées en fin de fichier

La redirection s'applique à la dernière commande (parenthèser pour une redirection portant sur un groupe de commandes. La redirection de **stderr** nécessite d'utiliser son descripteur de valeur 2 :

commande 2 > fichier

echo "bonjour" > &2 redirige le message sur la sortie standard d'erreurs

commande > test 2 > &1 redirige la sortie vers le fichier test et la sortie standard des erreurs vers la sortie standard

Pour rediriger stdin pendant toute la session d'exécution de *exec*, on utilisera :
exec < fichier. De même exec > fichier.

Il est possible de transmettre des arguments à *main* () d'un exécutable C lors du lancement de son exécution grâce aux deux arguments : **main (int argc, char **argv)** dans cet ordre où :

argc (arguments count) est le compteur d'arguments, nom du programme exécutable inclus

argv (arguments vector) est un vecteur de pointeurs sur chaînes de caractères qui contiennent les arguments de la ligne de commande, à raison de un par chaîne ; argv [0] pointe sur le nom de l'exécutable et argv [argc] vaut NULL.

Exemple : tp1.e toto titi
argc vaut 3, argv[0] pointe sur "tp1.e" ; argv[1] pointe sur "toto" ; argv[2] pointe sur "titi"

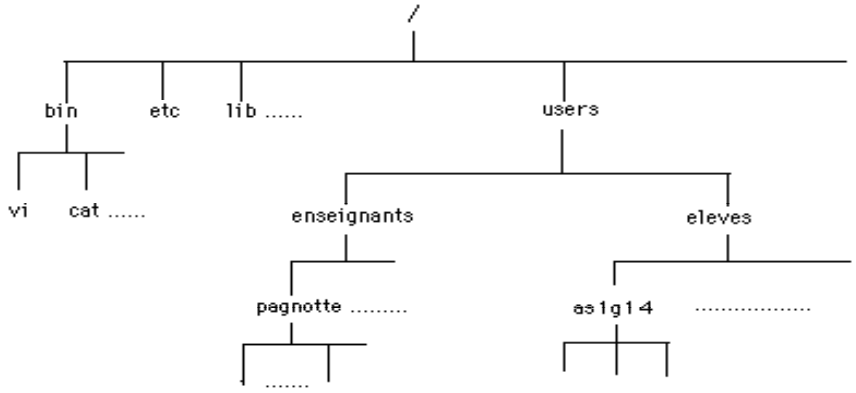
1.3 Utilitaires standards

UNIX dispose d'**utilitaires standards** nombreux parmi lesquels (voir annexe pour certains) :

- compilateurs (notamment C) avec son vérificateur **lint**, ses débogueurs **adb**, **sdb** et **dbx** et son archiveur de bibliothèques **ar**
- gestionnaire d'applications **make** et **sccs**
- outils de génération de compilateurs **lex** et **yacc**

éditeurs de texte / manipulateurs de documents **ed, sed, vi, emacs, nroff, troff**

2. LE SYSTEME DE GESTION DE FICHIERS (SGF)



Le S.G.F. gère une structure d'arbre :
- la racine est désignée par /
- les nœuds sont les répertoires non vides
- les feuilles sont les répertoires "vides" et les fichiers.

Les chemins sont décrits avec le séparateur /

Exemple : */users/eleves/m-dupont96* est une référence absolue (commence par /)
source/tp1.c est une référence relative (par rapport au répertoire courant), car ne commence pas par /

Un fichier est une suite d'octets, généralement non structurée.
Les noms de fichiers et répertoires sont limités à 14 caractères (/ est interdit et - ne peut être le premier caractère).

A chaque instant, l'utilisateur accède à un *répertoire courant* dans cette arborescence. Il se déplace avec la commande **cd**.

2.1 Les i-nodes

A tout fichier est attaché un nœud d'index, ou **i-node** contenant des informations sur ce fichier. C'est une structure de quelques dizaines d'octets décrite dans **/usr/include/sys/inode.h** qui contient généralement les champs suivants :

- le type (fichier ordinaire, spécial, catalogue ,...),
- les protections,
- les dates de création, accès et mise à jour (en s. écoulées depuis le 1/1/1970),
- la taille du fichier en octets,
- le nombre de liens (**un lien** d'un fichier est un autre nom de ce fichier),
- les éléments d'identification du propriétaire et de son groupe,
- l'adresse physique d'implantation sur disque (cf. chapitre 7) sous forme de 13 adresses de blocs disques

Bon exemple : Braquelaire p. 370-378.

L'i-node ne contient pas de "nom de fichier". La désignation d'un fichier se fait par l'intermédiaire du répertoire dans lequel est stocké le numéro d'i-node (entier naturel sur 2 octets) et le "nom relatif"

sur 14 octets. Ainsi, tout fichier est référencé de manière unique par son ***numéro d'i-node*** , même si son "nom relatif" peut être non unique dans l'arborescence.

L'i-node de n° 0 n'est jamais utilisée. L'i-node de n° 1 permet la gestion des blocs défectueux. La racine de l'arborescence possède l'i-node de n° 2.

Dans chaque répertoire,'•' est un lien au répertoire lui-même (dans le répertoire, en regard de ce "nom", figure le n° d'i-node du répertoire).De même, '••' est un lien au répertoire père (dans le répertoire, en regard de ce nom, figure le n° d'i-node du répertoire père). Un répertoire UNIX n'est donc jamais totalement vide.

Les i-nodes des fichiers d'un disque sont implantés par numéro croissant au début du disque (*voir 2.4*), ce qui en facilite beaucoup l'accès. Les i-nodes des fichiers ouverts sont copiés en mémoire centrale dans la **table des i-nodes**.

2.2 Les différents types de fichiers

On distingue :
- les fichiers ordinaires : données, programmes, textes

- les fichiers spéciaux : ils correspondent à des ressources. Ils n'ont pas de taille. Les opérations de lecture/écriture sur ces fichiers activent les dispositifs physiques associés. On distingue :

  - **character devices :** fichiers spéciaux en mode caractère (par exemple : terminaux)
- **block devices :** fichiers spéciaux en mode bloc (par exemple : disques)
  un bloc comprend 512 octets, 1024 ou davantage.

- les fichiers répertoires contiennent des informations sur d'autres fichiers et permettent la structuration arborescente

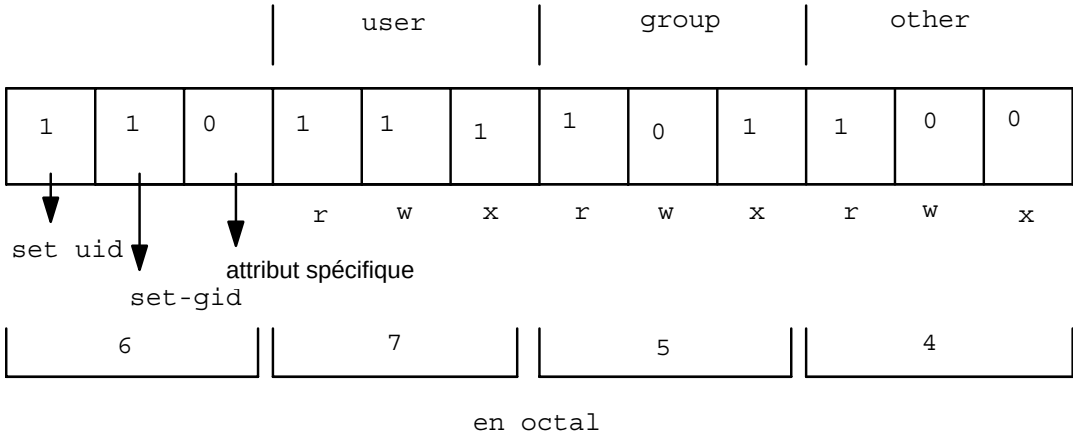
- les fichiers de liens symboliques : les fichiers qui sont des tubes nommés, les fichiers qui sont des prises réseaux.

2.3 Les protections des fichiers

A la création d'un compte d'**utilisateur**, l'administrateur affecte celui-ci à un **groupe**. Chaque groupe possède un nom. Un utilisateur peut changer de groupe par la commande **newgrp**. On peut donner des droits d'accès particuliers à certains fichiers pour les membres d'un même groupe.

Au total, il existe 3 types d'accès : propriétaire (**u** = user), groupe (**g** = group), autres (**o** = other). L'ensemble est désigné par **a** (all).

L'expression des droits nécessite 12 bits :



Pour un fichier, **r** = droit de lecture, **w** = droit d'écriture, **x** = droit d'exécuter (1 = oui, 0 = non)

Pour un répertoire,

r :on peut consulter la liste des fichiers qui y sont contenus
w : on peut créer ou effacer des fichiers à l'intérieur
x : on peut utiliser ce répertoire en argument d'une commande et s'y positionner

Ces 9 bits sont précédés par 3 autres bits pour compléter la description des protections :

- **le bit "set-uid" :** quand il vaut 1 pour un fichier exécutable, le processus d'exécution a les droits du propriétaire du fichier et non de l'utilisateur qui le lance

- **le bit "set-gid" :** même rôle, mais relativement au groupe

- **l'attribut spécifique :** outre la valeur "-" pour les fichiers ordinaires, il peut prendre la valeur :

d fichier répertoire
c fichier spécial en mode caractère (cas des terminaux)
b fichier spécial en mode bloc (cas de lecteurs de disques ou de bandes)
s bit de collage ou "sticky-bit" : il maintient le fichier en zone de recouvrement (swap) à partir de laquelle le chargement est plus rapide

La commande **ls** avec l'option **l** permet d'afficher les protections de fichiers. La commande **chmod** permet de modifier les protections d'un fichier.

Exemple : *d rwx rwx - - - 139 pagnotte profess 352 Nov 25 1999 tp*
tp est un répertoire (d)
Son propriétaire est *pagnotte*, du groupe *profess*
les protections rwx rwx - - - sont à interpréter selon les indications ci-dessus

La commande **ls - ias** permet l'affichage succinct des numéros d'i-nodes du répertoire.

2.4 SGF mono- ou multi-volumes

Un SGF UNIX peut être subdivisé en plusieurs disques logiques appelés "**filesystem**" ou "volume" (cas d'un seul disque partitionné, cas de plusieurs disques sur la même machine ou cas d'un système de fichiers réparti sur plusieurs machines). Chaque *filesystem* comporte une arborescence de fichiers

à partir de sa propre racine. L'un d'eux est appelé "**root filesystem**" ; il fédère les autres arbres. Cet "accrochage" est réalisé par l'administrateur du système grâce à la commande **/etc/mount** : on dit qu'une arborescence est *montée* sur le SGF.Un n° d'i-node peut alors ne plus être unique : il doit être obligatoirement associé au n° du volume.

Le système gère une table des volumes "montés" avec deux copies : l'une en mémoire centrale, l'autre sur disque (fichier */etc/mttab* sur UNIX System V).

L'ensemble des informations sur disque associées à un arbre est divisée en 4 zones :

- **zone 0** : pas utilisée par le SGF, mais par UNIX lui-même. C'est le **bootstrap**, contenant la procédure d'initialisation du système, de montage/démontage des volumes.

- **zone 1** : elle est appelée **superbloc**. Elle contient des informations sur le S.G.F. :
  - caractéristiques du SGF (nom du volume, nom et type du SGF, état du SGF pour savoir si le système a bien été arrêté normalement, date de mise à jour, taille en blocs du SGF, etc...)
  - caractéristiques des blocs libres (nombre, liste partielle, index vers le tableau des blocs libres, verrou d'accès aux blocs libres, etc...). Le superbloc est défini dans **/usr/include/sys/fs/s5param.h**
- **zone 2** : appelée **i-list** (liste d'index). Elle contient les i-nodes du volume.
  Pour augmenter la performance, le superbloc et la i-list sont copiés en mémoire et sauvés périodiquement sur disque.

- **zone 3** : contient les fichiers proprement dits.

Le *root filesytem* contient un certain nombre de répertoires standards :

/bin,/usr/bin utilitaires standards
/dev fichiers spéciaux pour périphériques (cf. ci-dessous)
/etc commandes, fichiers de maintenance/administration du système
/lib et /usr/lib bibliothèques standards
/unix le noyau du système
/usr/adm les fichiers de comptabilité
/tmp, /usr/tmp fichiers temporaires.
/usr/include fichiers en-têtes, d'extension .h
/usr/src les sources du système

L'adresse de sa racine figure "en dur" dans le noyau du système. Cette référence servira à construire la totalité de l'arbre des i-nodes, à contrôler les appels système de recherche, de création et de modification des i-nodes.

La *variable d'environnement* **HOME** mémorise le chemin du répertoire courant de l'utilisateur.

3. LES COMMANDES

Il existe de nombreuses commandes de base décrites dans la section 1 du Manuel de référence ou accessibles par la commande

man nom_de_commande

On trouvera ci-après les principales . Leur syntaxe n'est pas homogène. Elle est souvent de la forme :

nom [-options] [arguments]

Toute commande donne une **valeur de retour** nulle si le résultat est sans erreur ou vrai, non nulle sinon.

Un **tube** | permet de relier la sortie standard d'une commande à l'entrée standard de la suivante. Un tube retourne la valeur de retour de sa dernière commande
exemple :

```
ls | wc
ls | grep  pa | wc - l
#  on recherche les lignes du  répertoire contenant pa et  on affiche leur nombre)
who | tee  /tmp/g
# les informations transmises à la sortie standard le sont aussi au fichier cité, ici  /tmp/g
```

Une **liste** est une séquence de commandes ou de tubes séparés par des ; ou && ou || ou & et terminée par ; ou & Ainsi :

liste1 ; liste2 produit une exécution séquentielle des listes.

*liste1***&&** *liste2* : liste2 est exécutée si la valeur de retour de liste1 est VRAI (0)

liste1 || *liste2* : liste2 est exécutée si la valeur de retour de liste1 n'est pas VRAI (1)

Une liste entre des ( ) est exécutée dans un nouvel environnement, c'est à dire un processus shell fils créé à cet effet

Substitution de commande : **$(commande)**

```
$ echo la date est $(date)
la date est Mon Oct 14 21:32  MET 1999
```

Les alias : un alias est un surnom de commande

alias [-x] [nom[=valeur]]
-x : l'alias est exporté (réutilisable)
aucun **nom** : la liste des alias est visualisée
pas de **valeur** : la valeur de l'alias désigné est visualisée
Exemple : $ alias la="ls -ia"
Pour supprimer un alias, **unalias nom**

alias permet d'abréger des noms de commandes longs ou fréquemment utilisés

at **at** *18 mar 15 fic1* lance l'exécution de fic1 le 15 mars à 18 heures pour les utilisateurs répertoriés dans **/usr/lib/cron/at.allow** et non répertoriés dans **/usr/lib/cron/at.deny**

basename *chaîne [suffixe]*
basename */usr/include/sys/sgtty.h* affiche sgtty.h
basename */usr/include/sys/sgtty.h .h* affiche sgtty

cal *[mois] année* affiche le calendrier de l'*année*, ou seulement du *mois*.

cancel *ident_req* annule la requête d'impression *ident_req* obtenue par la commande **lp**

cat **cat** *fic1 fic2 > fic3* concatène fic1 à fic2 dans fic3 (*cat = catenate and print*)
cat *fic1* concatène fic1 à rien et redirige par défaut sur la sortie standard, donc affiche le contenu de fic1

cat > *fic2* concatène l'entrée standard à rien et la redirige sur fic3, donc saisit dans fic3 [ré]initialisé ce qui est frappé au clavier, jusqu'à la frappe de <EOF> (CTRL-D par défaut).

cd *dir* (*change directory*) change le répertoire courant pour *dir*. En l'absence d'argument spécifié, dir = HOME

chgrp *group filenames* change le groupe des fichiers *filenames* pour *group*.
L'option **-R** effectue un changement récursif sur toute une arborescence de racine donnée

chmod *mode filename* affecte la protection *mode* à *filename* (cf. ci-dessus).
chmod *544 fic1* affecte la protection r-xr--r-- à *fic1*
chmod *g+x fic1* transforme cette protection en r-xr-xr--
chmod *a+w fic1* donne à tout le monde le droit d'écriture dans *fic1*
chmod *u-w fic1* retire ce droit au propriétaire.

chown *owner filenames* change le propriétaire des fichiers *filenames* pour *owner*.
L'option **-R** effectue un changement récursif sur toute une arborescence de racine donnée

cmp *filename1 filename2* donne le n° de l'octet et de la ligne où se produit la première différence entre *filename1* et *filename2*.

cp **cp** *filename1 filename 2* copie *filename1* sur *filename2*
cp -r *directory1 directory2* copie récursivement *directory1* avec son sous-arbre dans *directory2*, en le créant s'il n'existe pas.
cp *filenames directory* copie les *filenames* dans *directory*.
-i demande confirmation
-s produit simplement un code de retour (0 : identiques, 2 : différents, 3 : erreur)

crontab Le processus système **cron**, créé à la génération du système, gère toutes les tâches périodiques, y compris de l'arrivée du courrier dans les boîtes aux lettres. La commande **crontab** permet à un utilisateur de rajouter, de lister ou de retirer un fichier de tâches périodiques (batch) :
crontab *filename* ajoute le *filename* de tâches périodiques
crontab -*l* liste le fichier de tâches périodiques de l'utilisateur
crontab -*r* retire le fichier de tâches périodiques de l'utilisateur

Un fichier de tâches périodiques comprend une ligne par tâche. Chaque ligne comprend 6 champs séparés par des blancs ou des TAB , * spécifiant n'importe quelle valeur :
les minutes (0-59) de l'heure de lancement,
l'heure (0-23) de la date de lancement,
le jour de lancement (ou les jours de lancement) dans le mois (0-31),
le mois de lancement
le jour de lancement dans la semaine (0-6, 0=dimanche),
le texte de la commande à lancer
*0 14 * * 1-5 fic1* exécutera *fic1* chaque semaine du lundi au samedi à 14 h.
*0 0 1,15 * 1 fic2* exécutera *fic2* chaque lundi, et chaque 1er et 15 de chaque mois à 0 h.

crypt *chaîne* crypte l'entrée standard avec la clé chaîne ou la décrypte

cut permet d'extraire des caractères ou des champs de chaque ligne d'un fichier.
ls|cut -c*1,4,7* extrait les caractères 1, 4 et 7 de la sortie de ls
ls|cut -c*1-3,8* extrait les caractères 1 à 3, et 8 de la sortie de ls
l'option **-f** permet de spécifier des champs au lieu de caractères
l'option **-d***c* permet de respécifier le délimiteur de champ

(TAB par défaut, nouvelle valeur *c*)

date affiche la date

du **du** affiche le nombre de blocs (512 ou 1024 octets) occupés par chaque sous-arbre du répertoire courant
du -a affiche cette information pour tout fichier, tout répertoire du répertoire courant.
du -s affiche seulement le total

echo *chaîne_de_caractères* affiche *chaîne_de _caractères* sur l'écran.
echo "Votre nom \c" le curseur ne va pas à la ligne

expr interprète les arguments pour des calculs numériques. Voir exemple plus loin

file *filename* affiche la nature présumée du contenu de *filename*

find *pathname_list expression* parcourt récursivement le sous-arbre dont le chemin d'accès est *pathname_list* en recherchant les fichiers satisfaisant *expression*.
find */ -name fic1 -print* recherche *fic1* dans toute l'arborescence.
find */ -user gr1605 -print* affiche la liste des fichiers appartenant à gr1605
find *source -perm 777 -type f -exec rm {}\;* recherche les fichiers ordinaires (*-type f*; b désignerait un fichier de type bloc, c un fichier de type caractère et d un répertoire) dans le sous-répertoire *source*, les efface (après -exec, on peut utiliser n'importe quelle commande en la faisant suivre de "\;"). Les accolades {} remplacent le nom du fichier trouvé

grep *[-civw] pattern filenames*
"globally find regular expressions and print". Les *pattern* sont des expressions régulières de **ed** ou **sed**. Les lignes satisfaisant à ces critères sont affichées. Voir exemple ci-dessus.
Les options :
-c (count) compte les lignes sans les afficher
-i considère les majuscules et minuscules de façon identique
-v (reverse) cherche les lignes ne correspondant pas au critère
-w (word) cherche *pattern* comme un mot isolé

head *[-n] filename* affiche les *n* premiers caractères de *filename* (10 par défaut)

id affiche le nom et le n° (UID) de l'utilisateur, son nom et son n° de groupe (GID)

ln *filename1 filename2* crée un nouveau lien de nom *filename2* pour le fichier *filename1*
Il n'y a ni copie du fichier, ni création d'un nouvel i-node, mais simplement augmentation du compteur de référence du fichier dans un répertoire, c'est à dire ajout d'un couple (n°, référence)

logname affiche l'identification de l'utilisateur (valeur de la variable d'environnement LOGNAME)

lp *filename* ajoute la requête d'impression de *filename* à la queue du spoule

lpstat affiche des informations sur l'imprimante et son spoule

ls *[-alR] filename* affiche le contenu du répertoire *filename*.
Sans argument, le répertoire courant est traité. Les principales options :
-a permet l'affichage des noms de fichiers commençant par .
-l format long. Sont affichés : nombre de liens, propriétaire, taille en octets, date de dernière modification, mode (d = répertoire, b = fichier bloc, c = fichier caractère, l = lien symbolique, p = tube nommé, - = fichier ordinaire)

-i affiche les numéros d'i-nodes
-R récursif, génère la liste du sous-arbre tout entier

mkdir *dir* crée un nouveau répertoire.
L'option **- m** *droits* crée un nouveau répertoire avec les *droits* spécifiés

more *filename* affiche le contenu du fichier texte *filename*, un écran complet à la fois. Si l'on frappe :

ESPACE une ligne supplémentaire est affichée
RETURN un nouvel écran est affiché
b l'écran précédent est affiché
h affiche l'aide (liste complète des options)
q on quitte more

mv **mv** *filename1 filename2* change le nom (lien) *filename1* en *filename2*.
mv *directory1 directory2* change le nom *directory1* en *directory2* si *directory2* n'existe pas déjà et si *directory1* et *directory2* ont le même père.
mv *filenames directory* crée un lien entre les *filenames* et le *directory*

paste *filename1 filename2* concatène les lignes de même numéro de *filename1* et de *filename2*

pwd (*print working directory*) affiche la référence absolue du répertoire courant

rm *[-ri] filenames* efface les liens des *filenames* dans le répertoire courant, si le droit d'écriture dans le répertoire existe.
Le contenu d'un fichier qui n'a plus de lien est perdu. Options :
-i interactif. Demande une confirmation pour chaque fichier.
-r récursif pour tout le sous-arbre, si l'argument est un répertoire

rmdir *dir* supprime le répertoire *dir* s'il est vide
-i avec confirmation

rpl *chaine 1 chaine 2 < fic1 > fic2*
remplace toutes les occurences de chaine1 par chaine2 dans fic1 et émet dans fic2
ex.: rpl " IT " "Italie" < films.cine > films.tele

sort *[options] [+n1 -n2] filename1 [-o filename2]*
trie, selon l'ordre lexicographique du code, les lignes de *filename1*, affiche le résultat ou le redirige sur *filename2*. Options :
-b on ignore les espaces de tête
-d seuls les chiffres, lettres et espaces sont significatifs dans les comparaisons,
-f majuscules et minuscules sont confondues,
-i les caractères dont le code ASCII est extérieur à l'intervalle [32,126] sont ignorés dans les comparaisons,
-n les débuts de lignes numériques sont triés numériquement,
-t*c* définit comme *c* le séparateur de champs au lieu de TAB

Les options +n1 -n2 délimitent la clé : du champ n° n1 inclus au champ n° n2 exclu. La numérotation des champs commence à 0.
sort permet aussi le tri sur des portions de champs, sur plusieurs clés, ou des fusions de fichiers (voir Nebut p. 135-137)

split -*number infile outfile* fragmente le fichier *infile* en fichiers de *number* lignes (1000 par défaut), de noms *outfile*aa, *outfile*ab, etc... (*outfile* = x s'il n'est pas spécifié)

stty [-a] *options*

L'utilisation des terminaux fait appel à 10 caractères particuliers, désignés par un nom symbolique :
<NEWLINE> : dans l'utilisation normale bufferisée, le processeur d'E/S reçoit un caractère du clavier, envoie un écho à l'écran. Mais il n'envoie les caractères au processeur central qu'après réception de <NEWLINE>. Par défaut le caractère NL du code.
<ERASE> : effacement du caractère précédent. Par défaut :# ou le caractère BS du code ou CTRL-H
<KILL> : effacement de la ligne courante. Par défaut @ ou CTRL-U
<EOL> : marqueur de fin de ligne. Par défaut \0 ou CTRL-@
<EOF> : marqueur de fin de fichier s'il est en début de ligne. Par défaut le caractère EOT du code ou CTRL-D.
<STOP> : interrompt le flux de caractères vers la sortie. Par défaut le caractère DC3 du code ou CTRL-S.
<START> : relance le flux de caractères vers la sortie. Par défaut le caractère DC1 du code ou CTRL-Q.
<INTR> : émission du signal SIGINT. Par défaut le caractère DEL du code ou CTRL-C.
<QUIT> : émission du signal SIGQUIT. Par défaut CTRL-\ ou CTRL-Z

stty *caractère_de_contrôle c* donne la nouvelle valeur *c* au *caractère_de_contrôle* avec la syntaxe '^D' pour CTRL-D et '^?' pour DEL

stty -a affiche les options actuelles de la sortie standard.
stty sane rétablit les valeurs par défaut

tail *[options] filename* affiche la fin de *filename*.
tail *-14c fic1* affiche *fic1* à partir du 14ème caractère avant la fin
Au lieu de c, on peut utiliser l (ligne) ou b (bloc)
tail *+25l fic2* affiche la fin de fic2 à partir de la 25ème ligne

tee *[-a] filename* l'affichage de la sortie standard est en même temps dirigé sur *filename*
L'option **-a** signifie >>

test *exp* ou **[** *exp* **]** retourne 0 si la valeur de exp est vrai, 1 sinon. exp peut être :
-a<filename> : vrai si filename existe
-r<filename> : vrai si filename existe et peut être lu,
-w<filename> : vrai si filename existe et peut être écrit,
-x<filename> : vrai si filename existe et peut être exécuté,
-f<filename> : vrai si filename existe et est un fichier ordinaire,
-d<filename> : vrai si filename existe et est un catalogue,
-b<filename> : vrai si filename existe et est de type bloc,
-c<filename> : vrai si filename existe et est de type caractère,
-s<filename> : vrai si filename existe et est de taille non nulle,
-z<chaîne> : vrai si la chaîne est de longueur nulle,
-n<chaîne> : vrai si la chaîne est de longueur non nulle,
<chaîne1> = <chaîne2> : vrai si les deux chaînes sont égales,
<chaîne1> != <chaîne2> : vrai si les deux chaînes sont différentes,
<valeur1> -eq <valeur2> : vrai si les deux valeurs sont égales,

autres prédicats possibles : -ne (≠),-gt (>),-lt (<),-ge (≥), -le (≤)
On peut composer les conditions avec les opérateurs booléens **-a** ou && (ET), **-o** ou || (OU), **!** (NON)

tr *string1 string2* l'entrée standard est copiée sur la sortie standard, mais un caractère ayant une occurence dans *string1* est remplacé par le caractère de même rang dans *string2*. Avec l'option **-d**, les caractères en entrée, présents dans *string1*, sont supprimés en sortie.
tr *[a-z][A-Z] < fic1* affiche fic1 en remplaçant les minuscules par des majuscules.

tr *-d '\012' < fic1* affiche fic1 en supprimant (option **-d**) les caractères de code ASCII 12 en octal, donc 10 en décimal, donc en supprimant les sauts de ligne
tr *-s "a-z" "A-Z" < fic1* réduit toute chaîne de caractères identiques en un seul
tr *abc xyz < fic1* traduit a en x, b en y, c en z

tty affiche le nom du terminal associé à la sortie standard

wc *[-lwc] filename*
wc *fic1* affiche le nombre de lignes, mots, et caractères de fic1
wc *-lc fic1* affichera le nombre de lignes et de caractères de fic1
L'option **-w** correspond aux mots, **-l** aux lignes, **-c** aux caractères

whereis *filename* affiche le chemin d'accès à *filename*.

who affiche la liste des utilisateurs connectés

xd *filename* affiche *filename* en hexadécimal et "en clair".
ls|xd affichera les noms complets des fichiers du répertoire courant, y compris les caractères non affichables qui pourraient être présents dans les noms de fichiers

4. LE SHELL

Le shell (littéralement coquille autour du noyau d'UNIX) est l'interpréteur de commandes d'UNIX. Tout à la fois :
- il exécute en mode interactif les commandes émises par l'utilisateur,
- il propose un langage de programmation interprété permettant de générer de nouvelles commandes ou procédures cataloguées ("scripts shell"), C étant le langage le plus adapté pour construire les nouvelles commandes que le shell ne peut traduire.

Le shell ne fait pas partie du noyau d'UNIX et n'est pas résident en mémoire principale. Ainsi, on peut disposer facilement de plusieurs interpréteurs de commandes : Bourne-shell, C-shell, Korn-shell, ...

Dans tout shell d'UNIX, un **blanc** est un espace ou un caractère de tabulation. Un **séparateur** est un blanc ou un caractère NL. Un **mot** est une chaîne de caractères entre deux séparateurs. Tout ce qui est entre deux ' est considéré comme un seul mot.

4.1 Les variables shell

leur ***nom*** : une suite de caractères lettres, chiffres et _ commençant par une lettre ou _
ex.: a=paul chemin=/users/eleves/m-durand99

leur ***valeur*** : $a ou ${a} désigne la valeur de la variable a et ${a}c désigne la valeur de a suivie de c.
ex.: a=paul
b=chou
echo $a $b

On utilise trois ***caractères génériques*** :
* toute sous-chaîne, même vide,
? tout caractère,
[...] tous les caractères d'un intervalle.
Toute fin de ligne commençant par # est un commentaire

ex.: ls - l *.c # édite les noms de fichiers d'extension .c

ls -l [a-e]* # édite les noms de fichiers commençant par a à e.

métacaractères : < * ? | & , \ ont un sens spécial.

ex.: a="bijou * caillou "
b=chou ; c=caillou ; r="$a $b";echo $r

Précédés de \, les métacaractères perdent leur signification particulière

ex.: echo * ; echo \\ echo abc***d

les ***délimiteurs de chaînes*** :

dans une chaîne délimitée par des " , les caractères $, \, ', ` sont des caractères spéciaux.

dans une telle chaîne, un caractère " doit être précédé de \

dans une chaîne délimitée par des ' , tous les caractères perdent leur aspect spécial

types de variables : la commande **typeset [-filrux] [variable [= valeur]]** permet de typer une variable

et/ou de lui affecter une valeur:

-f : variable est une fonction
-i : variable est de type entier
-l : majuscules transformées en minuscules
-r : variable accessible seulement en lecture
-u : minuscules transformées en majuscules
-x : variable exportée

Exemples :

```
$ typeset -r v=abc
$ echo $v
abc
$ typeset v=hier
/bin/ksh  v is read only
$ w=3+5
$ echo $w
3+5
$  typeset -i w
$ echo $w
8
$ typeset +r  v      (déprotection de v)
$ v=123
$ echo $v
123
```

règles de substitution : elles utilisent les { }

${#variable} : longueur de la variable

```
$ X=abc
$ print ${#X}
3
```

${variable :- mot} : valeur de *variable* si elle est définie et non nulle, sinon valeur de *mot*

```
$ X=abc
$ print ${X-:cde}
abc
```

```
$ unset X
$ print ${X:-cde}
cde
```

${variable := mot} : valeur de *variable* si elle est définie et non nulle, sinon *variable* prend la de *mot*

${variable :? mot} : valeur de *variable* si elle est définie et non nulle, sinon valeur de *mot* et sortie

${variable :+ mot} : valeur de mot si *variable* est définie et non nulle, sinon pas de substitution

```
$ Y=abc
$ print ${Y=+def}
def
$ unset Y
$ print ${Y:+def}
$ .....
```

${variable#pattern}** ou **${variable##pattern} : valeur de *variable* à laquelle on extrait en partant de la gauche la plus petite partie (ou la plus grande) correspondant au *pattern*

```
$ X=abcabcabc
$ print ${X#abc*}
abcabc
 $ print ${X##abc*}
$  (rien)
```

${variable % pattern}** ou **${variable %% pattern} : idem à ci-dessus, mais en partant de la droite

les **tableaux** de variables : en assignant à X [1] une valeur, la variable X est *transformée* en tableau

```
$ X=A
$ X[1]=B     (alors, X [0] vaut A )
```

On peut **affecter** ainsi :

variable[0]=valeur0 ; variable[1]=valeur1 ; .... ; variable[n]=valeurn
ou encore : ***set*** -A variable valeur0 valeur1 ..... valeurn
ou encore :
typeset variable[0]=valeur0 variable[1]=valeur1...variable[n]=valeurn

On peut **réaffecter** ainsi : ***set*** -A variable valeur0 valeur1 ..... valeurn
ou encore en ne donnant que certaines valeurs :

```
$ set -A  X  one  two  three  d  e  f
$ print ${X[*]}
one  two  three  d  e  f
$ set -A  X  a  b  c
$ print ${X[*]}
a  b  c  d  e  f
$ print ${X[1]}
b
$ print ${#X[*]}
6      (nombre d'éléments du tableau)
```

variables prédéfinies gérées automatiquement par le shell :

$# nombre de paramètres d'une commande, ceux-ci étant désignés par $1 à $9

($0 le nom de la commande elle-même).

$* la liste des paramètres $1 $2 ...

$$ le numéro du processus en cours (très utile dans la suite)

$! le n° du dernier processus lancé en arrière plan

$? le code de retour de la dernière commande exécutée

variables d'environnement prédéfinies :

HOME l'argument par défaut pour la commande cd, c'est à dire le chemin correspondant au sous-répertoire à l'ouverture de la session
LOGNAME le nom de l'utilisateur (sous SYSTEM V) ou ***USER*** (sous système BSD)
PATH la liste des répertoires à chercher pour exécuter une commande
PS1 le prompt principal
PS2 le prompt secondaire
IFS la liste des séparateurs de mots pour le shell (inter fields separators)
SHELL le shell courant
TERM le type de terminal
TZ le fuseau horaire (time zone)

Le fichier **.profile** permet de donner des valeurs non standards aux *variables d'environnement* et d'exécuter des commandes comme :

uname -a (nom de la machine)
umask (masque des autorisations d'accès au fichier par défaut)
ulimit (taille maximale des fichiers)
banner (affichage d'une bannière en grandes lettres).

La commande **set** permet d'afficher la valeur des *variables d'environnement*

Exemple d'un fichier *.profile* :

```
HOME=/home/$LOGNAME
PATH=$PATH:$HOME/mes_tp
PS1="OK ?"
banner  HELLO WORLD
export  HOME PATH PS1
```

4.2 Les shell scripts

L'interpréteur de commandes est un fichier exécutable se trouvant dans /bin. Le shell est donc lui-même une commande.

Ainsi, pour disposer d'une nouvelle procédure cataloguée ("***script shell***"), on saisit dans un fichier (par exemple ici *fic*) une liste de commandes avec un éditeur

Pour exécuter ce fichier, on a deux solutions:

sh fic [arg] ou mieux **chmod u+x fic**
fic [arg]

Le langage de commandes du shell dispose des structures classiques if, for, while, until.

4.3 Boucle for

```
for var  [in  mot1 [mot2 ...]]
            do  liste_de_commandes
            done
```

var prend les valeurs *mot1, mot2*, ... et exécute la liste de commandes pour chaque valeur

```
ex.:   for i do grep   $i   /users/eleves/m-durand99/telno
           done

# on range ce script dans le fichier TEL qu'on rend exécutable
# ici, la liste des valeurs est vide; par défaut,
# elle est prise égale à $*, la liste de paramètres de commande

$ TEL  paul  bernard
# affichera les lignes de /users/eleves/m-durand99/telno contenant paul  ou bernard
```

4.4 Boucle while

```
while liste_de_commandes
    do liste_de_commandes
    done
```

On boucle tant que la dernière commande de la liste renvoie 0 (vrai).

```
    ex.:   while test $#  -ne 0              # tant qu'il y a des paramètres...
              do echo $1                     #...on affiche le premier
                 shift                       # on décale les paramètres
              done
           # on range ce script dans le fichier ACTION qu'on rend exécutable

           ACTION  toto   titi          #  affiche toto titi
```

4.5 Boucle until

```
until  liste_de_commandes
            do liste_de_commandes
            done
```

On boucle jusqu'à ce que la dernière commande de la liste renvoie 0 (vrai).

```
ex.:        until test -f  fic
             do sleep 50       #attendre 50 s.
              done
```

4.6 Choix if

```
if  liste_de_commandes
```

```
then  liste_de_commandes
        [elif  liste_de_commandes
           then  liste_de_commandes ]
           [else liste de commandes ]
fi
```

On exécute "then" si la dernière commande de la liste de "if" retourne 0 (vrai).

```
ex.:    if test -f  "$1"
           then echo fichier  $1 trouve
           else echo  pas de fichier de nom  $1
        fi
```

4.7 Sélection case

case vise à éviter les imbrications de if :

```
ex.:    # fichier ajout
         case $# in
           1) cat  >> $1;;
           2) cat  >> $2  <$1;;
          *) echo  "syntaxe : ajout [origine] destination";;
         esac

        ajout  fich      #ajoute en fin de fichier jusqu'à <CTRL-D>
        ajout  f1  f2    # f1 copié en fin de f2
```

On notera aussi les commandes **break** et **continue** (idem à C). C-Shell contient aussi **foreach**.

4.8 Sélection select

select est une commande intermédiaire entre case et for, utile pour la gestion des menus

```
ex.:    # fichier stest
        select i in Choix-A  Choix-B  Choix-C
        do
            if  [$i  = Choix-[A-C]]
            then
                print "Vous avez choisi  $REPLY : $i"
            else
                 print "$REPLY : mauvais choix"
                 continue
            fi
        done

        $ stest
        1) Choix-A
        2) Choix-B
        3) Choix-C
        #?  5
        5 : mauvais choix
        #?  3
        Vous avez choisi : 3
```

```
#? <RETURN>              # menu réaffiché
.............
```

4.9 Utilisation du résultat d'une commande

On peut utiliser comme paramètre, ou affecter à une variable, la sortie standard d'une commande placée entre ` `

exemples 1 :
```
for i in `ls`
    do echo "      "$i
    done
```

2 :
```
c=`pwd`    # voir commande pwd
  ls $c    # affiche le catalogue de travail
```

3 :
```
set  `date` ; echo $3 $2 $6
  # set range dans les variables prédéfinies $1, $2, etc....
  # il s'affiche jour, mois année
```

4.10 Autres commandes

read
```
echo "Votre nom ? \c"
read name
echo "Bonjour  $name, ca va ?"

print "this is a play again" | read X  Y  Z
print $X
this
print $Y
is
print  $Z
a play again
```

exec réalise des redirections d'E/S
```
exec 1 > std.out      # redirige la sortie standard vers le fichier std.out

exec < fic            # redirige l'entrée standard sur le fichier fic
.......
exec < /dev/tty       # rétablit l'entrée standard

exec 5<>fic.out
# le fichier fic.out est ouvert en lecture-écriture avec le descripteur 5
print  -u5  "Tout va bien"
cat  < &5                 # lecture à partir du fichier de descripteur 5
exec 5 < &-               # fermeture du fichier de descripteur 5
```

let let "expression arithmétique" ou ((expression arithmétique))

exemples :
```
 ((X=-7))
 ((Y=-X+2))
 echo $Y
```
toutes les opérations arithmétiques de C sont permises avec let

true retourne 0

false retourne 1

5. LE NOYAU D'UNIX

Le noyau est responsable de presque toutes les tâches de base qui sont exécutées par des procédures système :

- **initialisation du système** ,
- **gestion des ressources :** temps, mémoire primaire,

- **gestion des fichiers :** création, utilisation, disparition, gestion de la mémoire secondaire,

- **gestion de la mémoire :** gestion de la segmentation et de la pagination.
- **gestion des processus :** contrôle de la création, de la terminaison, de la synchronisation, du partage de temps (ordonnancement), de la communication entre processus,

- **gestion des E/S :** sélection des pilotes (drivers) pour contrôler l'E/S en fonction du n° majeur et du n° mineur du fichier spécial sélectionné,

- **gestion des communications :** réseaux locaux, X25, etc...

Pour assurer ces activités, le noyau dispose d'un certain nombre de tables :
- table des processus décrivant l'état des processus,
- table des fichiers ouverts,
- table des i-nodes en mémoire,
- pour chaque processus, une table des descripteurs de fichiers,
- table des volumes montés,
- table des textes (pour les exécutables partageables)

Le noyau réside en mémoire principale tant que le système est en fonctionnement.

BIBLIOGRAPHIE

S.BOURNE , Le système UNIX, Interéditions 1985
J.P. BRAQUELAIRE, Méthodologie de la programmation en Langage C, Masson, 1993
A.B. FONTAINE, Ph. HAMMES, UNIX Système V, Masson, 1993
B. KERNIGHAN et R. PIKE , L'environnement de programmation UNIX, Interéditions, 1985
J.L. NEBUT , UNIX pour l'utilisateur, Technip, 1990
J.M. RIFFLET , La programmation sous UNIX, Mac Graw-Hill, 1992
A. TANENBAUM, Systèmes d'exploitation, Prentice Hall, 1999

ANNEXE : QUELQUES UTILITAIRES

1. ar

ar permet de construire et de mettre à jour des bibliothèques utilisées par l'éditeur de liens **ld**.
On peut ainsi rassembler plusieurs modules objets en un seul fichier d'archive.

Syntaxe : **ar** *clé lib liste_ref* avec :

lib nom du fichier archive (la bibliothèque),
clé **d** supprimer la liste de l'archive
r si l'un des modules de la liste est dans l'archive, il sera remplacé; sinon, il sera créé
q ajouter de nouveaux modules à la fin de l'archive, sans examiner l'existant
t ou **p** lister le contenu de l'archive
x extraire des modules de l'archive
liste_ref liste de fichiers objets

2. awk

awk (du nom de ses créateurs, Aho, Weinberger et Kernighan), est un micro-langage de programmation, accessible par le shell d'UNIX, inspiré de C et interprété.

appel : awk 'programme' *nom_fichier* **ou** awk -f *refprog nom_fichier*
où ***refprog*** est le fichier contenant le programme
nom_fichier est le fichier de données pour le programme ou une liste de fichiers

Un programme est de la forme :

```
BEGIN {.....}
{ .....}
END { .....}
```

Les sections BEGIN et END sont facultatives.

Dans le corps du programme (section 2), on peut utiliser la syntaxe complexe de *grep* pour exprimer des expressions ("motifs").

identificateurs : comme en C, mais pas de déclarations. Les types flottant et chaîne de caractères sont implicites.

Des variables sont prédéfinies :
$0 : l'enregistrement courant du fichier
$1, $2, ... : ses champs
FS : séparateur de champs (SP et HT par défaut)
RS : séparateur d'enregistrements (LF par défaut)
OFS : séparateur de champs en sortie (SP par défaut)
ORS : séparateur d'enregistrements en sortie (LF par défaut)
NF : nombre de champs de $0
NR : numéro de l'enregistrement courant $0

opérateurs : + - * / % ++ - < > <= >= == != ! && ||

fonctions : *length* (ch) retourne la longueur de la chaîne *ch*

int (exp) convertit *exp* en entier par troncature
index (ch1,ch2) retourne la position de la première occurence de *ch2* dans *ch1* ou 0 s'il n'y en a pas
split (ch,t,c) éclate *ch* dans *t[1], t[2],* ... avec *c* comme séparateur de zones.
substr (ch,m,n) retourne la sous-chaîne de *ch* commençant en position *m* et de longueur au-plus *n*

instructions :

if else comme en C
while comme en C
for comme en C
for (<var> in <tableau>) <instruction>
break continue comme en C
exit passage à la section END, ou abandon si on s'y trouve
next passage à l'enregistrement suivant

On dispose aussi de la fonction *printf* de C et de *print* (cf. exemple).

On peut passer la valeur d'une variable du shell à une variable locale d'un programme awk:

read reponse
awk ..... x=$reponse ....

en supposant que *x=$reponse* est en tête de la liste ***nom_fichier*** décrite plus haut, précédée de -v

Un commentaire commence par # et finit à la fin de la ligne.

3. make

make est un outil puissant pour gérer les projets. Il permet la maintenance des fichiers objets et des fichiers exécutables en lançant les seules compilations indispensables après des mises à jour.

make utilise un fichier appelé *makefile* par défaut qui contient une description des liens de dépendance entre les fichiers et les actions à réaliser.
Exemple :

```
prog:fic1.o fic2.o
        cc -o prog fic1.o fic2.o
fic1.o:fic1.c menu.h
        cc -c fic1.c
fic2.o:fic2.c
        cc -c fic2.c
```

Il peut y avoir plusieurs points d'entrée (tels que *prog*) dans *makefile*.
Les lignes indentées sont précedées de TAB.

Syntaxe : **make** *prog*
make *-f file prog* si *file* est utilisé au lieu de *makefile*

4. sccs

sccs permet le contrôle des fichiers source et de leurs différentes versions, notamment extraire une copie d'une version à partir d'un fichier historique, verrouiller une version contre tout changement pour la protéger, éditer les différences entre les versions, etc... (Voir Nebut, pages 223-229).

5. sed (et ed)

sed est un ancien éditeur de texte non interactif reprenant la plupart des commandes du très ancien éditeur ligne **ed** (qui n'a plus d'intérêt pris isolément). La syntaxe d'appel est :

sed -f fc fa > fb
 # crée un nouveau fichier fb à partir d'un fichier fa en prenant les directives dans le fichier de commandes fc

ed offre essentiellement les fonctionnalités suivantes :

- impression :

p imprime (envoie dans stdout) tout le fichier courant
2 p imprime la ligne n° 2
2,5 p imprime les lignes n° 2 à 5
8,$ p imprime de la ligne n° 8 à la fin de fichier
- désigne la ligne précédente, . la ligne courante et + la ligne suivante

- remplacement :

La commande **s/text_anc/texte_nouveau/g** remplace les occurences de **texte_anc** par **texte_nouveau** dans tout (à cause de **g**) le fichier courant.
Le point représente "tout caractère". L'astérisque désigne 0 ou n occurences.
L'accent circonflexe désigne le début de ligne. $ désigne la fin de ligne, [...] désigne un intervalle et [^...] l'intervalle complémentaire, * désigne une répétition de caractères.
Exemples :

1,$s/./,/g	remplace tous les caractères par des virgules
1,$s/\./,/g	remplace tous les points par des virgules
s/x *y/x y/g	remplace tous les blancs entre x et y par un seul blanc
1,$s/^[0-9]*//g	supprime tous chiffres en début de ligne
v/^[0-9]*/d	détruit toutes les lignes ne commençant pas (**v**) par un nombre
g/eau/s/&/vin/	change dans tout le fichier les lignes contenant eau en vin
/sanglots/	recherche la 1ère occurence de **sanglots** vers l'avant du fichier
s/longs/& longs/	remplace dans cette occurence sanglots par sanglots longs
s/.*/(&)/g	enferme entre parenthèses toute ligne du fichier
g?Jean?	recherche toutes les lignes contenant Jean vers le haut du fichier

sed –n " /PAGNOTTE/p " fichier
sed " s/PAGNOTTE/Pagnotte " fic > res

Le critère de recherche **^[^Uu]r..\.** désigne une ligne ne comportant ni (^) U, ni u en début de ligne, mais incluant un r suivi de 2 caractères quelconques et d'un point.

- destruction :

2d détruit la ligne 2
2,5d détruit les lignes 2 à 5
.,/123/d détruit toutes les lignes entre la ligne courante et la 1ère ligne contenant 123

- mouvement :

2,5 t 19	insère une copie des lignes 2 à 5 à partir de la ligne 19
2,5 m 19	déplace les lignes 2 à 5 pour les insérer à partir de la ligne 19
0r fic	insère le fichier fic en début du fichier courant

Chapitre 3 LES PROCESSUS

1. ASPECTS GENERAUX DES PROCESSUS

Un processus est un programme qui s'exécute, ainsi que ses données, sa pile, son compteur ordinal, son pointeur de pile et les autres contenus de registres nécessaires à son exécution. Un processus fournit l'image de l'état d'avancement de l'exécution d'un programme.

Attention : ne pas confondre un processus (**objet dynamique** dont l'exécution peut être suspendue, puis reprise), avec le texte d'un programme, source ou exécutable.

1.1 Simultanéité, ressources

On appelle **simultanéité** l'activation de plusieurs processus au même moment.

Si le nombre de processeurs est au moins égal au nombre de processus, on parle de simultanéité totale ou vraie, sinon de pseudo-simultanéité.

pseudo-simultanéité : c'est par exemple l'exécution enchevêtrée de plusieurs processus sur un seul processeur. La simultanéité est obtenue par commutation temporelle d'un processus à l'autre sur le processeur. Si les basculements sont suffisamment fréquents, l'utilisateur a l'illusion d'une simultanéité totale.

Dans le langage de la programmation structurée, on encadre par les mots-clés **parbegin** et **parend** les sections de tâches pouvant s'exécuter en parallèle ou simultanément.

Avec un SE qui gère le temps partagé (pseudo-parallèlisme par commutation de temps), conceptuellement chaque processus dispose de son propre processeur virtuel. Ainsi, concrètement il n'existe qu'un seul compteur ordinal dont le contenu est renouvelé à chaque commutation. Conceptuellement, tout se passe comme si chaque processus disposait de son propre compteur ordinal.

Dans certains SE, il existe des appels système pour créer un processus, charger son contexte et lancer son exécution. Dans d'autres SE, un processus particulier (INIT sous UNIX) est lancé au démarrage de la machine. Il crée un processus par terminal. Chacun de ces processus attend une éventuelle connexion, et lorsqu'une connexion est validée, il lance un nouveau processus chargé de lire et d'interpréter les commandes de l'utilisateur (Shell sous UNIX). Chacune de ces commandes peut elle-même créer un nouveau processus, etc... On aboutit ainsi à une **arborescence de processus**.

De façon simplifiée, on peut imaginer un SE dans lequel les processus pourraient être dans trois états :

- **élu** : en cours d'exécution. Un processus élu peut être arrêté, même s'il peut poursuivre son exécution, si le SE décide d'allouer le processeur à un autre processus

- **bloqué** : il attend un événement extérieur pour pouvoir continuer (par exemple une ressource; lorsque la ressource est disponible, il passe à l'état "prêt")

- **prêt** : suspendu provisoirement pour permettre l'exécution d'un autre processus

Le modèle processus permet une approche claire du fonctionnement de l'ordinateur : processus utilisateurs, processus systèmes (processus terminaux, processus disques, etc...) qui se bloquent par

défaut de ressource, qui passent de l'état élu à l'état prêt. Dans ce modèle, la couche la plus basse du SE est l'ordonnanceur (scheduler). Il est surmonté d'une multitude de processus. La gestion des interruptions, la suspension et la relance des processus sont l'affaire de l'ordonnanceur.

1.2 Réalisation des processus

Pour mettre en œuvre le modèle des processus, le SE gère une **table des processus** dont chaque entrée correspond à un processus. Chaque ligne peut comporter des informations sur un processus : son état, son compteur ordinal, son pointeur de pile, son allocation mémoire, l'état de ses fichiers ouverts, et d'autres paramètres.

Dans bien des SE, on associe à chaque périphérique d'E/S une zone mémoire appelée **vecteur d'interruptions**. Il contient les adresses des procédures de traitement des interruptions cf. ch. 9). Lorsqu'une interruption survient, le contrôle est donné à l'ordonnanceur qui détermine si le processus élu doit être suspendu ou bien (par exemple, s'il s'agit du processus traitant une interruption plus prioritaire) s'il doit poursuivre son exécution en empilant l'interruption qui vient d'arriver.

1.3 Mécanismes de commutation

- par interruptions

- par trap ou déroutement sur erreur : il s'agit d'une extension du mécanisme des interruptions. En cas de détection d'erreur interne au processus (ex : division par 0, erreur d'adressage), le contrôle est passé au SE en cas d'erreur mettant en cause son intégrité (ex : erreur d'adressage) ou à une fonction de traitement de l'erreur du processus courant.

- appel au superviseur (noyau du SE) : dans un SE multi-utilisateur multi-programmé, toutes les interruptions sont contrôlées par le superviseur. Une raison (pas unique) : l'événement qui interrompt un processus est peut être destiné à un autre; comme le SE doit garantir l'indépendance des processus, c'est lui qui doit récupérer l'événement pour le transmettre au processus destinataire. Ainsi, en cas d'interruption :

  - on sauve le mot d'état et le contexte du processus en cours et on passe en mode superviseur (ou noyau ou système)

  - le traitement de l'interruption est réalisé soit par le superviseur lui-même (horloge, coupure), soit par un processus spécifique (pilote de périphérique d'E/S), soit par le processus interrompu (erreur interne)

  - on élit un nouveau processus à exécuter (peut-être celui qui avait été interrompu)

2. MODELE DE REPRESENTATION DE PROCESSUS

2.1 Décomposition en tâches

On appelle **tâche** une unité élémentaire de traitement ayant une cohérence logique. Si l'exécution du processus P est constituée de l'exécution séquentielle des tâches T_1, T_2,..., T_n, on écrit :

$$P = T_1\ T_2 ... T_n$$

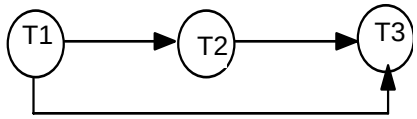
A chaque tâche T_i, on associe sa date de début ou d'initialisation d_i et sa date de terminaison ou de fin f_i.

Une **relation de précédence**, notée <, sur un ensemble E est une relation vérifiant :

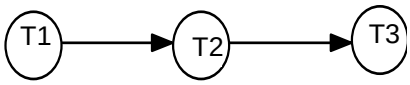
- $\forall$ T $\in$ E, on n'a pas T < T
- $\forall$ T $\in$ E et $\forall$ T' $\in$ E, on n'a pas simultanément T < T' et T' < T
- la relation < est transitive

La relation $T_i < T_j$ entre tâches signifie que f_i inférieur à d_j entre dates. Si on n'a ni $T_i < T_j$, ni $T_j < T_i$, alors on dit que T_i et T_j sont **exécutables en parallèle**.

Une relation de précédence peut être représentée par un graphe orienté. Par exemple, la chaîne de tâches S = ((T_1, T_2, T_3), ($T_i < T_j$ pour i inférieur à j)) a pour graphe :



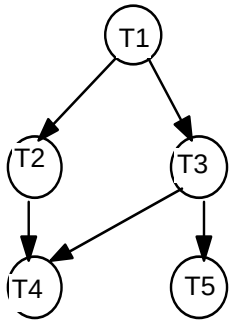
qu'on peut simplifier, en représentant le graphe de la plus petite relation qui a même fermeture transitive que la relation de précédence : c'est le **graphe de précédence** :



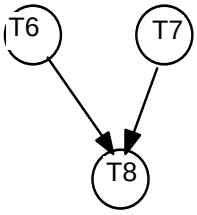
On munit l'ensemble des graphes de précédence de deux opérations :

- la composition parallèle : G_1 et G_2 étant deux graphes de précédence correspondant à des ensembles de tâches disjoints , G_1 // G_2 est l'union de G_1 et de G_2

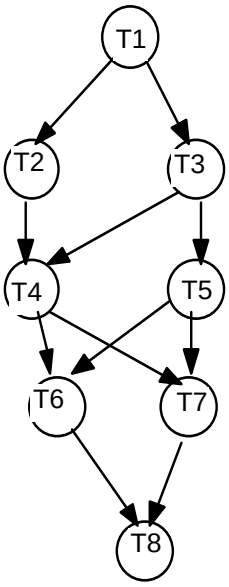
- le produit : G_1 * G_2 reprend les contraintes de G_1 et de G_2 avec en plus la contrainte qu'aucune tâche de G_2 ne peut être initialisée avant que toutes les tâches de G_1 ne soient achevées.



G1



G2



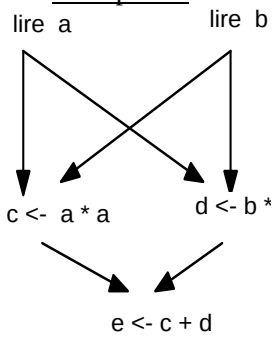
G1 * G2

2.2 Parallèlisation de tâches dans un système de tâches

La mise en parallèle s'effectue par la structure algorithmique :

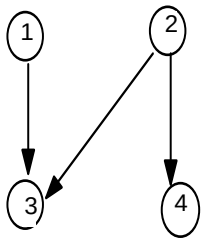
parbegin
.....
parend

exemple 1 :



```
soit   debut
           parbegin
                 lire  a
                 lire  b
           parend
           parbegin
                 c ← a * a
                 d ← b * b
           parend
           e ← c + d
       fin
```

exemple 2 : peut-on représenter le graphe suivant de tâches par un programme parallèle utilisant **parbegin** et **parend** ?



Pour augmenter le degré de multiprogrammation, donc le taux d'utilisation de l'UC, on peut exécuter en parallèle certaines tâches d'un processus séquentiel.

exemple : (système S0)

```
T1    lire  X
T2    lire  Z
T3    X ← X + Z
T4    Y ← X + Z
T5    afficher  Y
```

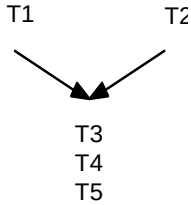
Le système S1 répartit les 5 tâches en 2 processus parallèles organisés chacun séquentiellement :

(S1)

P1	**P2**
T1	T2
T3	T4
T5	

S1 n'est pas équivalent à S0 (affichage pas forcément identique). S1 est **indéterminé**.

(S2)



Et S 2 ?

Méthode d'étude :

Supposons la mémoire composée de **m** cellules : $M = (C_1, C_2, ..., C_m)$

L'état initial du système est $S_0 = (C_1 (0), C_2 (0), ..., C_m (0))$
L'état k du système ($k \in [1 , l]$) est $S_k = (C_1 (k), C_2 (k), ..., C_m (k))$
après l'événement a_k du comportement $w = a_1 a_2 ....a_l$

Par exemple, considérons le problème précédent et $w = d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$
avec M = (X, Y, Z) et $S_0 = (0 , 0 , 0)$

		d_1	f_1	d_2	f_2	d_3	f_3	d_4	f_4	d_5	f_5
X	0	0	α	α	α	α	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$	$\alpha + \gamma$
Y	0	0	0	0	0	0	0	0	$\alpha + 2\gamma$	$\alpha + 2\gamma$	$\alpha + 2\gamma$
Z	0	0	0	0	γ	γ	γ	γ	γ	γ	γ
	0	1	2	3	4	5	6	7	8	9	10

A une tâche T, on associe :
- *son domaine de lecture* : $L = \{ C'_1, ..., C'_p\}$ ou ensemble des cellules lues par T
- *son domaine d'écriture* : $L = \{ C''_1, ..., C''_p\}$ ou ensemble des cellules écrites par T

écrire signifie plus généralement *modifier* l'état d'une ressource.

Dans l'exemple que nous développons, nous obtenons :

L1 = ∅ E1 = { X }
L2 = ∅ E2 = { Z }
L3 = { X , Z } E3 = { X }
L4 = { X , Z } E4 = { Y }
L5 = { Y } E5 = ∅

A une tâche T, on associe une fonction F_T de L dans E

Dans l'exemple que nous développons, $F_{T4} (x , z) = x + z$, x et $z \in N$

Sous-suite des valeurs écrites dans une cellule par un comportement w :

$V (X , w) = (0 , \alpha , \alpha + \gamma)$
$V (Y , w) = (0, \alpha + 2 \gamma)$
$V (Z , w) = (0, \gamma)$

2.3 Caractère déterminé d'un système de tâches. Conditions de Bernstein

Définition : un système de tâches S est **déterminé** si pour tous comportements w et w' et pour toute cellule C de M, on a : V (C, w) = V (C, w')

Avec l'exemple que nous traitons , soient $w = d_1 f_1 d_2 f_2 d_3 f_3 d_4 f_4 d_5 f_5$
et $w' = d_1 f_1 d_2 f_2 d_4 f_4 d_3 f_3 d_5 f_5$
$V (Y , w) = (0, \alpha + 2 \gamma)$ et $V (Y , w') = (0, \alpha + \gamma)$
Donc le système de tâches est non déterminé

Tout système séquentiel est déterminé

Il existe une relation entre le caractère déterminé d'un système et une propriété de non-interférence de tâches.

Définition : deux tâches T et T' sont **non-interférentes** vis à vis du système de tâches S si :

- T est un prédécesseur ou un successeur de T'

ou - $L_T \cap E_{T'} = L_{T'} \cap E_T = E_T \cap E_{T'} = \varnothing$

(conditions de Bernstein)

Avec l'exemple que nous développons, dans le système S1, T3 et T4 sont interférentes car T3 n'est ni prédécesseur, ni successeur de T4 et d'autre part $L_4 \cap E_3 \neq \varnothing$ (vaut { X })

Théorème : Si un système S est constitué de tâches 2 à 2 non interférentes, alors il est déterminé pour toute interprétation

Si un système S est déterminé pour toute interprétation et si $\forall$ T , $E_T \neq \varnothing$, les tâches de S sont 2 à 2 non interférentes.

Nous sommes donc en mesure de construire, à partir d'un système S, un système S' comportant moins de contraintes de précédence. A condition bien sûr que les contraintes d'écriture dans les différentes cellules ne soient pas modifiées.

2.4 Parallélisme maximal

Définition : deux systèmes S et S' construits sur le même ensemble de tâches sont **équivalents** si :

- S et S' sont déterminés

et - pour tout comportement w de S, pour tout comportement w' de S', pour toute cellule C de M, on a : V (C, w) = V (C , w')

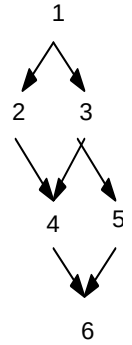
Autrement dit, la suite des valeurs écrites dans toute cellule par tout comportement de l'un ou l'autre système est unique.

Définition : un système S est de **parallélisme maximal** si :

- S est déterminé

et - son graphe de précédence G vérifie : la suppression de tout arc (T, T') entraîne l'interférence des tâches T et T'

Exemple : soit le système S3



M = (M1, M2, M3, M4, M5)

L1 = { M1 }	E1 = { M4 }
L2 = { M3, M4 }	E2 = { M1 }
L3 = { M3, M4 }	E3 = { M5 }
L4 = { M4 }	E4 = { M2 }
L5 = { M5 }	E5 = { M5 }
L6 = { M1, M2 }	E6 = { M4 }

Si l'on supprime l'arc (2 , 4), 2 n'est ni prédécesseur, ni successeur de 4,
mais L2 ∩ E4 = L4 ∩ E2 = E4 ∩ E2 = ∅. Donc 2 et 4 ne deviennent pas interférentes.
Donc le système S3 n'est pas de parallélisme maximal.

Théorème : S = (E, <) étant un système déterminé, il existe un **unique** système S' de parallélisme maximal équivalent à S. S' est tel que : S' = (E , <') avec <' fermeture transitive de la relation :

$$R = \{ (T , T') \mid T < T' \text{ et } (L_T \cap E_{T'} \neq \emptyset \text{ ou } L_{T'} \cap E_T \neq \emptyset \text{ ou } E_T \cap E_{T'} \neq \emptyset) \text{ et } E_T \neq \emptyset \text{ et } E_{T'} \neq \emptyset \}$$

Algorithme : il découle du théorème :

- construire le graphe de R
- éliminer tous les arcs (T , T') redondants, c'est à dire tels qu'il existe un chemin de T à T' contenant plus d'un arc

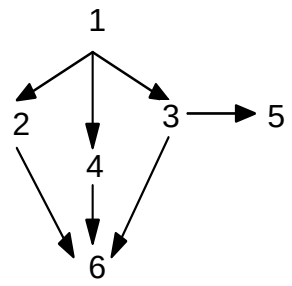
Exemple : avec le système S3 précédent : on démontre qu'il est déterminé

graphe de R : il comporte les arcs :

(1 , 2) car	L1 ∩ E2 = { M1 }
(1 , 3)	L3 ∩ E1 = { M4 }
(1, 4)	L4 ∩ E1 = { M4 }
(1 , 6)	E6 ∩ E1 = { M4 }
(2 , 6)	L2 ∩ E6 = { M4 }
(3 , 5)	E3 ∩ E5 = { M5 }
(3 , 6)	L3 ∩ E6 = { M4 }
(4 , 6)	L4 ∩ E6 = { M4 }

Les arcs (1,5) , (2,4) , (3,4) et (5,6) ne sont pas présents

L'arc (1 , 6), redondant, peut être supprimé.
D'où le système de parallélisme maximal équivalent à S3 :



3. ORDONNANCEMENT DES PROCESSUS

L'ordonnanceur (scheduler) définit l'ordre dans lequel les processus prêts utilisent l'UC (en acquièrent la ressource) et la durée d'utilisation, en utilisant un algorithme d'ordonnancement. Un bon algorithme d'ordonnancement doit posséder les qualités suivantes :

- équitabilité : chaque processus reçoit sa part du temps processeur
- efficacité : le processeur doit travailler à 100 % du temps
- temps de réponse : à minimiser en mode interactif
- temps d'exécution : minimiser l'attente des travaux en traitement par lots (batch)
- rendement : maximiser le nombre de travaux effectués par unité de temps

L'ensemble de ces objectifs est contradictoire, par exemple le 3ème et le 4ème objectif.

On appelle **temps de traitement moyen** d'un système de tâches la moyenne des intervalles de temps séparant la soumission d'une tâche de sa fin d'exécution. On appelle **assignation** la description de l'exécution des tâches sur le ou les processeurs. A chaque tâche T_i du système de tâches, on associe deux réels :

- t_i : sa date d'arrivée
- τ_i : sa durée

Il existe deux familles d'algorithmes :

- **sans réquisition (ASR) :** le choix d'un nouveau processus ne se fait que sur blocage ou terminaison du processus courant (utilisé en batch par exemple)

- **avec réquisition (AAR) :** à intervalle régulier, l'ordonnanceur reprend la main et élit un nouveau processus actif (algorithmes 3.2 à 3.7)

3.1 ASR

On utilise :

- soit l'ordonnancement dans l'ordre d'arrivée en gérant une file des processus. Cet algorithme est facile à implanter, mais il est loin d'optimiser le temps de traitement moyen

- soit l'ordonnancement par ordre inverse du temps d'exécution (PCTE) : lorsque plusieurs travaux d'égale importance se trouvent dans une file, l'ordonnanceur élit le plus court d'abord. On démontre que le temps moyen d'attente est minimal par rapport à toute autre stratégie **si toutes les tâches sont présentes dans la file d'attente au moment où débute l'assignation**. En effet, considérons 4 travaux dont les durées d'exécution sont a, b, c, d. Si on les exécute dans cet ordre, le temps moyen d'attente sera : (4a + 3b + 2c + d)/4. Comme a possède le plus grand poids (4), il faut donc que a soit le temps le plus court, etc...

Exemple : soient 5 tâches A, B, C, D et E de temps d'exécution respectifs 2 , 4 , 1 , 1 , 1 arrivant aux instants 0 , 0 , 3 , 3 , 3. L'algorithme du plus court d'abord donne l'ordre A , B, C , D , E et une attente moyenne de 2,8. L'ordre B , C , D , E , A donnerait une attente moyenne de 2,6

τ_i peut aussi être remplacé par la durée estimée e_i de l'exécution de la tâche T_i. On peut par exemple calculer les durées estimées selon une méthode de moyenne pondérée, à partir d'une valeur initiale e_0 arbitraire :

$$e_i = \alpha * e_{i-1} + (1 - \alpha) * \tau_{i-1}$$, avec souvent $\alpha = 0,5$

3.2 Ordonnancement circulaire ou tourniquet (round robin)

Il s'agit d'un algorithme ancien, simple et fiable. Le processeur gère une liste circulaire de processus. Chaque processus dispose d'un quantum de temps pendant lequel il est autorisé à s'exécuter. Si le processus actif se bloque ou s'achève avant la fin de son quantum, le processeur est immédiatement alloué à un autre processus. Si le quantum s'achève avant la fin du processus, le processeur est alloué au processus suivant dans la liste et le processus précédent se trouve ainsi en queue de liste.

La commutation de processus (overhead) dure un temps non nul pour la mise à jour des tables, la sauvegarde des registres. Un quantum trop petit provoque trop de commutations de processus et abaisse l'efficacité du processeur. Un quantum trop grand augmente le temps de réponse en mode interactif. On utilise souvent un quantum de l'ordre de 100 ms.

3.3 Ordonnancement PCTER

Il s'agit d'une généralisation avec réquisition de l'algorithme PCTE : à la fin de chaque quantum, on élit la tâche dont le temps d'exécution restant est minimal (PCTER = plus court temps d'exécution restant). Cet algorithme fournit la valeur optimale du temps moyen de traitement pour les algorithmes avec réquisition.

3.4 Ordonnancement avec priorité

Le modèle du tourniquet suppose tous les processus d'égale importance. C'est irréaliste. D'où l'attribution de priorité à chaque processus. **L'ordonnanceur lance le processus prêt de priorité la plus élevée**. Pour empêcher le processus de priorité la plus élevée d'accaparer le processeur, l'ordonnanceur abaisse à chaque interruption d'horloge la priorité du processus actif. Si sa priorité devient inférieure à celle du deuxième processus de plus haute priorité, la commutation a lieu.

Une **allocation dynamique de priorité** conduit souvent à donner une priorité élevée aux processus effectuant beaucoup d'E/S. On peut aussi allouer une priorité égale à l'inverse de la fraction du quantum précédemment utilisée. Par exemple un processus ayant utilisé 2 ms d'un quantum de 100 ms, aura une priorité de 50.

A chaque niveau de priorité, on associe une liste (classe) de processus exécutables, gérée par l'algorithme du tourniquet. Par exemple, supposons 4 niveaux de priorité. On applique à la liste de priorité 4 l'algorithme du tourniquet. Puis, quand cette liste est vide, on applique l'algorithme à la liste de priorité 3, etc... S'il n'y avait pas une évolution dynamique des priorités, les processus faiblement prioritaires ne seraient jamais élus : risque de famine.

Exemple : VAX/VMS de DEC et OS/2 d'IBM

3.5 Ordonnancement avec files multiples

On associe une file d'attente et un algorithme d'ordonnancement à chaque niveau de priorité. Si les priorités évoluent dynamiquement, le SE doit organiser la remontée des tâches.

Exemple : UNIX dont le mode d'ordonnancement sera présenté dans un chapitre ultérieur

3.6 Ordonnancement par une politique

S'il y a n utilisateurs connectés, on peut garantir à chacun qu'il disposera de 1/n de la puissance du processeur. Le SE mémorise combien chaque utilisateur a consommé de temps processeur et il calcule le rapport : *temps processeur consommé/temps processeur auquel on a droit* . On élit le processus qui offre le rapport le plus faible jusqu'à ce que son rapport cesse d'être le plus faible.

3.7 Ordonnancement à deux niveaux

La taille de la mémoire centrale de l'ordinateur peut être insuffisante pour contenir tous les processus prêts à être exécutés. Certains sont contraints de résider sur disque.

Un ordonnanceur de bas niveau (*CPU scheduler*) applique l'un des algorithmes précédents aux processus résidant en mémoire centrale. Un ordonnanceur de haut niveau (*medium term scheduler*) retire de la mémoire les processus qui y sont resté assez longtemps et transfère en mémoire des processus résidant sur disque. Il peut exister un ordonnanceur à long terme (*job scheduler*) qui détermine si un processus utilisateur qui le demande peut effectivement entrer dans le système (si les temps de réponse se dégradent, on peut différer cette entrée).

L'ordonnanceur de haut niveau prend en compte les points suivants :
- depuis combien de temps le processus séjourne-t-il en mémoire ou sur disque ?
- combien de temps processeur le processus a-t-il eu récemment ?
- quelle est la priorité du processus ?
- quelle est la taille du processus ? (s'il est petit, on le logera sans problème)

3.8 Ordonnancement de chaînes de tâches

Supposons que l'on ait r chaînes de tâches à exécuter en parallèle : $C_1, C_2, ..., C_r$
avec : $C_i = T_{i1}\ T_{i2}\ T_{i3} .... T_{i\ ki}$, c'est à dire que la chaîne C_1 comprend k1 tâches, la chaîne C_2 comprend k2 tâches, etc... On note : k1 + k2 + ... + kr = n (nombre total de tâches).
On note : τ_{ij} la durée de la tâche T_{ij}

Il existe un algorithme produisant une assignation de temps moyen de traitement minimal :

pour chaque chaîne C_i , calculer la durée moyenne d'exécution (pas de traitement) ε_{ip} de chaque sous-chaîne C'_{ip} , pour p = 1 à ki
tant qu'il existe des tâches non assignées
déterminer j tel que : $\varepsilon_{j\,pj} = \min \{ \varepsilon_{m\,pm} \mid m = 1 , 2 , ... , n \}$
ajouter à l'assignation les tâches de la sous-chaîne $C'_{j\,pj}$ et les supprimer de la chaîne C_j
recalculer $\varepsilon_{j\,pj}$ pour la nouvelle chaîne C_j
fin tant que

Exemple : $C_1 : \tau_{11} = 9, \tau_{12} = 6 , \tau_{13} = 3 , \tau_{14} = 5 , \tau_{15} = 11$
$C_2 : \tau_{21} = 4 , \tau_{22} = 1 , \tau_{23} = 13$
$C_3 : \tau_{31} = 8 , \tau_{32} = 4 , \tau_{33} = 1 , \tau_{34} = 11$

On détermine le minimum : $\tau_{22} = 2{,}5$, donc : l'assignation T_{21} , T_{22}
puis $\tau_{33} = 4{,}33$; on complète l'assignation avec T_{31} , T_{32} , T_{33} ; etc ...
D'où l'assignation finale :

$$T_{21} , T_{22} , T_{31} , T_{32} , T_{33} , T_{11} , T_{12} , T_{13}, T_{14}, T_{15}, T_{34}, T_{23}$$

12

BIBLIOGRAPHIE

J. BEAUQUIER, B. BERARD, Systèmes d'exploitation, Ediscience, 1993

A. TANENBAUM, Les systèmes d'exploitation, Prentice Hall, 1999

Exercices sur le chapitre 3

1. Donner et comparer les assignations produites par les algorithmes FIFO, PCTE, tourniquet avec un quantum de 1, PCTER dans l'exemple suivant :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	7	4	6	1	2	4	1
date d'arrivée	0	0	1	1	1	2	2

2. Sur un ordinateur, l'ordonnanceur gère l'ordonnancement des processus par un tourniquet avec un quantum de 100 ms. sachant que le temps nécessaire à une commutation de processus est de 10 ms, calculer le temps d'exécution moyen pour :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	700	400	600	100	200	400	100
date d'arrivée	0	0	100	100	150	200	200

   Si l'on définit le rendement du processeur comme le rapport *temps pendant lequel l'UC exécute les processus/temps total de traitement*, calculer le rendement en ce cas.

3. Un SE utilise 3 niveaux de priorité (numérotés par ordre croissant). Le processus se voit affecter un niveau fixe. Une file de processus est attachée à chaque niveau. Chaque file est gérée par un tourniquet avec un quantum de 0,5. Un tourniquet de niveau n n'est activé que si toutes les files de niveau supérieur sont vides.

   Que peut-il se passer ?

   Donner l'assignation pour :

ordre d'arrivée des tâches	T1	T2	T3	T4	T5	T6	T7
durée	7	4	6	1	2	4	1
date d'arrivée	0	0	1	1	1	2	2
priorité	2	3	1	2	3	1	2

   Maintenant, on suppose que la priorité n'est pas fixe. Toutes les 2 unités de temps, tout processus n'ayant pas disposé de l'UC monte d'un niveau, alors que ceux en ayant disposé 2 fois en descendent. Donner la nouvelle assignation.

Chapitre 4 LES PROCESSUS AVEC UNIX

1. GENERALITES SUR LES PROCESSUS SOUS UNIX

1.1 Contexte d'un processus

On distingue des **processus utilisateurs** et des **processus système**. Ces derniers :

- ne sont sous le contrôle d'aucun terminal
- ont comme propriétaire le super utilisateur (processus **démons**). Ils restent résidents en MC en attente d'une requête
- ils assurent des services généraux accessibles à tous les utilisateurs
- ils peuvent être créés au lancement du système ou à des dates fixées par l'administrateur.

Exemples de processus système :

cron lance à des dates spécifiées des commandes particulières
lpsched assure l'ordonnancement des requêtes d'impression

Le contexte d'un processus, défini à un instant donné de l'exécution, comprend l'ensemble des 3 environnements suivants nécessaires à son exécution :

- **environnement utilisateur :** (il se trouve dans le fichier exécutable)
  zone programme (code à exécuter) ou TEXT
  zone données (variables) ou DATA
  zone pile utilisateur (pour les variables locales et les appels de fonctions) ou BSS, dont la taille peut être modifiée dynamiquement.

La zone programme peut être partagée entre plusieurs processus. En outre, un processus peut partager des segments de mémoire (données) avec d'autres processus,

- **environnement machine :** ensemble des registres utilisés pour l'exécution (compteur ordinal, pointeur de pile, registres de travail, registres de données, registres pour la mémoire virtuelle, etc...)

- **environnement système :** situé dans le noyau, il comprend au moins les 3 structures suivantes :

  * *table des processus :* entité globale; chaque processus correspond à une ligne de la table (pid, état, priorité, utilisateur réel [celui qui l'a créé], utilisateur effectif [celui qui précise les droits d'accès], etc...)

  * *structure U* : il en existe une par processus. Elle contient des informations de contrôle : répertoire courant; pointeurs vers les fichiers ouverts; variables d'environnement)

  * *table des textes :* utilisée par la gestion du code partageable. Chaque ligne contient les informations suivantes : taille du texte, pointeur vers la structure U, vers l'i-node du fichier exécutable, nombre de processus qui exécutent ce texte, etc...)

L'environnement système initial est créé par le shell lors de la création du processus.

Un processus peut s'exécuter dans deux modes différents :

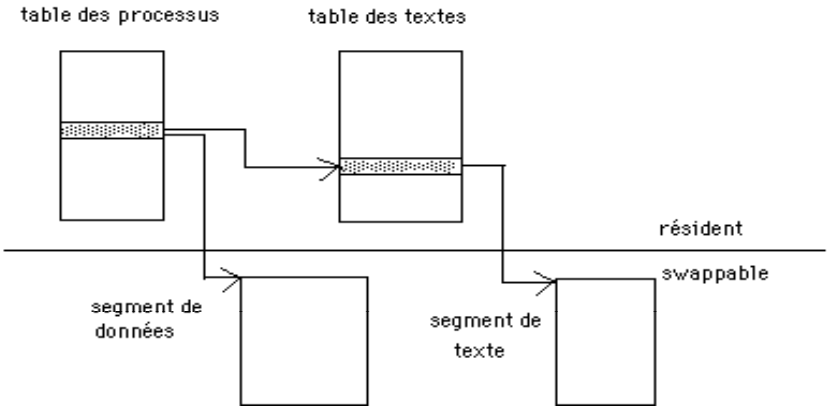
- **mode utilisateur (user mode) :** le processus n'accède qu'à son espace d'adressage et n'exécute que des instructions ordinaires du programme chargé

- **mode noyau ou système (kernel mode) :** le processus exécute des instructions n'appartenant pas au programme, mais au noyau. Il accède à des données externes à son espace d'adressage (tables systèmes). On peut passer en mode noyau soit par une interruption appropriée, soit par la réalisation d'un appel système par le processus lui-même (exemple : requête de lecture au clavier dans le programme)

1.2 Structure d'un processus sous UNIX

Le noyau gère une *table des processus* , listable par la commande **ps** avec une entrée par processus, et contenant des informations sur celui-ci. Cette entrée est allouée à la création du processus, désallouée à son achèvement.

Un processus est identifié par un *pid* (*process identifier*), entier de 0 à 32767 retourné par le noyau. Il est caractérisé aussi par sa priorité, son propriétaire et son groupe propriétaire, son terminal de rattachement.



1.3 Initialisation

A l'initialisation du système, s'exécute un bootstrap primaire résidant sur la piste 0 du disque qui charge un boostrap secondaire (incluant les fonctions d'un **échangeur**), et exécutant le noyau d'UNIX, de pid 0. C'est l'échangeur (**swapper)** qui assure la gestion du va-et-vient (overlay) en mémoire en remplissant les rôles suivants :

- *chargement* : trouver parmi les processus activables celui qui attend depuis le plus longtemps et qui peut s'implanter dans l'espace libre de la mémoire.

- *élimination :* on élimine d'abord les processus bloqués dans l'ordre de leur durée de résidence en mémoire centrale, et éventuellement d'autres processus.

Le noyau découpe la mémoire centrale, teste les volumes, vérifie leur cohérence, charge les tables systèmes nécessaires. Le noyau utilise l'échangeur pour créer 4 processus dont l'exécution de /etc/ **init, de pid 1**. **init** est l'ancêtre de tous les processus. Il consulte des tables (dont /sbin/inittab qui contient les conditions initiales du système). **init** crée par filiation autant de processus **getty** qu'il y a de lignes dans /etc/inittab.

Chaque utilisateur est identifié par un numéro individuel **UID** (*user identity*). La correspondance entre le nom et **UID** est assurée par le fichier **/etc/passwd**. Un groupe est un ensemble d'utilisateurs ayant des points communs. Un utilisateur appartient à un ou plusieurs groupes. Un groupe est identifié par un **GID** (*group identity*) et possède un nom. Leur correspondance est gérée par **/etc/grou**p.

→ La commande **uid** affiche l'**UID**, le login name, le **GID** et le nom du groupe de l'utilisateur.

Exemples:

1. connexion des utilisateurs (getty) : les processus **getty** , lancés par init, réalisent les 2 opérations suivantes :

- préparation du terminal en fonction des éléments de /etc/gettydefs, initialisation du groupe avec **setpgrp** et ouverture du fichier spécial correspondant au terminal défini
- bouclage sur les 3 opérations suivantes :
  - affichage d'un prompt contenu dans /etc/issue et d'un message d'invitation à la connexion,
  - lecture du nom de connexion
  - exécution du processus login qui se substitue à getty (même pid, chef de groupe)

2. administration de l'utilisateur (login) : login, qui reçoit le nom d'utilisateur lu par getty, lit le mot de passe, compare le login et le mot de passe crypté aux données de **/etc/passwd** . Si une concordance est bonne, login :

- met à jour les fichiers de comptabilité,
- modifie les droits de propriété du terminal pour concordance avec ceux de l'usager,
- détermine le répertoire d'accueil d'après /etc/passwd, initialise les variables HOME, PATH, LOGNAME,...
- exécute (par recouvrement) le programme nommé à côté de ce mot de passe dans /etc/passwd. Lorsque la session est achevée (fin de sh par exemple), il y a retour à **init** et suppression de tous les processus attachés au défunt shell
- exécute /etc/profile et aussi **/etc/.profile** (commun à tous les utilisateurs), **.profile** (s'il existe).

3. Lorsqu'on demande une **redirection de sortie** (> ou >>), le shell ferme le fichier de descripteur 1 (stdout en général), ouvre le fichier de redirection qui prend le descripteur 1 venant d'être libéré, et exécute le fork évoqué ci-dessus. Il en est de même pour une redirection d'entrée (cf. chapitre 11).

4. La fonction C **system** crée un processus shell qui interrompt le programme en cours (processus père) jusqu'à la fin du processus fils (argument de system).

5. La fonction **ioctl** prototypée dans termio.h permet de reconfigurer les pilotes de terminaux. Très bon exemple dans Braquelaire p. 358-359, 361-368.

1.4 Ordonnancement d'un processus

L'**ordonnanceur** ou **scheduler** (gestionnaire des ressources et d'enchaînement des processus) gère plusieurs processus concurremment et autorise le multitâche par partage de temps. Il attribue des **quanta** de temps. A l'expiration de son **quantum** de temps, le processus actif, s'il n'est pas terminé, sera obligé de relâcher le processeur. Un autre processus est alors élu et attribué au processeur. Le processus suspendu reprend son exécution quand son tour arrive à nouveau pour disposer du processeur.

Sous UNIX, l'horloge de l'ordonnanceur délivre 100 **tops**, ou cycles mineurs, par seconde et un **quantum**, ou cycle majeur, correspond à plusieurs tops (par exemple 100, soit 1 seconde).

Un processus peut prendre l'un des 9 états décrits dans **/usr/include/sys/proc.h** (codés dans le champ **S** dans la table des processus), parmi lesquels :

- actif : en mode utilisateur (pas de droit d'accès aux ressources du système) ou en mode système (ou noyau)

- activable ou prêt à exécuter : en mémoire ou en zone de swap. Il est éligible par l'ordonnanceur. Ces deux états sont codés **R**

- bloqué : en attente d'un événement (horloge, montage d'un disque, résultat d'un autre processus, ...), en mémoire ou en zone de swap. Il ne consomme pas de temps CPU. Il repasse à l'état activable dès que les conditions le permettent. Désigné par **S** (sleep) pendant les 20 premières secondes), puis par **I**

- non prêt : en création ou zombie (désigné par **Z**, le processus est achevé, mais son père n'en a pas encore connaissance). Il ne consomme pas de temps CPU.

L'ordonnanceur d'UNIX SystemV met en œuvre un algorithme de calcul de la priorité des processus candidats au processeur à partir de quatre paramètres figurant dans la table des processus :

C ou p_pcpu: rend compte de l'utilisation du processus. Si le processus utilise le processeur, C est incrémenté à chaque top d'horloge et divisé par 2 à chaque quantum.

NI ou p_nice: (nice) paramètre fixé par l'utilisateur (fonction UNIX **nice**) entre 0 (priorité maximale) et 39 (priorité minimale) ou à défaut par le système à la valeur 20.

Exemple : nice - 12 tp.e
abaisse de 12 la priorité du processus associé à l'exécution de tp.e
nice admet des arguments entre 1 et 19 (10 par défaut)

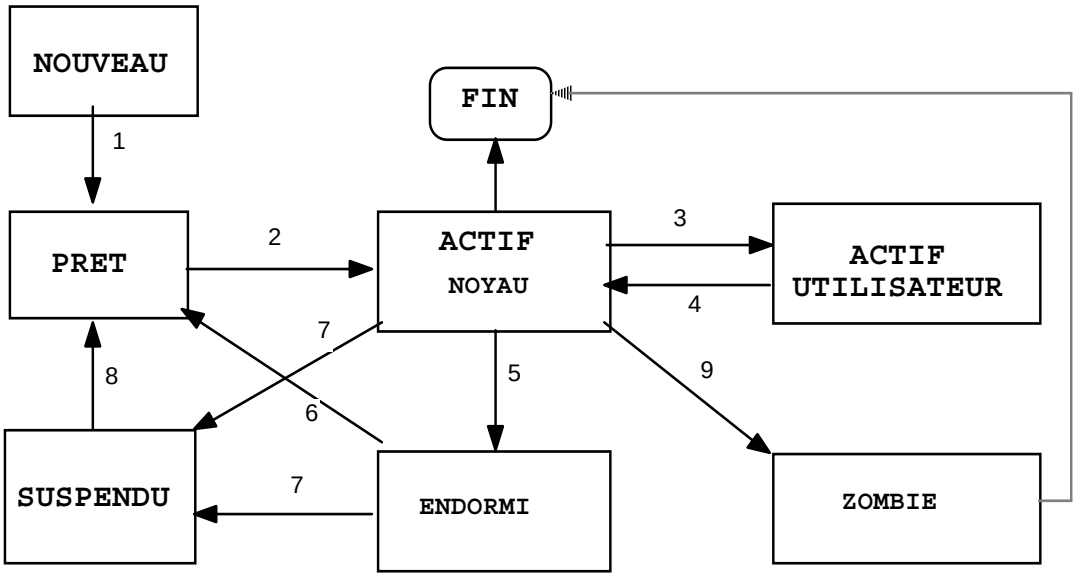
PRI ou p_user: (priority) est fixé à 60.

NZERO: est fixé à 20.

La priorité globale d'un processus utilisateur à un quantum donné vaut :

$$(C/2) + PRI - NZERO + NI$$

Elle ne peut donc être inférieure à 40. Les valeurs 0 à 39 sont réservées aux processus système (PRI = 0). Plus sa valeur est élevée, moins un processus a de chances d'accéder au processeur.



1 : le processus créé par fork a acquis les ressources nécessaires à son exécution
2 : le processus vient d'être élu par l'ordonnanceur
3 : le processus revient d'un appel système ou d'une interruption
4 : le processus a réalisé un appel système ou une interruption est survenue
5 : le processus se met en attente d'un événement (libération de ressource, terminaison de processus par wait). Il ne consomme pas de temps UC
6 : l'événement attendu par le processus s'est produit
7 : conséquence d'un signal particulier
8 : réveil du processus par le signal de continuation SIGCONT
9 : le processus s'achève par exit, mais son père n'a pas pris connaissance de sa terminaison. Il ne consomme pas de temps UC et ne mobilise que la ressource table des processus

La commande **ps** propose 4 options : **- a** (tous les processus attachés au terminal), **- e** (tous les processus), **- f** (full : plus de renseignements), **- l** (long : presque tous les renseignements). Les champs de la tables des processus qui sont édités sont les suivants en fonction des options - f ou - l :

F (- l) localisation d'un processus (0 = hors MC, 1 = en MC, 2 processus système, 4 = verrouillé car en attente de fin d'E/S, 10 = en vidage)

S (-l) state (O = non existant, S = sleeping, W = waiting, R = running, Z = zombie, X = en évolution, P = pause, I = attente d'entrée au terminal ou input, L = attente d'un fichier verrouillé ou locked)

UID	nom du propriétaire
PID et PPID	
PRI et NI (- l)	voir ci-dessus
SZ	taille en nombre de blocs
WCHAN (- l)	dans l'état W, l'événement attendu
STIME (- l)	heure de démarrage
TTY	ligne associée au terminal

TIME	temps d'exécution cumulé (mm:ss)
CMD	commande (et ses arguments si - f) exécutée par le processus

1.5 Gestion du va-et-vient (overlay)

L'échangeur (processus système de pid 0, créé au moment du démarrage du système) assure la gestion du va-et-vient de la mémoire. Il assure trois fonctions :

- *allocation d'un espace de swap :* la gestion de l'espace disque d'échange est différente de celle du reste du disque. On gère un ensemble contigu de blocs de taille fixe, à raison d'une map par périphérique d'échange, en utilisant les fonctions **swapalloc** (int taille) et **swapfree** (int adresse, int taille) de demande et de libération d'unités d'échange. Il peut y avoir plusieurs périphériques d'échange, créés et détruits dynamiquement.

- *transfert en mémoire :* l'échangeur examine tous les processus qui sont "prêt et transféré sur disque" et sélectionne celui qui a été transféré sur disque depuis le plus longtemps (allocation de MC, lecture du processus sur disque et libération de l'espace de swap). Il recommence jusqu'à ce qu'il n'y ait plus de processus "prêt et transféré sur disque" (il s'endort) ou bien jusqu'à ce qu'il n'y ait plus de MC libre (il exécute le point suivant).

- *transfert hors de la mémoire :* l'échangeur cherche à transférer d'abord les processus "bloqué" avant les "prêt", mais jamais les "zombie", ni les processus "verrouillé par le SE". Un processus "prêt" ne peut être transféré que s'il a séjourné au moins 2 s. en MC (même sans avoir été élu). S'il n' y a aucun processus transférable, l'échangeur s'endort et est réveillé toutes les secondes jusqu'à la fin de la nécessité de transfert.

Il peut y avoir "étreinte fatale" lorsque tous les processus en MC sont "endormi", tous les processus "prêt" sont transférés et il n' y a plus de place en MC ni en zone d'échange sur disque.

En plus du va-et-vient normal, il existe deux possibilités de transfert :

- à l'exécution d'un **fork**, lorsque la création du fils ne peut être faite directement en MC : on crée le fils sur disque par recopie du père

- lors de l'accroissement de la taille d'un processus : le SE transfère le processus sur disque en réservant une zone de swap plus grande; au retour en MC, le processus sera "plus grand".

2. PROGRAMMATION DE PROCESSUS AVEC LE SHELL

Exécution en parallèle (background)

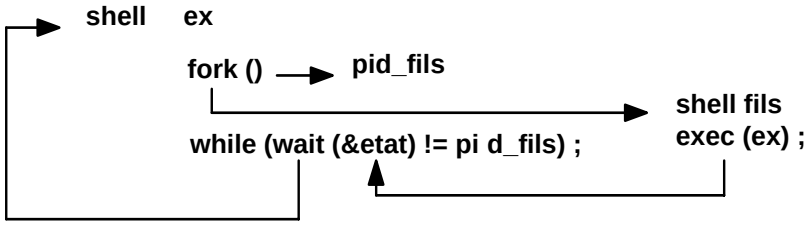
Soit **ex** un fichier de commandes exécutable.

1er cas : on tape **$ ex** Voici ce qui se passe :

1. le shell lit la commande ex
2. il se duplique au moyen de la fonction fork ; il existe alors un shell père et un shell fils
3. grâce à la fonction exec, le shell fils ***recouvre*** son segment de texte par celui de ex qui s'exécute.

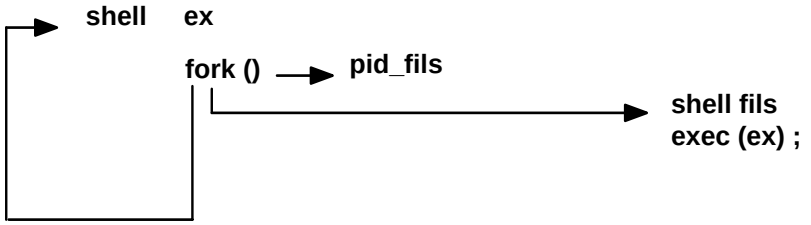
4. le shell père récupère le pid du fils retourné par fork ().
5. à la fin de l'exécution de ex, le shell père reprend son exécution.

La fonction wait lui permet de connaître son achèvement : wait (&etat) retourne le pid du processus fils à son achèvement (etat : entier).



2ème cas : on tape **$ ex&** pour que ex s'exécute en arrière-plan (en parallèle au shell, mais n'accède pas à l'entrée standard réservée au shell). Voici ce qui se passe :

1. à 4. comme ci-dessus
5. le shell père reçoit le pid du fils, émet un prompt, puis reprend son activité sans attendre la fin du fils.



Attention, *un processus lancé en arrière-plan ne réagit pas aux interruptions émises au clavier.*

Les variables d'un processus shell ne sont pas transmises à son fils, à l'exception des variables d'environnement. En particulier, login transmet les variables d'environnement à ses fils. Si l'on veut que le fils d'un processus shell reconnaisse la variable x de son père, on exécutera dans le père la commande

export x

Si l'on veut retirer cette variable de l'environnement accessible au fils, on exécutera :

unset x

Il n'existe aucun moyen d'exporter une variable d'un sous-shell vers son père.

Exemple :$ x=bonjour

```
$ export x
$ ksh                 # création d'un nouveau sous-shell
$ echo $x
bonjour
$ x='au revoir'
$ <CTRL-D>            # retour au shell de départ
$ echo $x
bonjour
```

La commande **nohup** (*no hangup*, pas d'arrêt imprévu) empêche l'arrêt d'un processus, même si l'utilisateur se déconnecte.
stdout et stderr sont redirigés par défaut sur le fichier **nohup.out**

Exemple : $ nohup sort oldfic > newfic &

550
$

3. PROGRAMMATION DE PROCESSUS AVEC C

Les fonctions UNIX citées ci-dessus sont toutes accessibles depuis C.

3.1 fork ()

fork engendre un fils **partageant** le segment de textes du père, s'il est partageable, et disposant d'une **copie** de son segment de données. Il retourne 0 au fils et le pid du fils au père si la création a réussi (-1 sinon). Chaque processus a accès à la description du *terminal de contrôle* (terminal où l'on s'est connecté) contenue dans **/dev/tty**. Un processus fils hérite du terminal de contrôle du père. Le fils hérite d'une **copie** des descripteurs de fichiers ouverts. Les pointeurs sur fichiers sont partagés. Père et fils ont leurs propres tampons d'accès aux fichiers, dupliqués initialement par fork (). A l'exécution d'un fork (), le fils ferme les E/S standards si elles sont redirigées, puis il ouvre les fichiers de redirection. Il hérite donc des E/S redirigées.

Un processus fils n'hérite pas du temps d'exécution du père (initialisé à 0 pour le fils). Il n'hérite pas de la priorité du père, sauf de son paramètre nice.Il n'hérite pas des verrous sur les fichiers détenus par le père. Il n'hérite pas des signaux en suspens (signaux envoyés au père, mais pas encore pris en compte).

Tout processus, sauf le processus de pid 0, est créé par fork.

3.2 exit ()

void exit (int etat)

La fonction provoque la terminaison du processus avec le code de retour **etat** (0 = bonne fin). Si le père est un shell, il récupère **etat** dans la variable **$?**

A l'exécution de **exit**, tous les fils du processus sont rattachés au processus de pid 1.**exit** réalise la libération des ressources allouées au processus et notamment ferme tous les fichiers ouverts. Si le père est en attente sur **wait**, il est réveillé et reçoit le code de retour du fils.

3.3 wait ()

int wait (int *p_etat)

- suspend le processus jusqu'à ce qu'un de ses fils prenne fin. Si le fils se termine normalement, et que le père ne fait pas **wait** pour l'attendre, on dit que le fils est devenu un **processus zombie**. Il n'est plus pris en compte par l'ordonnanceur, son compteur d'alarme est annulé, mais son entrée dans la table des processus n'est pas enlevée. Elle est signalée par la mention **<defunct>**

- retourne le pid du fils à son achèvement, ou -1 s'il n'y a plus de fils actif. Si un ou plusieurs fils sont zombis, c'est le pid de l'un de ces fils qui est retourné. L'algorithme de wait privilégie donc la suppression des zombies.

Si le père s'achève sans attendre la fin d'un fils par **wait**, le fils sera pris en charge par **init**, le processus système de pid 1.

Si **p_etat** est non nul, ***p_etat** fournit des informations sur le type de terminaison du processus fils :

- si le processus fils s'est achevé normalement par un appel à **exit,** l'avant-dernier octet de ***p_etat** contient la valeur de l'argument de **exit** (l'octet de faible poids est à zéro)

- sinon, le processus fils s'est terminé par suite de la réception d'un **signal** (voir paragraphe 6). Alors, les bits 0 à 6 donnent le numéro du signal (0 à 127) et les octets de fort poids sont à zéro. Si le bit 7 est à 1 (valeur de l'octet de faible poids : 128 à 255), il y a eu un fichier **core** créé. On peut récupérer le numéro du signal en enlevant 128 au contenu de l'octet de faible poids.

3.4 La famille des fonctions de recouvrement (exec)

Il s'agit d'une famille de 6 fonctions permettant le lancement de l'exécution d'un nouveau programme par un processus. Le segment de texte est remplacé par celui du nouveau programme. Il y a recouvrement en mémoire, **mais pas de création d'un nouveau processus**. Le retour est -1 en cas d'erreur.

Les 6 fonctions sont **execl, execv, execle, execve, execlp, execvp**. Par exemple :

int execv (char *nom, char * argv [])

nom : pointeur sur le nom du programme à exécuter,
argv : liste de pointeurs (terminée par NULL) sur les arguments transmis au nouveau programme avec argv[0] : "nom" du programme pointé par nom

Un excellent exemple est donné dans Rifflet p. 251. Attention aux fichiers : seuls les descripteurs des fichiers ouverts (situés en zône système) sont conservés par un appel à exec. Les modes d'ouverture, les structures FILE associées sont recouverts, ainsi que les tampons d'accès. Il est donc recommandé d'appeler **fflush** avant exec pour vider les tampons dans les fichiers, y compris pour **stdout**. Un très bon exemple en est donné dans Rifflet p. 253.

Autre exemple intéressant : Braquelaire p. 448-449

3.5 Exemples

Exemple 1 :

```
main ()
{
   int pid_fils;
   bloc1
   if ((pid_fils = fork()) == 0)
       bloc2
   else
       bloc3
}
```

Le processus père exécutera le bloc 1. Puis il créera un fils identique par **fork** et exécutera le bloc 3 puisque **fork** lui retournera une valeur non nulle.

Le processus fils n'exécutera pas le bloc 1 car à sa création, son compteur ordinal pointera sur la ligne contenant **fork**. Comme **fork** lui retourne 0, il exécutera le seul bloc2. L'affichage des résultats des blocs 2 et 3 peut être entrelacé, et pas nécessairement de façon identique d'une exécution à une autre.

Exemple 2 :

```
main ()
 {
   int m,n;
   printf ("processus père.Fils non encore créé\n");
   if (fork () == 0)
   {
        printf ("pid du fils %d\n", getpid());
             exit (1);
    }
   else {
        printf ("pid du père %d\n", getpid());
             m = wait (&n);
             printf ("fin du processus de pid %d avec valeur de retour de wait
%d\n",m,n);
          }
}
```

3.6 Les événements ou signaux

Le fichier **/usr/include/signal.h** contient notamment la définition de NSIG = 32 constantes caractérisant des événements, dont plusieurs génèrent un fichier **core** (copie de l'image mémoire sur disque; ils sont suivis d'un * dans le tableau ci-dessous) :

- interruptions matérielles (frappe d'un caractère,...)
- interruptions logicielles externes (terminaison d'un autre processus,...) ou internes(erreur arithmétique, violation mémoire,...).

NOM	NUMERO	ROLE
SIGHUP	1	émis à tous les processus associés à un terminal ou un modem quand il déconnecte
SIGINT	2	émis à tous les processus associés à un terminal lorsque <INTR> (par défaut <DEL> ou CTRL-C) est frappé au clavier
SIGQUIT	3*	émis à tous les processus associés à un terminal lorsque <QUIT> (par défaut CTRL- \ ou CTRL-Z) est frappé au clavier
SIGILL	4*	instruction illégale
SIGTRAP	5*	émis après chaque instruction en cas d'exécution en mode trace, par un débogueur
SIGABR	6*	abort
SIGEMT	7	piège d'instruction débogueur
SIGFPE	8*	erreur dans une instruction en virgule flottante
SIGKILL	9	arrêt obligatoire du processus (ne peut être ni ignoré, ni capturé)
SIGBUS	10*	erreur d'adressage sur le bus
SIGSEGV	11*	violation des limites de l'espace mémoire
SIGSYS	12*	mauvais argument dans un appel système
SIGPIPE	13	écriture dans un tube sans lecteur
SIGALRM	14	signal associé à une horloge (cf. ci-dessous fonction système alarm).
SIGTERM	15	signal de terminaison normale d'un processus
SIGUSR1	16	à la disposition des utilisateurs pour la communication entre processus
SIGUSR2	17	idem au précédent
SIGCHLD	18	mort d'un fils provoquée par exit
SIGPWR	19	panne d'alimentation électrique
SIGWINCH	20	changement de taille de fenêtre
SIGURG	21	message socket urgent
SIGPOLL	22	événement stream pollable (cf. sockets)
SIGSTOP	23	signal stop envoyé
SIGSTP	24	stop par l'utilisateur

SIGCONT	25	continuation
SIGTTIN	26	stop sur l'entrée du terminal
SIGTOU	27	stop sur la sortie du terminal
SIGVTALRM	28	expiration d'un temporisateur virtuel
SIGPROF	29	temporisateur de profile expiré
SIGXCPU	30*	temps maximal CPU expiré
SIGXFSZ	31*	taille maximum du fichier dépassée

Les signaux 18 à 21, ainsi que le signal 25, sont ignorés par défaut. Les autres provoquent l'arrêt du processus, avec éventuellement la création d'un fichier core.
Les signaux 4, 6, 7, 10 et 11 sont bien décrits dans BRAQUELAIRE, p. 434.

La fonction **signal**, prototypée par :

void (* signal (int sig, void (*fonc)(int))) (int)

permet d"intercepter le signal **sig** et d'exécuter la fonction (ou **handler**) pointée par **fonc**.
Le processus reprend son exécution à la fin de l'exécution de **fonc**.

fonc peut prendre des valeurs prédéfinies (macros) :

SIG_IGN : signal ignoré, sauf SIGKILL
SIG_DFL : fin du processus, avec ou sans fichier core selon **sig**

Exemple :

```
#include <signal.h>
        main ()
        {       signal (SIGINT,SIG_IGN);
        signal (SIGQUIT,SIG_IGN);
 /* à partir d'ici, les interruptions SIGINT (par défaut CTRL-C) et SIGQUIT (par défaut
CTRL- \) seront désarmées   */
                        .....
                        signal (SIGINT, SIG_DFL);
                signal (SIGQUIT,SIG_DFL);
  /* à partir d'ici, SIGINT ou SIGQUIT mettront fin au processus */
                        .....
```

Lorsque **fonc** est un pointeur sur une fonction utilisateur, 3 actions se produisent à l'arrivée de **sig** :

- le programme courant du processus est interrompu
- le processus est dérouté vers la fonction pointée par **fonc**, *avec en argument le n° du signal*
- après exécution de cette fonction, le programme courant du processus reprend son exécution, sauf si la fonction a réalisé un exit

Lorsqu'un processus est lancé en arrière-plan, les commandes signal (SIGINT,SIG_IGN) et signal (SIGQUIT,SIG_IGN) sont lancées et empêchent son interruption par les moyens classiques (CTRL-C ou CTRL- \). Il faut recourir à kill -9 pour l'interrompre.

Un processus fils hérite du comportement de son père vis à vis des signaux.

La fonction prototypée par : **unsigned alarm (unsigned s)**, et décrite dans <unistd.h>, envoie SIGALRM au processus après s secondes (s <= MAX_ALARM définie dans <sys/param.h>). Une telle demande annule une demande antérieure du même type. Si s = 0, on annule simplement la demande antérieure. Après un *fork*, l'horloge du fils est mise à 0, quelque soit celle du père. Ce n'est pas le cas après un *exec*.

alarm retourne le temps restant avant l'envoi du signal correspondant à cette demande antérieure.

Un processus peut envoyer un signal **s** à un processus de pid **p**, à condition qu'ils aient tous deux le même propriétaire, grâce à la fonction **kill** prototypée par :

int kill (int p, int s)

kill retourne 0 si l'opération s'est bien déroulée, -1 sinon.
La valeur **s** = 0 permet de tester si le processus de pid **p** existe.

Il existe évidemment la commande système **kill** bénéficiant des mêmes propriétés.
Exemple : kill -9 12356 envoie le signal 9 au processus de pid 12356

La commande shell **trap** permet d'exécuter un processus de déroutement à la réception d'un signal donné.

Exemples : trap 'rm toto.c;exit' 3 4
a pour effet de demander l'exécution de la séquence rm toto;exit à la réception des signaux 3 ou 4 (exit est bien sûr nécessaire pour assurer la fin du processus).

trap " 3 4 permettrait d'ignorer les signaux 3 et 4

La fonction **int pause (void)** décrite dans <unistd.h> met le processus en attente de l'arrivée du premier événement non masqué; pause retourne -1 si l'événement provoque le déroutement sur une autre fonction.

De bons exemples faisant intervenir signal, alarm, wait et pause sont donnés dans BRAQUELAIRE, p. 435-440, 461-464 et surtout 440-443.

A noter aussi les fonctions **int sigsetjmp et int siglongjmp**, décrites dans <setjmp.h>, qui respectivement sauve le contexte d'un processus et restaure ce contexte.

BIBLIOGRAPHIE

J.-P.BRAQUELAIRE, Méthodologie de la programmation en Langage C, Masson,1993

A.B. FONTAINE et Ph. HAMMES, UNIX Système V, Masson, 1993

J.L. NEBUT, UNIX pour l'utilisateur, Technip, 1990

J.M. RIFFLET, La programmation sous UNIX, Mc Graw-Hill, 1992

Chapitre 5 PROCESSUS ET RESSOURCES

1. GENERALITES

Un processus à exécuter a besoin de *ressources* : procédures et données, mémoire, processeur (UC , et d'autres éventuellement), périphériques, fichiers, ...

Une ressource est soit *locale*, soit *commune* à plusieurs processus

locale : utilisée par un seul processus. Ex.: fichier temporaire, variable de programme.

commune ou partageable . Ex. : disque, imprimante, fichier en lecture.

Une ressource peut posséder un ou plusieurs *points d'accès* à un moment donné :

* un seul : on parle de *ressource critique*
Ex: un lecteur de disquettes partagé entre plusieurs utilisateurs. Une zone mémoire partagée en écriture entre plusieurs utilisateurs.

* plusieurs points d'accès: par exemple, un fichier en lecture utilisable par plusieurs processus.

Le SE doit contrôler l'utilisation des ressources dans une **table** indiquant si la ressource est disponible ou non, et, si elle est allouée, à quel processus. A chaque ressource est associée une **file d'attente**, pointée par la table précédente, contenant les BCP des processus qui l'attendent. Chaque fois qu'un nouveau processus fait une demande de la ressource et que cette dernière n'est pas disponible, son BCP est ajouté en queue de la file d'attente. Lorsqu'une demande survient de la part d'un processus plus prioritaire que celui qui utilise la ressource, on empile l'état de la ressource, on la retire au processus en cours pour l'attribuer par **réquisition** au processus prioritaire.

Dans le cadre des ressources limitées gérées par le système, il n'est pas toujours possible d'attribuer à chaque processus, dès sa création, toutes les ressources nécessaires. Il peut y avoir **blocage** .

2. LE PROBLEME DE L'EXCLUSION MUTUELLE

On appelle processus indépendants des processus ne faisant appel qu'à des ressources locales. On appelle processus parallèles pour une ressource des processus pouvant utiliser simultanément cette ressource. Lorsque la ressource est **critique** (ou en accès exclusif), on parle d'**exclusion mutuelle** (par exemple, sur une machine monoprocesseur, l'UC est une ressource en exclusion

Def.: On appelle **section critique** la partie d'un programme où la ressource est seulement accessible par le processus en cours. **Il faut s'assurer que deux processus n'entrent jamais en même temps en section critique sur une même ressource. A ce sujet, aucune hypothèse ne doit être faite sur les vitesses relatives des processus**.

Exemple : la mise à jour d'un fichier (deux mises à jour simultanées d'un même compte client).La section critique comprend :

- lecture du compte dans le fichier,
- modification du compte,
- réécriture du compte dans le fichier.

Def.: *programmation multitâche* : ensemble de plusieurs processus séquentiels dont les exécutions sont imbriquées.

Règle 1: les processus doivent être en relation fortuite. **La défaillance d'un processus en dehors d'une section critique ne doit pas affecter les autres processus**.

Règle 2: un programme multitâche est *juste* s'il répond aux critères de *sécurité* comme l'exclusion mutuelle.

Def.: On appelle *interblocage* la situation où tous les processus sont bloqués.

Ex.: chacun attend que l'autre lui envoie un message pour continuer.

Ex.: chacun exécute une boucle d'attente en attendant une ressource disque indisponible. C'est la situation d'un carrefour avec priorité à droite pour tous et des arrivées continues de véhicules.

Def.: on appelle *privation* la situation où quelques processus progressent normalement en bloquant indéfiniment d'autres processus. C'est la situation d'un carrefour giratoire avec 4 voies d'accès dont deux ont des arrivées continues de véhicules.

Règle 3 : Un programme multitâche est *juste* s'il répond aux critères de *viabilité* comme la non privation ou le non interblocage.

Examinons quelques solutions au problème de l'exclusion mutuelle. On s'intéresse symboliquement ici au cas de deux processus P_1 et P_2 exécutant chacun une boucle infinie divisée en deux parties :

- la section critique ($crit_1$, $crit_2$ respectivement),
- le reste du programme ($reste_1$, $reste_2$ respectivement).

Les exécutions de $crit_1$ et de $crit_2$ ne doivent pas être interrompues et ne doivent pas se chevaucher.

Les algorithmes peuvent être classés en deux catégories :

- **algorithmes par attente active :**

  * masquage des interruptions : le processus qui entre en section critique masque les interruptions et ne peut donc plus être désalloué (exemple : accès du SE à la table des processus). Mais s'il y a plusieurs processeurs ou si le processus "oublie" de démasquer les interruptions, il y a problème !

  * algorithmes 2.1 à 2.4

En ce cas, les processus restent dans la liste des processus prêts puisqu'ils ont toujours quelque chose à tester : d'où une consommation inutile de temps UC

- **algorithmes par attente passive :** les sémaphores

2.1 La méthode des coroutines

Algorithme :

```
int tour ;      /* variable de commutation de droit à la section critique */
                /* valeur 1 pour P1, 2 pour P2                            */
        main ()
{
        tour = 1;               /* P1 peut utiliser sa section critique   */
        parbegin
                p1();
```

```
            p2() ;
        parend
}
/*********************************************************************/
p1()
{
        for ( ; ; )
        {
                while (tour == 2);      /* c'est au tour de P2 ; P1 attend   */
                crit1;
                tour = 2;               /* on redonne l'autorisation à P2    */
                reste1;
        }
}
/*********************************************************************/
p2()
{
        for (; ; )
        {
                while (tour == 1);   /* c'est au tour de P1 ; P2 attend    */
                crit2;
                tour = 1;            /* on redonne l'autorisation à P1    */
                reste2 ;
        }
}
```

Avantages :

- l'exclusion mutuelle est satisfaite. Pour chaque valeur de *tour* , une section critique et une seule peut s'exécuter, et ce jusqu'à son terme.

- l'interblocage est impossible puisque *tour* prend soit la valeur 1, soit la valeur 2 (Les deux processus ne peuvent pas être bloqués en même temps).

- la privation est impossible: un processus ne peut empêcher l'autre d'entrer en section critique puisque *tour* change de valeur à la fin de chaque section critique.

Inconvénients :

- P_1 et P_2 sont contraints de fonctionner avec la même fréquence d'entrée en section critique

- Si l'exécution de P_2 s'arrête, celle de P_1 s'arrête aussi: le programme est bloqué. La dépendance de fonctionnement entre P_1 et P_2 leur confère le nom de **coroutines.**

2.2 Seconde solution

Chaque processus dispose d'une clé d'entrée en section critique (c1 pour P_1, c_2 pour P_2). P_1 n'entre en section critique que si la clé c_2 vaut 1. Alors, il affecte 0 à sa clé c_1 pour empêcher P_2 d'entrer en section critique.

Algorithme :

```
int c1, c2 ;
/* clés de P1 et P2 - valeur 0: le processus est en section critique */
/*                    valeur 1: il n'est pas  en section critique    */
 main ()
```

```
{
        c1=c2 = 1;    /* initialement, aucun processus en section critique */
        parbegin
                p1() ;
                p2 ();
        parend
}
/*********************************************************************/
 p1 ()
 {
        for ( ; ; )
        {
                while (c2 == 0);       /* P2 en section critique, P1 attend  */
                c1 = 0                 /* P1 entre en section critique       */
                crit1 ;
                c1 = 1;                /* P1 n'est plus en section critique  */
                reste1 ;
        }
}
/*********************************************************************/
p2 ()
{
        for ( ; ; )
        {
                while (c1 == 0);       /* P1 en section critique, P2 attend  */
                c2 = 0;                /* P2 entre en section critique       */
                crit2 ;
                c2 = 1 ;               /* P2 n'est plus en section critique  */
                reste2;
        }
}
```

Avantage : on rend moins dépendants les deux processus en attribuant une clé de section critique à chacun.

Inconvénients : Au début c_1 et c_2 sont à 1. P_1 prend connaissance de c_2 et met fin à la boucle while. Si la commutation de temps a lieu à ce moment, c_1 ne sera pas à 0 et P_2 évoluera pour mettre c_2 à 0, tout comme le fera irrémédiablement P_1 pour c_1.
La situation $c_1 = c_2 = 0$ qui en résultera fera entrer simultanément P_1 et P_2 en section critique : **l'exclusion mutuelle ne sera pas satisfaite.**

Si l'instruction $c_i = 0$ était placée avant la boucle d'attente, l'exclusion mutuelle serait satisfaite, mais on aurait cette fois interblocage.
A un moment donné, c_1 et c_2 seraient nuls simultanément P_1 et P_2 exécuteraient leurs boucles d'attente indéfiniment.

2.3 Troisième solution

Lorsque les deux processus veulent entrer en section critique au même moment, l'un des deux renonce temporairement.

Algorithme :

```
int  c1,c2;            /* 0 si le processus veut entrer en section critique,  1 sinon      */
```

```
main ()
{
        c1= c2 = 1;     /* ni P1,ni P2 ne veulent entrer en section critique au départ */
        parbegin
                p1();
                p2 () ;
        parend
}
/*************************************************************************/
 p1()
{
         for ( ; ; )
        {
                 c1 = 0;                /* P1 veut entrer en section critique            */
                while (c2 == 0)     /* tant que P2 veut aussi entrer en section critique ... */
                {
                        c1 = 1;      /* ....P1 abandonne un temps son intention... */
                        c1 = 0;      /* ..... puis la réaffirme                          */
                 }
                crit1;
                c1 = 1;                /* fin de la section critique de P1              */
                 reste1 ;
        }
}
/*************************************************************************/
 p2()
 {
         for ( ; ; )
        {
                c2 = 0 ;                /* P2 veut entrer en section critique            */
                while (c1 == 0)     /* tant que P1 veut aussi entrer en section critique ... */
                 {
                        c2 = 1;      /* .. P2 abandonne un temps son intention .... */
                        c2 = 0 ;     /* ..... puis la réaffirme                          */
                 }
                crit2;
                c2 = 1 ;                /*  fin de la section critique de P2             */
                reste2 ;
        }
}
```

Commentaires :

- **l'exclusion mutuelle est satisfaite**. (cf. ci-dessus)

- il est possible d'aboutir à la situation où c_1 et c_2 sont nuls simultanément. Mais il n'y aura pas interblocage car cette situation instable ne sera pas durable (Elle est liée à la commutation de temps entre $c_i = 1$ et $c_i = 0$ dans while).

- il y aura donc d'inutiles pertes de temps par **famine limitée**.

2.4 Algorithme de DEKKER

DEKKER a proposé un algorithme issu des avantages des 1ère et 3ème solutions pour résoudre l'ensemble du problème sans aboutir à aucun inconvénient. Par rapport, à l'algorithme précédent, un processus peut réitérer sa demande d'entrée en section critique, si c'est son tour.

Algorithme :

```
int tour ,   /* valeur i si c'est au tour de Pi de pouvoir entrer en section critique  */
c1,c2 ;      /* valeur 0 si le processus veut entrer en section critique,1 sinon        */
 main ()
{
        c1 = c2 = tour = 1;   /* P1 peut entrer en section critique, mais... */
        parbegin              /* ...  ni P1, ni P2 ne le demandent           */
                p1 () ;
                p2 () ;
        parend
}
 p1 ()
 {
        for ( ; ; )
         {
                c1 = 0;               /*  P1 veut entrer en section critique        */
                while (c2 == 0)       /*  tant que P2 le veut aussi.....            */
                if (tour == 2)        /*  ..... si c'est le tour de P2 .....        */
                {
                        c1 = 1;       /*  ..... P1 renonce .....                    */
                        while (tour == 2) ;  /* ..... jusqu'à ce que ce soit son  tour .....  */
                        c1 = 0;               /*  ..... puis réaffirme son intention        */
                }
                crit1;
                tour = 2;             /*  C'est le tour de P2                       */
                c1 = 1;               /* P1 a achevé sa section critique            */
                 reste1;
        }
}
/*************************************************************************/
p2 ()
 {
         for (; ; )
        {
                c2 = 0;             /* P2 veut entrer en section critique          */
                 while (c1 == 0)    /* tant que P1 le veut aussi .....             */
                 if (tour == 1)     /* ..... si c'est le tour de P1 .....          */
                {
                        c2 = 1;     /* ..... P2 renonce .....                      */
                        while (tour == 1);   /*  ..... jusqu'à ce que ce soit son tour ....  */
                        c2 = 0;              /*  .....  puis réaffirme son intention        */
                 }
                crit2;
                 tour = 1;                   /* C'est le tour de P1                 */
                c2 = 1;                      /* P2 a achevé sa section critique     */
                 reste2;
        }
}
```

Remarques :

- Si p1 veut entrer en section critique ($c_1 = 0$), alors que p2 le veut aussi ($c_2 = 0$), et que c'est le tour de p1 (tour = 1), p1 insistera (*while* ($c_2 == 0$) sera actif). Dans p2, la même boucle aboutira à c2 = 1 (renoncement temporaire de p2).

- On démontre [Ben-Ari pages 51 à 53] que cet algorithme résout l'exclusion mutuelle sans privation, sans interblocage, sans blocage du programme par arrêt d'un processus.

- Cet algorithme peut être généralisé à n processus au prix d'une très grande complexité.

- De manière générale, les algorithmes par attente active présentent un défaut commun : les boucles d'attente et le recours fréquent à la variable *tour* gaspillent du temps UC. Nouvelle idée : mettre en sommeil un processus qui demande à entrer en section critique dès lors qu'un autre processus y est déjà. C'est l'attente passive.

En outre, l'utilisation par les algorithmes de variables globales n'est pas d'une grande élégance.

2.5 Les sémaphores

La résolution des problèmes multi-tâches a considérablement progressé avec l'invention des sémaphores par E.W. DIJKSTRA en 1965. Il s'agit d'un outil puissant, facile à implanter et à utiliser. Les sémaphores permettent de résoudre un grand nombre de problèmes liés à la programmation simultanée, notamment le problème de l'exclusion mutuelle.

Un **sémaphore** est une structure à deux champs :

- une variable entière, ou valeur du sémaphore. Un sémaphore est dit **binaire** si sa valeur ne peut être que 0 ou 1, **général** sinon.
- une file d'attente de processus ou de tâches.

Dans la plupart des cas, la valeur du sémaphore représente à un moment donné le nombre d'accès possibles à une ressource.

Seules deux fonctions permettent de manipuler un sémaphore :

- **P (s)** ou down (s) ou WAIT (s)
- **V (s)** ou up (s) ou SIGNAL (s)

2.5.1 P (s)

La fonction P(S) décrémente le sémaphore d'une unité à condition que sa valeur ne devienne pas négative.

```
début
   a ← 1 /* a est un registre */
   TestAndSet (&a, &verrou)  /* copie le contenu de verrou (variable globale initialisée à 0) dans a
                et range 1 dans le mot mémoire verrou. TestAndSet est ininterruptible Exécutée par
le matériel, elle permet de lire et d'écrire un mot mémoire */
   tant que a = 1 /* si a = 0, le verrou est libre et on passe, sinon le verrou est mis */
         TestAndSet (&a, &verrou)
   fin tant que
   si  (valeur du sémaphore > 0)
         alors   décrémenter cette valeur
   sinon  - suspendre l'exécution du processus en cours qui a appelé P(s),
         - placer le processus dans la file d'attente du  sémaphore,
```

- le processus passe de l'état ACTIF à l'état ENDORMI.

finsi
verrou ← 0
fin

2.5.2 V (s)

La fonction V(S) incrémente la valeur du sémaphore d'une unité si la file d'attente est vide et si cette incrémentation est possible.

```
début
        a ← 1  /* a est un registre */
        TestAndSet (&a, &verrou)  /* cf. ci-dessus*/
        tant que a = 1 /* si a = 0, le verrou est libre et on passe, sinon le verrou est mis */
                TestAndSet (&a, &verrou)
        fin tant que
        si  (file d'attente non vide)
                alors
                        - choisir un processus dans la file d'attente du  sémaphore,
                        - réveiller ce processus. Il passe de l'état ENDORMI à  l'état ACTIVABLE.
        sinon    incrémenter la valeur du sémaphore si c'est possible
         finsi
        verrou ← 0
fin
```

Remarques:

1. Du fait de la variable globale ***verrou***, les fonctions **P** et **V** sont **ininterruptibles** (on dit aussi **atomiques**). **Les deux fonctions P et V s'excluent mutuellement.** Si P et V sont appelées en même temps, elles sont exécutées l'une après l'autre dans un ordre imprévisible. Dans un système multiprocesseur, les accès à la variable partagée ***verrou*** peuvent s'entrelacer. Il est donc nécessaire de verrouiller le bus mémoire chaque fois qu'un processeur exécute TestAndSet.

2. On supposera toujours que le processus réveillé par V est **le premier entré** dans la file d'attente, donc celui qui est en tête de la file d'attente.

3. L'attente active sur ***a*** consomme du temps UC.

2.5.3 Application des sémaphores à l'exclusion mutuelle

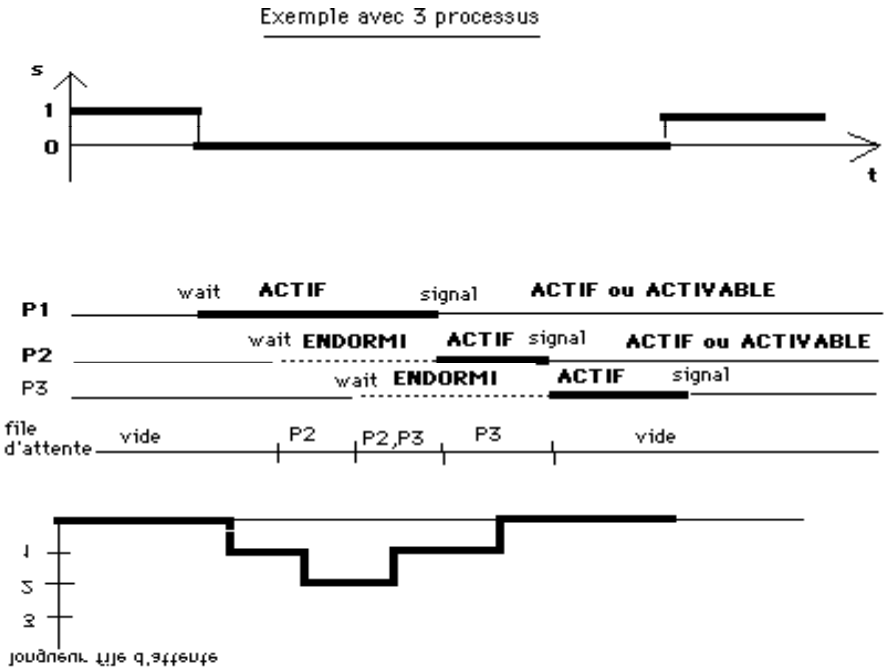
Avec le schéma de programme précédent :

```
SEMAPHORE s;   /* déclaration très symbolique */
main ()
 {
        SEMAB (s,1);    /* déclaration très symbolique ; initialise le sémaphore binaire s à 1 */
        parbegin
                p1 () ;
                p2 () ;       /* instructions très symboliques
        parend                                                  */
 }
/*************************************************************************/
p1 ()       /*  premier processus   */
 {
        for ( ; ; )
        {
                        P (s);
```

```
                    ..... /* section critique de p1        */
                    V (s);
                    ..... /* section non critique de p1 */
        }
}
/*************************************************************************/
p2 ()    /*  second processus   */
{
        for ( ; ; )
        {
            P (s);
                    ..... /* section critique de p2        */
            V (s);
            ..... /* section non critique de p2   */
}
}
```

La démonstration formelle que l'algorithme résout l'exclusion mutuelle et ne crée pas d'interblocage est donnée dans Ben-Ari page 63. Si l'on étend cet algorithme à n processus (n > 2) , il peut y avoir privation (par exemple, P1 et P2 peuvent se réveiller mutuellement et suspendre indéfiniment P3 ou d'autres processus).

J.M. Morris a proposé en 1979 un algorithme de résolution de l'exclusion mutuelle sans privation pour n processus dans *Information Processing Letters, volume 8 , pages 76 à 80* .



3. COMMUNICATION INTER-PROCESSUS

Les processus ont besoin de communiquer, d'échanger des informations de façon plus élaborée et plus structurée que par le biais d'interruptions. Un modèle de communication entre processus avec partage de zone commune (tampon) est le **modèle producteur-consommateur.**

Le producteur doit pouvoir ranger en zone commune des données qu'il produit en attendant que le consommateur soit prêt à les consommer. Le consommateur ne doit pas essayer de consommer des données inexistantes.

Hypothèses :
- les données sont de taille constante
- les vitesses respectives des deux processus (producteur consommateur) sont quelconques.

Règle 1 : Le producteur ne peut pas ranger un objet si le tampon est **plein**
Règle 2 : Le consommateur ne peut pas prendre un objet si le tampon est **vide.**

PRODUCTEUR

```
Faire toujours
        produire un objet
        si nb d'objets ds tampon < N
                alors déposer l'objet ds le tampon
        finsi
Fait
```

CONSOMMATEUR

```
Faire toujours
        si nb d'objets ds tampon >0
                alors prendre l'objet
                      consommer l'objet
        finsi
Fait
```

Règle 3 : exclusion mutuelle au niveau de l'objet : le consommateur ne peut prélever un objet que le producteur est en train de ranger.

Règle 4 : si le producteur (resp. consommateur) est en attente parce que le tampon est plein (resp. vide), il doit être averti dès que cette condition cesse d'être vraie.

Le tampon peut être représenté par une liste circulaire. On introduit donc deux variables caractérisant l'état du tampon :

NPLEIN : nombre d'objets dans le tampon (début : 0)
NVIDE : nombre d'emplacements disponibles dans le tampon (N au début).

PRODUCTEUR :

```
Faire toujours
        Produire un objet       /* début d'atome ininterruptible      */
        si NVIDE >0             /* s'il existe au moins un emplacement vide dans le tampon */
                alors NVIDE --
                sinon s'endormir
        finsi                           /* fin d'atome ininterruptible       */
        ranger l'objet dans le tampon   /* début d'atome ininterruptible   */
        si consommateur endormi
                alors réveiller le consommateur
                sinon  NPLEIN ++
        finsi
Fait
```

CONSOMMATEUR :

```
Faire toujours
        si NPLEIN > 0   /* s'il existe au moins un objet dans le tampon */
                alors NPLEIN --
                sinon   s'endormir
        finsi
        prélever l'objet dans le tampon
        si producteur endormi
                 alors réveiller le producteur
                sinon  NVIDE ++
         finsi
        consommer l'objet
Fait
```

3.1 Solution avec des sémaphores

On peut considérer NVIDE et NPLEIN comme des sémaphores :

PRODUCTEUR

```
Faire toujours
        produire un objet
        P  (NVIDE)
        déposer un objet
        V (NPLEIN)
Fait
```

CONSOMMATEUR

```
Faire toujours
P  (NPLEIN)
prélever un objet
V  (NVIDE)
consommer l'objet
Fait
```

On démontre que le producteur et le consommateur ne peuvent être bloqués simultanément.

Cas où le nombre de producteur (consommateur) est supérieur à 1

Si plusieurs producteurs (consommateurs) opèrent sur le même tampon, il faut assurer l'exclusion mutuelle dans l' opération **déposer un objet** (**prélever un objet)** afin que le pointeur queue (tete) garde une valeur cohérente, de même que pour les objets pointés par queue (tete).

Si l'on veut s'assurer de plus qu'il n'y aura aucun problème dans l'accès au tampon, on peut décider que les opérations **prélever** et **déposer** ne s'exécutent pas simultanément. Déposer et prélever doivent donc figurer en section critique pour protéger les valeurs ressources (tampon, queue, tete). D'où l'utilisation d'un **sémaphore binaire :**

PRODUCTEUR

```
produire un objet
P  (NVIDE)
P  (MUTEX)
tampon[queue] = objet
queue = (queue ++) % N
V (MUTEX)
V (NPLEIN)
```

CONSOMMATEUR

```
P  (NPLEIN)
P  (MUTEX)
objet = tampon [tete]
tete = (tete ++) % N
V  (MUTEX)
V  (NVIDE)
consommer l'objet
```

3.2 Solution avec un compteur d'événements

D.P. REED et R.K. KANODIA ont proposé en 1979 une solution qui utilise une variable entière appelée compteur d'événements. Trois primitives permettent de manipuler une variable compteur d'événements E, commune à tous les processus concernés :

- Read (E) donne la valeur de E

- Advance (E) incrémente E de 1 de manière atomique
- Await (E , v) attend que E ≥ v

```
constante TAILLE   /* nombre de places dans le tampon */
compteur_d_événements    in = 0,     /* nb d'objets mis dans le tampon */
                         out  = 0   /* nb d'objets retirés du tampon    */
producteur
        Faire toujours
                produire l'objet suivant
                nb_produits ++    /* nb_produits : nombre d'objets produits */
                await (out, nb_produits - TAILLE)
                mettre l'objet en position (nb_produits - 1) % TAILLE
                advance (in)
        Fait


consommateur
        Faire toujours
                nb_retirés ++    /* nb_retirés : nombre d'objets retirés */
                await (in, nb_retirés)
                retirer l'objet en position (nb_retirés - 1) % TAILLE
                advance (out)
                consommer l'objet
        Fait
```

3.3 Solution avec un moniteur

On peut aisément imaginer qu'une erreur de programmation des primitives **P()** et **V()** ou l'oubli d'une de ces primitives peut être très grave : interblocage, incohérence des données, etc... On ne peut donc pas concevoir tout un système d'exploitation uniquement à partir des sémaphores. Un outil de programmation plus sûr s'avère nécessaire.

Définition : Un moniteur est une structure de variables et de procédures pouvant être paramétrée et partagée par plusieurs processus. Ce concept a été proposé par C.A.R. HOARE en 1974 et P. BRINCH-HANSEN en 1975. Le type moniteur existe dans certains langages de programmation, tels que Concurrent Pascal.

Le corps du moniteur est exécuté dès que le programme est lancé pour i**nitialiser** les variables du moniteur. Les variables moniteur ne sont accessibles qu'à travers les procédures moniteur.

La seule manière pour un processus d'accéder à une variable moniteur est d'appeler une procédure moniteur.

On peut prévoir plusieurs moniteurs pour différentes tâches qui vont s'exécuter en parallèle. Chaque moniteur est chargé d'une tâche bien précise et chacun a ses données et ses instructions réservées. Si un moniteur M1 est le seul moniteur à avoir accès à la variable u1, on est sûr que u1 est en exclusion mutuelle. De plus, comme les seules opérations faites sur u1 sont celles programmées dans M1, il ne peut y avoir ni affectation, ni test accidentels.

On dit que l'entrée du moniteur par un processus exclut l'entrée du moniteur par un autre processus.
Les moniteurs présentent plusieurs avantages :

- au lieu d'être dispersées dans plusieurs processus, les sections critiques sont transformées en procédures d'un moniteur
- la gestion des sections critiques n'est plus à la charge de l'utilisateur, mais elle est réalisée par le moniteur, puisqu'en fait le moniteur tout entier est implanté comme une section critique.

Des exemples de moniteurs sont donnés dans Beauquier, p. 139-141.

Il utilise des variables de type **condition** et deux primitives agissant sur elles :
- WAIT : bloque le processus appelant et autorise un processus en attente à entrer dans le moniteur

- SIGNAL : réveille le processus endormi en tête de la file d'attente. Puis, ou bien le processus courant est endormi (solution de Hoare), ou bien le processus courant quitte le moniteur (solution de Brinch Hansen, la plus usitée), afin qu'il n'y ait pas deux processus actifs dans le moniteur.

Voici une solution du moniteur du problème producteur-consommateur :

```
moniteur ProdCons /* moniteur, condition : types prédéfinis */
        condition plein, vide
        int compteur
        /* début du corps du moniteur */
        compteur := 0
        /* fin du corps du moniteur
procédure ajouter ()
{
        if compteur = N  then WAIT (plein) /* seul un SIGNAL (plein) réveillera le processus */
        ..... /* ranger l'objet dans le tampon */
        compteur ++
        if compteur = 1 then SIGNAL (vide)
        /* réveille un processus endormi parce que le tampon était vide */
}

procédure retirer ()
{
        if compteur = 0 then WAIT (vide) /* seul un SIGNAL (vide) réveillera le processus */
        ..... /* retirer l'objet du tampon */
        compteur --
        if compteur = N-1 then SIGNAL (plein)
        /* réveille un processus endormi parce que le tampon était plein */
}
fin du moniteur

procédure producteur ()
{
        faire toujours
                produire (élément)
                ProdCons . ajouter ()
        fin faire
}


procédure consommateur ()
{
        faire toujours
                retirer (élément)
                ProdCons . retirer ()
        fin faire
}
```

3.4 Solution avec échanges de messages

Certains ont estimé que les sémaphores sont de trop bas niveau et les moniteurs descriptibles dans un nombre trop restreint de langages. Ils ont proposé un mode de communication inter-processus qui repose sur deux primitives qui sont des appels système (à la différence des moniteurs) :

- send (destination , &message)
- receive (source , &message), où source peut prendre la valeur générale ANY

Généralement, pour éviter les problèmes dans les réseaux, le récepteur acquitte le message reçu. L'émetteur envoie à nouveau son message s'il ne reçoit pas d'acquittement. Le récepteur, s'il reçoit deux messages identiques, ne tient pas compte du second et en tire la conclusion que l'acquittement s'est perdu.

Dans le contexte d'un mode client-serveur, le message reçu contient le nom et les paramètres d'une procédure à lancer. Le processus appelant se bloque jusqu'à la fin de l'exécution de la procédure et le message en retour contient la liste des résultats de la procédure. On parle d'appel de procédure à distance.

On peut proposer une solution au problème producteur-consommateur par échange de messages avec les hypothèses suivantes :

- les messages ont tous la même taille
- les messages envoyés et pas encore reçus sont stockés par le SE dans une mémoire tampon
- le nombre maximal de messages est N
- chaque fois que le producteur veut délivrer un objet au consommateur, il prend un message vide, le remplit et l'envoie. Ainsi, le nombre total de messages dans le système reste constant dans le temps
- si le producteur travaille plus vite que le consommateur, tous les messages seront pleins et le producteur se bloquera dans l'attente d'un message vide ; si le consommateur travaille plus vite que le producteur, tous les messages seront vides et le consommateur se bloquera en attendant que le producteur en remplisse un.

producteur

```
Faire toujours
        produire_objet (&objet)                 /* produire un nouvel objet          */
        receive (consommateur , &m)             /* attendre un message vide          */
        faire_message (&m , objet)              /* construire un message à envoyer   */
        send (consommateur , &m)                /* envoyer le message                */
Fait
```

consommateur

```
pour (i = 0 ; i < N ; i++) send (producteur , &m)   /* envoyer N messages vides  */
Faire toujours
        receive (producteur , &m)               /* attendre un message           */
        retirer_objet (&m , &objet)             /* retirer l'objet du message    */
        utiliser_objet (objet)
        send (producteur , &m)                  /* renvoyer une réponse vide     */
Fait
```

On peut également imaginer une solution de type boîte aux lettres de capacité N messages , avec un producteur se bloquant si la boîte est pleine et un consommateur se bloquant si la boîte est vide.

3.5 Propriétés des solutions précédentes

On démontre les résultats d'équivalence suivants entre sémaphores, moniteurs et échanges de messages :

- on peut utiliser des sémaphores pour construire un moniteur et un système d'échanges de messages
- on peut utiliser des moniteurs pour réaliser des sémaphores et un système d'échanges de messages
- on peut utiliser des messages pour réaliser des sémaphores et des moniteurs

3.6 Le problème des philosophes

Il s'agit d'un problème très ancien, dont DIJKSTRA a montré en 1965 qu'il modélise bien les processus en concurrence pour accéder à un nombre limité de ressources. *"5 philosophes sont assis autour d'une table ronde. Chaque philosophe a devant lui une assiette de spaghettis si glissants qu'il lui faut deux fourchettes pour les manger. Or, il n'y a qu'une fourchette entre deux assiettes consécutives. L'activité d'un philosophe est partagée entre manger et penser. Quand un philosophe a faim, il tente de prendre les deux fourchettes encadrant son assiette. S'il y parvient, il mange, puis il se remet à penser. Comment écrire un algorithme qui permette à chaque philosophe de ne jamais être bloqué ? "*

Il faut tout à la fois éviter les situations :

- d'interblocage : par exemple tous les philosophes prennent leur fourchette gauche en même temps et attendent que la fourchette droite se libère
- de privation : tous les philosophes prennent leur fourchette gauche, et, constatant que la droite n'est pas libre, la reposent, puis prennent la droite en même temps, etc...

Voici une solution (une autre est donnée dans Beauquier p. 155-156) :

```
#define  N    5                              /* nombre de philosophes  */
#define  GAUCHE     (i - 1) % N              /* n° du voisin gauche de i */
#define  DROITE     (i + 1) % N              /* n° du voisin droite de     i */
#define PENSE   0                           /* il pense  */
#define FAIM       1                        /* il a faim   */
#define  MANGE 2                            /* il mange  */
typedef  int  semaphore;
int  etat [N];                          /* pour mémoriser les états des philosophes */
semaphore  mutex = 1,                       /* pour section critique */
                s [N];                      /* un sémaphore par philosophe */
/*****************************/
void philosophe (int  i)
{
        while (TRUE)
        {
                penser ();
                prendre_fourchettes (i);        /* prendre 2 fourchettes ou se bloquer */
                manger ();
                poser_fourchettes (i);
        }
}
/*****************************/
void prendre_fourchettes (int  i)
{
        P (mutex);                              /* entrée en section critique */
        etat [i] = FAIM;
        test (i);                               /* tentative d'obtenir 2 fourchettes */
        V (mutex);                              /* sortie de la section critique */
        P (s [i]);                              /* blocage si pas 2 fourchettes */
}

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

```
void poser_fourchettes (int i)
{
        P (mutex);                                      /* entrée en section critique */
        etat [i] = PENSE;
        test (GAUCHE);
        test (DROITE);
        V (mutex) ;                                     /* sortie de la section critique */
}
/*****************************/
void test (int  i)
{
        if (etat [i] == FAIM  &&  etat [GAUCHE] != MANGE  && etat [DROITE] != MANGE)
        {
                etat [i] = MANGE;
                V (s [i] );
        }
}
```

3.7 Le problème des lecteurs et des rédacteurs

Il s'agit d'un autre problème classique dont P.J. COURTOIS a montré en 1971 qu'il modélise bien les accès d'un processus à une base de données. On peut accepter que plusieurs processus lisent la base en même temps; mais dès qu'un processus écrit dans la base, aucun autre processus ne doit être autorisé à y accéder, en lecture ou en écriture.

Voici une solution qui donne la priorité aux lecteurs (COURTOIS a proposé une solution qui donne la priorité aux rédacteurs) :

```
typedef int  semaphore;
int  rc  = 0;                   /* nb de processus qui lisent ou qui veulent écrire */
semaphore  mutex = 1,                                   /* pour contrôler l'accès à rc     */
                bd = 1;                                 /* contrôle de l'accès à la base */
/*****************************/
void lecteur ()
{
        while (TRUE)
        {
                P (mutex);                              /* pour obtenir un accès exclusif à rc */
                rc ++;                                  /* un lecteur de plus */
                if (rc == 1) P (bd);                    /* si c'est le 1er lecteur */
                V (mutex);
                lire_base_de_donnees ();
                P (mutex);                              /* pour obtenir un accès exclusif à rc */
                rc --;                                  /* un lecteur de moins */
                if (rc == 0) V (bd);                    /* si c'est le dernier lecteur */
                V (mutex);
                utiliser_donnees_lues ();
        }
}
/*****************************/
void redacteur ()
{
        while (TRUE)
        {
                creer_donnees ();
                P (bd);
                ecrire_donnees ();
                V (bd);
        }
```

}

Autre bonne solution : Beauquier p. 146-148

4. APPLICATION DES SEMAPHORES A LA SYNCHRONISATION

4.1 Synchronisation

Def.: On dit qu'un processus P est **synchronisé** avec un processus Q lorsque l'activité de P (resp. Q) dépend d'un événement modifié par Q (resp. P).

La synchronisation exige la mise en place d'un mécanisme permettant à un processus actif de bloquer un autre processus ou de se bloquer lui-même ou d'activer un autre processus. On distingue:

- les actions directes : le processus agit directement sur le processus cible par des primitives telles que **bloque** (pid) ou **réveill**e (pid) ou encore **dort** () qui s'applique à lui-même

- les actions indirectes : le processus utilise un objet commun, tel que sémaphore

Exemple :

Soient 3 processus P1, P2, P3 chargés du calcul de (a+ b) * (c + d) - (e/f)
P2 calcule c + d, P3 calcule e/f et P1 le résultat.
On initialise les sémaphores s1 et s2 à 0.

P1	P2	P3
t1 = a + b	t2 = c + d	t3 = e/f
• P (s1)	V (s1)	V(s2)
t4 = t1 * t2		
Δ P (s2)		
res = t4 - t3		

P1 ne peut se poursuivre au-delà de • tant que P2 n'a pas exécuté V
P1 ne peut se poursuivre au-delà de Δ tant que P3 n'a pas exécuté V

Plus généralement, si l'on veut empêcher un processus A d'exécuter son instruction K avant qu'un autre processus B n'ait exécuté son instruction J, on initialise un sémaphore binaire s à 0. On place P (s) avant l'instruction K de A et V (s) après l'instruction J de B.

4.2 Interblocage

Si les primitives P et V ne sont pas utilisées correctement (erreurs dans l'emploi, le partage ou l'initialisation des sémaphores), un problème peut survenir.

Exemple : avec deux processus p1 et p2, deux sémaphores s1 et s2
(s1 = s2 = 1)

p1	p2
....	.....
P (S1)	P (S2)
.....	.....
P (S2)	P (S1)

.....
V (S2)

Si p1 fait P (S1) alors S1 = 0
puis p2 fait P (S2) et S2 = 0
puis p1 fait P (S2) et p1 est en attente, ENDORMI
puis p2 fait P (S1) et p2 est ENDORMI

Il y a donc évidemment interblocage (puisque aucun des processus ne peut faire V (On dit aussi étreinte fatale ou deadlock)

BIBLIOGRAPHIE

J. BEAUQUIER, B. BERARD, Systèmes d'exploitation, Ediscience, 1993

M. BEN-ARI, Processus concurrents, Masson 1986

A. SCHIPER, Programmation concurrente, Presses Polytechnique Romandes 1986

A. TANENBAUM, Les systèmes d'exploitation, Prentice Hall, 1999

Exercices sur le chapitre 4

1. Expliquer pourquoi l'exclusion mutuelle n'est plus assurée si les opérations P et V sur un sémaphore ne sont plus atomiques.

2. Une machine possède une instruction swap qui échange de manière atomique le contenu de deux variables entières. Soient les variables suivantes globales à n processus :

```
int libre = 0;
int tour [n];    /* tableau initialisé à 1 */
```

Le code source du processus n° i (i = 1 à n) est :

```
main ()
{
        while (1)
        {
                do {
                swap (libre, tour [i]);
                i++;
                } while (tour [i-1] == 1);
                /* puis section critique */
                ……
                swap (libre, tour [i]);
                /* puis section non critique */
        }
}
```

L'exclusion mutuelle est-elle satisfaite ?

Un processus désirant entrer en section critique peut-il attendre indéfiniment cette entrée ?

Chapitre 6 LES ENTREES-SORTIES

1. ENTREES-SORTIES PHYSIQUES

Les principes et les caractéristiques des périphériques ont été vus dans le cours de Technologie des Ordinateurs.

Dans certains ordinateurs, des instructions permettent la redirection des données issues de l'UC, non vers la mémoire, mais vers un périphérique (solution INTEL); ou bien un accès mémoire à une adresse réservée particulière est interprétée comme demande d'accès à un périphérique (solution Motorola).

Trois solutions sont utilisées pour faire communiquer l'UC et les périphériques :

1.1 Contrôleurs

Chaque périphérique est connecté à l'ordinateur par l'intermédiaire d'une carte électronique appelée **interface** ou **adaptateur** ou **contrôleur de périphérique**, qui transforme les signaux du périphérique en signaux adaptés à l'UC et vice-versa. Un contrôleur peut gérer un ou plusieurs périphériques. Le SE communique donc avec le contrôleur et non avec le périphérique lui-même. Les petits ordinateurs utilisent des liaison à **bus** entre les contrôleurs, la mémoire et l'UC.

Toutes les données transitent par l'UC qui gère toutes les phases de la transmission, du contrôle et de la synchronisation.

Exemple : imprimantes, clavier/écran sur PC.

1.2 DMA

Pour éviter le ralentissement de l'UC par ces tâches de bas niveau, on utilise parfois l'**accès direct à la mémoire (DMA)**. Il suffit de donner au contrôleur l'adresse où il doit accéder en mémoire, l'opération (lecture ou écriture), le nombre d'octets à transférer entre la mémoire principale et le tampon du contrôleur. Le contrôleur utilise les "temps morts" du bus (**vol de cycle** ou cycle stealing). L'UC conserve la responsabilité des fonctions de commande, de contrôle et une partie de la synchronisation. En fin de transfert, le contrôleur émet une interruption.

Exemple : disque sur PC.

1.3 Canal

Sur les gros ordinateurs, des **canaux d'E/S** allègent le travail du processeur principal pour sa communication avec les contrôleurs (contrôle et synchronisation). Un canal d'E/S est un processeur spécialisé qui gère soit un seul périphérique rapide, soit plusieurs périphériques multiplexés. Un canal utilise le DMA.

L'UC envoie au canal l'adresse en mémoire centrale du début du programme de transfert à exécuter (**programme canal**). L'UC peut interrompre l'exécution, scruter l'état d'avancement par consultation de registres du canal. Un canal est une solution adaptée aux périphériques rapides des mini- et gros ordinateurs.

2. PERIPHERIQUES VIRTUELS D'ENTREE-SORTIE

2.1 Mécanisme des périphériques virtuels

L'objectif est de rendre tout programme utilisateur indépendant des types de périphériques , nombreux et en évolution constante, et des contrôles qui en découlent. D'où la création de **périphériques virtuels** ou **flots d'E/S** ou **fichiers d'E/S**. Sous UNIX, par exemple, une console, un disque ou un fichier ont la même interface d'accès.

Les opérations possibles sur un flot sont :
- ouverture et association physique (open en C)
- fermeture et dissociation (close en C)
- lecture (read en C)
- écriture (write en C)
- paramétrage et contrôle (ioctl , seek en C)

Le système gère une table des flots par processus dans le BCP.

La concordance entre un périphérique virtuel et un périphérique physique associé est assurée par un **pilote de périphérique** (device driver) : soit un type de pilote par type de périphérique, soit un seul type de pilotes configurable par des **descripteurs de périphériques**. Un pilote est la seule partie du SE à connaître les registres du contrôleur, les secteurs et leur facteur d'entrelacement, les pistes, les cylindres, les têtes, les déplacements des bras, les moteurs, le temps de positionnement des têtes, etc..., ainsi que le traitement des erreurs.

2.2 Fonctionnement d'un périphérique virtuel

Un processus P demande une E/S avec un périphérique. La correspondance périphérique virtuel-périphérique physique est ainsi résolue :

- appel système : les paramètres de l'appel sont empilés et un appel système (Supervisor Call = requête SVC) est généré

- action du superviseur : il identifie la nature de l'appel et récupère les paramètres (nom du flot, nombre d'octets, adresse de transfert); puis il identifie le pilote concerné et le périphérique physique. Enfin, il appelle le pilote

- le pilote : le contexte du processus actif est commuté après sauvegarde. La procédure de traitement des E/S est exécutée. Ensuite, le pilote envoie une interruption pour le signaler. Le processus initial est rechargé et réactivé.

Comme il doit y avoir possibilité d'E/S simultanées, un pilote doit être réentrant. En outre, chaque pilote gère une table des processus en attente.

3. PROBLEMES D'INTERBLOCAGE DES PERIPHERIQUES

Avec des périphériques partagés (disques, ...), des problèmes d'interblocage peuvent survenir. Citons deux exemples :

- un processus A demande le dérouleur et un processus B demande le traceur de courbe. Les deux demandes sont prises en compte. Puis A demande le traceur sans libérer le dérouleur et B demande le dérouleur sans libérer le traceur

- un processus A verrouille l'enregistrement R1 d'une BD pour éviter les conflits d'accès et un processus B verrouille l'enregistrement R2. Puis chacun essaie de verrouiller l'enregistrement verrouillé par l'autre

COFFMAN, ELPHICK et SHOSHANI ont montré en 1971 qu'il faut réunir 4 conditions pour provoquer un interblocage :

- l'exclusion mutuelle : chaque ressource est soit disponible, soit attribuée à un seul processus

- la détention et l'attente : les processus qui détiennent des ressources peuvent en demander de nouvelles

- pas de réquisition : les ressources obtenues par un processus ne peuvent pas lui être retirées, mais il doit les libérer explicitement

- l'attente circulaire : il doit y avoir au moins deux processus, chacun d'eux attendant une ressource détenue par l'un des autres

Il existe une solution à chacune des 4 conditions précédentes pour prévenir les interblocages :

3.1 Exclusion mutuelle : usage d'un spoule

Considérons une imprimante avec un fichier tampon associé. Un processus peut ouvrir ce fichier et ne rien imprimer pendant des heures, bloquant tous les autres processus voulant accéder à la ressource.

On crée un processus particulier appelé **démon** (daemon) et un répertoire spécial, le **répertoire de spoule** (spool). Pour imprimer un fichier, un processus doit d'abord le lier au répertoire de spoule. Le démon, seul processus à être autorisé à accéder au fichier spécial de l'imprimante, imprime les fichiers du répertoire de spoule.

Une technique identique peut être utilisée pour la transmission de fichiers à travers un réseau (démon de réseau, répertoire de spoule de réseau).

Mais le spoule ne peut pas résoudre tous les problèmes d'exclusion mutuelle (par exemple, spoule plein et processus inachevés) , ni s'appliquer à tous les périphériques. **On se reportera donc aux algorithmes du chapitre précédent.**

3.2 Détention et attente : demande préalable de ressources

Si l'on oblige tout processus à demander à l'avance les ressources nécessaires, on n'activera le processus que lorsque toutes les ressources seront disponibles. Cette condition est bien sûr difficile à mettre en pratique. Une alternative : n'accorder une nouvelle ressource que s'il y a libération des anciennes ressources.

3.3 Pas de réquisition : pas de solution !

3.4 Attente circulaire : l'ordonnancement numérique des ressources

On numérote toutes les ressources (exemple : 1 : imprimante, 2 : traceur, 3 : dérouleur, etc...). Un processus peut demander toutes les ressources qu'il souhaite à condition que les numéros des ressources soient croissants (variante : un processus ne peut demander une ressource de numéro

inférieur à la plus haute ressource qu'il détient). Un interblocage se produirait si le processus A demande la ressource i détenue par B et si B demande la ressource k détenue par A. Si i > k, B ne pourra demander k et si i < k, A ne pourra demander i : il n'y aura pas interblocage.

3.5 Attente circulaire : l'algorithme du banquier

En 1965, DIJKSTRA a proposé un algorithme qui permet d'éviter l'interblocage : l'algorithme du banquier, qui reproduit le modèle de prêt à des clients par un banquier.

Considérons les deux tableaux suivants qui résument l'**état d'un ordinateur** à l'instant t :

proc.	P1	P2	P3	P4
A	3	0	1	1
B	0	1	0	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0
total	5	3	2	2

ressources actuellement attribuées

proc.	P1	P2	P3	P4
A	1	1	0	0
B	0	1	1	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

ressources encore demandées

ressources existantes : exist = (6 3 4 2)
ressources disponibles : dispo = exist - total = (1 0 2 0)

5 processus sont actifs (A,B,C,D,E) et il existe 4 catégories de périphériques P1 à P4 (exemple : P4 = imprimante). Le tableau de gauche donne les ressources déjà allouées et le tableau de droite les ressources qui seront encore demandées pour achever l'exécution.

Un état est dit **sûr** s'il existe une suite d'états ultérieurs qui permette à tous les processus d'obtenir toutes leurs ressources et de se terminer. L'algorithme suivant détermine si un état est sûr :

1. Trouver dans le tableau de droite une ligne L dont les ressources demandées sont toutes inférieures à celles de dispo ($L_i <= dispo_i$, $\forall$ i). S'il n'existe pas L vérifiant cette condition, il y a interblocage

2. Supposer que le processus associé à L obtient les ressources et se termine. Supprimer sa ligne et actualiser dispo

3. Répéter 1 et 2 jusqu'à ce que tous les processus soient terminés (l'état initial était donc sûr) ou jusqu'à un interblocage (l'état initial n'était pas sûr)

Ici, par exemple, l'état actuel est sûr car :
- on allouera à D les ressources demandées et il s'achèvera
- puis on allouera à A ou E les ressources demandées et A ou E s'achèvera
- enfin les autres

L'inconvénient de cet algorithme est le caractère irréaliste de la connaissance préalable des ressources nécessaires à l'achèvement d'un processus. Dans bien des systèmes, ce besoin évolue dynamiquement.

4. LES DISQUES

4.1 Caractéristiques physiques

Un disque est logiquement organisé en cylindres. Un cylindre contient autant de pistes qu'il y a de têtes. Chaque piste est divisée en secteurs(un nombre constant et de taille constante quelque soit le rayon de la piste). Un contrôleur de disques peut effectuer des recherches simultanées sur plusieurs disques, ou bien lire ou écrire sur un disque et attendre des données d'un ou plusieurs autres disques.

Pour le système, une adresse sur disque est formée du triplet : (n° de cylindre, n° de piste, n° de secteur).

Le temps moyen de lecture/écriture sur un secteur est égal à la somme de trois durées élémentaires :

- le temps de recherche : positionnement de la tête sur le bon cylindre
- le temps de latence ou délai rotationnel :pour atteindre le bon secteur (en moyenne 1/2 tour de rotation)
- le temps de transfert réel de l'information

4.2 Ordonnancement des accès

Algorithme FCFS (FIFO) : la gestion la plus simple du mouvement du bras du disque consiste à satisfaire les requêtes d'accès dans l'ordre où elles surviennent. Bien entendu, cet algorithme donne généralement de mauvais temps de réponse. Ainsi, si une suite de requêtes concerne les pistes : 1, 36, 16, 34, 9, 12, alors que les têtes sont sur le cylindre 11, il faudra se déplacer au total de 111 cylindres. Il s'agit de l'algorithme First Come First Served.

Algorithme SSF (ou PCD) : ici, le pilote choisit la requête la plus proche de sa position actuelle (Shortest Seek First, plus courte d'abord, variante du PCTR déjà étudié au chapitre 3, § 3). Avec l'exemple ci-dessus, les cylindres seront traités dans l'ordre : 12, 9, 16, 1, 34, 36. On se sera déplacé de 61 cylindres seulement. Mais les requêtes portant sur des cylindres éloignés peuvent être durablement différées, si d'autres requêtes surviennent pendant le traitement de la liste : l'équitabilité peut souffrir de la réduction du temps de réponse.

Algorithme de l'ascenseur (Look) : pour éviter cet inconvénient, on déplace la tête dans une direction donnée en traitant toutes les requêtes rencontrées, puis le sens du balayage s'inverse et on traite les requêtes rencontrées, etc... Dans l'exemple traité, on obtiendrait l'ordre : 12, 16, 34, 36, 9, 1 et un parcours total de 60 cylindres.

Une variante a été proposée par T.J. TEOREY en 1972 (C-Look, C pour circulaire) : la dernière piste est considérée comme adjacente à la première. La tête, lorsqu'elle est arrivée à une extrémité, retourne immédiatement à l'autre sans traiter de requête. On obtiendrait ici l'ordre : 12, 16, 34, 36, 1, 9, soit 68 cylindres parcourus.

Algorithme PCTL : lorsqu'un disque est fortement sollicité, on trouve fréquemment plusieurs références à une même piste ou à un même cylindre. Les requêtes doivent être ordonnées selon le secteur recherché pour réduire le temps de latence. L'algorithme PCTL (plus court temps de latence) traite les requêtes dans l'ordre de défilement des secteurs concernés sous la tête, pour une piste donnée, quel que soit leur ordre d'arrivée. Par exemple, si la tête se trouve au-dessus du secteur n° 2 et que des requêtes concernent les secteurs 11, 5, 8 et 7, on traitera dans l'ordre les secteurs 5, 7, 8, 11ce qui évitera d'attendre plus d'un tour pour traiter les secteurs 5, 8, 7. On peut parfois gagner en efficacité en entrelaçant les secteurs sur les pistes.

Souvent, les modèles probabilistes sont plus réalistes que les modèles déterministes. Dans les problèmes d'ordonnancement, en particulier, les modèles des files d'attente sont souvent performants. Ils sont bien traités dans BEAUQUIER et BERARD, ch. 10.

4.3 Mémoire cache pour les pistes du disque

En général, le temps de recherche est très supérieur au temps de transfert. Il importe donc peu de lire un secteur ou bien une piste complète pour simplifier le fonctionnement du contrôleur. On utilise une **mémoire cache** pour stocker une piste : mémoire à accès rapide, à accès direct par le contrôleur et par l'UC.

Il est préférable d'implanter cette antémémoire dans le contrôleur, plutôt que dans le pilote, pour que le transfert puisse se faire par DMA

4.4 Traitement des erreurs

Le fonctionnement des disques est soumis à de nombreuses sources d'erreurs :

- erreur de programmation (par exemple : accès à un secteur inexistant) nécessitant un arrêt de la requête
- erreur du contrôle de parité (checksum) : on déclare le secteur endommagé si l'erreur persiste au bout de plusieurs essais. On tient à jour un fichier des secteurs endommagés à ne jamais allouer et à ne jamais copier lors d'une sauvegarde
- erreur de positionnement du bras : un programme de recalibrage est lancé
- erreur de contrôleur

4.5 Disque virtuel

Un disque virtuel utilise une portion de la mémoire centrale pour sauvegarder des secteurs. Il convient bien pour stocker programmes et données fréquemment utilisées, qui seront ainsi accessibles très rapidement.

5. LES TERMINAUX

Il existe de très nombreuses variétés de terminaux. C'est donc au pilote de terminal de masquer ces différences pour qu'un programme soit le plus possible indépendant du terminal utilisé.

5.1 Caractéristiques physiques

Du point de vue du SE, on distingue deux catégories de terminaux :

5.1.1 Les terminaux à interface RS-232 standard

Ils communiquent avec une interface série et son dotés d'un connecteur à 25 broches (une broche pour transmettre les données, une pour en recevoir, une pour la masse et quelques autres pour les contrôles). Généralement un UART (Universal Asynchronous Receiver Transmitter) assure la conversion parrallèle-série des signaux et série-parallèle. Le pilote envoie octet par octet à l'UART et se bloque entre deux transferts (jusqu'à 19200 bits/s).

Les terminaux intelligents actuels possèdent un processeur puissant, une mémoire de grande capacité et peuvent télécharger des programmes à partir de l'ordinateur.

5.1.2 Les terminaux mappés en mémoire (memory-mapped terminals)

Ces terminaux ne communiquent pas avec l'ordinateur par une liaison série, mais font partie de l'ordinateur. Ils sont interfacés par une mémoire particulière (RAM vidéo) qui fait partie de la mémoire de l'ordinateur et est adressée par le processeur comme n'importe quelle partie de la mémoire. Cette solution es plus rapide que la précédente. Un autre composant, le contrôleur vidéo, retire des octets de la RAM vidéo et génère le signal vidéo qui contrôle l'affichage à l'écran.

Dans le cas des terminaux bitmap, chaque bit de la RAM vidéo contrôle un pixel de l'écran.

Le clavier est interfacé par une liaison parallèle, voire une liaison série. A chaque frappe de touche et à chaque relâchement, une interruption est déclenchée et le code de l'emplacement de la touche est placé dans un tampon.

5.2 Le logiciel du clavier

Deux types de pilotes sont envisageables :

- le pilote lit chaque octet du tampon, le convertit en code ASCII à l'aide de tables internes, et le transmet sans interprétation, ou plus souvent le stocke dans un tampon : il s'agit d'une approche caractère (**mode raw**)

- le pilote prend en compte tout ce qui se rapporte à l'édition de ce qui précède la validation (par RC ou NL) : BS, etc... et ne transmet que les lignes corrigées : il s'agit d'une approche ligne (**mode cooked**)

Sous UNIX, le mode **cbreak** est un compromis entre les modes raw et cooked : les caractères sont transmis au processus sans attendre la validation (comme en mode raw), mais DEL, CTRL-S, CTRL-Q et CTRL-\ sont traités comme en mode cooked. La fonction **ioctl** permet de changer de mode.

Il existe deux façons de gérer les tampons des claviers :

- le pilote réserve un ensemble de tampons de taille standard. A chaque terminal est associé un pointeur sur le premier tampon utilisé; l'ensemble des tampons associés à un terminal est organisé en une liste chaînée. Cette méthode est utilisée sur les grosses machines possédant beaucoup de terminaux

- au sein de la structure de données associée à chaque terminal dans la mémoire, on implante un tampon propre au terminal. Le pilote est plus simple. Cette solution convient bien aux petits systèmes ou aux PC.

Les pilotes de claviers doivent également s'acquitter de bien d'autres tâches : gérer la présence ou l'absence d'écho à l'écran, gérer les lignes de longueur supérieure à une ligne d'écran, gérer les caractères de tabulation, produire les caractères RC **et** NL lors d'une validation avec une convention donnée, gérer les caractères spéciaux (CTRL-D, CTRL-Q, CTRL-S, DEL, etc....). Sous UNIX, ces caractères

5.3 Le logiciel de l'écran

Dans le cas d'un terminal RS-232, le programme ou la fonction d'écho de la frappe envoie dans le tampon de l'écran la suite des octets à afficher. Si le tampon est plein ou bien si toutes les données sont transmises, le premier caractère est envoyé au terminal, le pilote est endormi. Une interruption provoque la transmission du caractère suivant, etc...

Dans le cas de terminaux mappés en mémoire, les caractères à afficher sont retirés un à un de l'espace utilisateur et placés dans la RAM vidéo (avec traitement particulier pour certains caractères, comme BELL ou les séquences classiques d'échappement).

Le pilote gère la position courante dans la RAM vidéo, en tenant compte des caractères tels que RC, NL, BS; il gère aussi le défilement (scrolling) lorsqu'un passage à la ligne en bas d'écran survient. Pour cela, il met à jour un pointeur, géré par le contrôleur vidéo, sur le premier caractère de la première ligne à l'écran. Le pilote gère aussi le curseur en tenant à jour un pointeur sur sa position.

Exercices :

1. Soit l'état suivant d'un ordinateur à l'instant t :

proc.	P1	P2	P3	P4
A	3	0	1	1
B	0	1	1	0
C	1	1	1	0
D	1	1	0	1
E	0	0	0	0
total	5	3	3	2

ressources actuellement attribuées

proc.	P1	P2	P3	P4
A	1	1	0	0
B	0	1	0	2
C	3	1	0	0
D	0	0	1	0
E	2	1	1	0

ressources encore demandées

ressources existantes : exist = (6 3 4 2)
ressources disponibles : dispo = exist - total = (1 0 0)

L'état du système est-il sûr ?
Si le processus E fait la demande (0, 0, 1, 0) le système qui utilise l'algorithme du banquier lui attribuera-t-il la ressource P3 ?

2. On exécute un ensemble S de n processus indépendants en parallèle :

$S = p_1 // p_2 // ... // p_n$

Le processus p_1 est constitué de 4 tâches séquentielles : $p_1 = T_1(1)\ T_2(1)\ T_3(1)\ T_4(1)$
Le processus p_2 est constitué de 2 tâches séquentielles : $p_2 = T_1(2)\ T_2(2)$
Le processus p_3 est constitué de 3 tâches séquentielles : $p_3 = T_1(3)\ T_2(3)\ T_3(3)$

Trois ressources sont disponibles : R_1 en quantité $N_1 = 7$, R_2 en quantité $N_2 = 8$,
R_3 en quantité $N_3 = 6$

A chaque tâche $T_i(k)$, on associe deux événements :

$d_i(k)$: initialisation de la tâche (lecture des paramètres d'entrée, acquisition de ressources, chargement d'information)

$f_i(k)$: terminaison de la tâche (écriture des résultats, libération de ressources)

On note $d_i(k)$ $(r_1, ..., r_n)$ l'événement correspondant à l'initialisation de la tâche $T_i(k)$ avec demande de r_1 ressources R_1 ,..., r_n ressources R_n et $f_i(k)$ $(r_1, ..., r_n)$ l'événement correspondant à la terminaison de la tâche $T_i(k)$ avec libération de r_1 ressources R_1 ,..., r_n ressources R_n .
Le comportement suivant est-il valide :

w = d_1 (1) (1,2,3) d_1(3) (1,1,1) f_1 (1) (0,1,2) d_1 (2) (0,2,1) f_1 (2) (0,1,0) d_2 (2) (1,2,0)
f_1 (3) (0,0,1) d_2 (1) (3,2,4) f_2 (2) (1,3,1) f_2 (1) (4,2,5) d_2 (3) (3,0,1) d_3 (1) (2,3,1)
f_3 (1) (2,4,0) d_4 (1) (2,0,0) f_2 (3) (3,0,1) d_3 (3) (2,1,0) f_3 (3) (3,2,0) f_4 (1) (2,0,1)

Quelles sont les valeurs maximales de N_1, N_2, N_3 de telle façon à ce que si le mot était valide, il ne le soit plus ? Quelles sont les valeurs minimales de N_1, N_2, N_3 de telle façon à ce que si le mot n'était pas valide, il le soit ?

Chapitre 7 GESTION DE LA MEMOIRE

Pour pouvoir utiliser un ordinateur en multiprogrammation, le SE charge plusieurs processus en mémoire centrale (MC). La façon la plus simple consiste à affecter à chaque processus un ensemble d'adresses contiguës.

Quand le nombre de tâches devient élevé, pour satisfaire au principe d'équitabilité et pour minimiser le temps de réponse des processus, il faut pouvoir **simuler** la présence simultanée en MC de tous les processus. D'où la technique de "va et vient" ou **recouvrement (swapping)**, qui consiste à stocker temporairement sur disque l'image d'un processus, afin de libérer de la place en MC pour d'autres processus.

D'autre part, la taille d'un processus doit pouvoir dépasser la taille de la mémoire disponible, même si l'on enlève tous les autres processus. L'utilisation de pages (mécanisme de **pagination**) ou de segments (mécanisme de **segmentation**) permet au système de conserver en MC les parties utilisées des processus et de stocker, si nécessaire, le reste sur disque.

Le rôle du gestionnaire de la mémoire est de connaître les parties libres et occupées, d'allouer de la mémoire aux processus qui en ont besoin, de récupérer de la mémoire à la fin de l'exécution d'un processus et de traiter le recouvrement entre le disque et la mémoire centrale, lorsqu'elle ne peut pas contenir tous les processus actifs.

1.GESTION SANS RECOUVREMENT, NI PAGINATION

1.1 La monoprogrammation

Il n'y a en MC que :

- un seul processus utilisateur,
- le processus système (pour partie en RAM, pour partie en ROM; la partie en ROM étant appelée BIOS [*Basic Input Output System*])
- les pilotes de périphériques

Cette technique en voie de disparition est limitée à quelques micro-ordinateurs . Elle n'autorise qu'un seul processus actif en mémoire à un instant donné.

1.2 La multiprogrammation

La multiprogrammation est utilisée sur la plupart des ordinateurs : elle permet de diviser un programme en plusieurs processus et à plusieurs utilisateurs de travailler en temps partagé avec la même machine.

Supposons qu'il y ait n processus indépendants en MC, chacun ayant la probabilité p d'attendre la fin d'une opération d'E/S. La probabilité que le processeur fonctionne est $1 - p^n$.(Il s'agit d'une estimation grossière, puisque sur une machine monoprocesseur, les processus ne sont pas indépendants (attente de libération de la ressource processeur pour démarrer l'exécution d'un processus prêt).

Toutefois, on remarque que plus le nombre n de processus en MC est élevé, plus le taux d'utilisation du processeur est élevé : on a donc intérêt à augmenter la taille de la MC.

Une solution simple consiste à diviser la mémoire en **n partitions fixes**, de tailles pas nécessairement égales (méthode MFT [*Multiprogramming with a Fixed number of Tasks*] apparue avec les IBM 360). Il existe deux méthodes de gestion :

- on crée une file d'attente par partition . Chaque nouveau processus est placé dans la file d'attente de la plus petite partition pouvant le contenir. Inconvénients :
  * on perd en général de la place au sein de chaque partition
  * il peut y avoir des partitions inutilisées (leur file d'attente est vide)

- on crée une seule file d'attente globale. Il existe deux stratégies :
  * dès qu'une partition se libère, on lui affecte la première tâche de la file qui peut y tenir. Inconvénient : on peut ainsi affecter une partition de grande taille à une petite tâche et perdre beaucoup de place

  * dès qu'une partition se libère, on lui affecte la plus grande tâche de la file qui peut y tenir. Inconvénient : on pénalise les processus de petite taille.

1.3 Code translatable et protection

Avec le mécanisme de multiprogrammation, un processus peut être chargé n'importe où en MC. Il n'y a plus concordance entre l'adresse dans le processus et l'adresse physique d'implantation. Le problème de l'adressage dans un processus se résout par l'utilisation d'un registre particulier, le **registre de base** : au lancement du processus, on lui affecte l'adresse de début de la partition qui lui est attribuée. On a alors :

adresse physique = adresse de base (contenu de ce registre)
+ adresse relative (mentionnée dans le processus)

De plus, il faut empêcher un processus d'écrire dans la mémoire d'un autre. Deux solutions :

* des **clés de protection** : une clé de protection pour chaque partition, dupliquée dans le mot d'état (PSW = Program Status Word) du processus actif implanté dans cette partition. Si un processus tente d'écrire dans une partition dont la clé ne concorde pas avec celle de son mot d'état, il y a refus.

* un **registre limite**, chargé au lancement du processus à la taille de la partition. Toute adresse mentionnée dans le processus (relative) supérieure au contenu du registre limite entraîne un refus.

2. GESTION AVEC RECOUVREMENT, SANS PAGINATION

Dès que le nombre de processus devient supérieur au nombre de partitions, il faut pouvoir simuler la présence en MC de tous les processus pour pouvoir satisfaire au principe d'équité et minimiser le temps de réponse des processus. La technique du recouvrement (swapping) permet de stocker temporairement sur disque des images de processus afin de libérer de la MC pour d'autres processus.

On pourrait utiliser des partitions fixes, mais on utilise en pratique des **partitions de taille variable**, car le nombre, la taille et la position des processus peuvent varier dynamiquement au cours du temps. On n'est plus limité par des partitions trop grandes ou trop petites comme avec les partitions fixes. Cette amélioration de l'usage de la MC nécessite un mécanisme plus complexe d'allocation et de libération.

2.1 Opérations sur la mémoire

Le **compactage** de la mémoire permet de regrouper les espaces inutilisés. Très coûteuse en temps UC, cette opération est effectuée le moins souvent possible.

S'il y a une requête d'**allocation dynamique de mémoire** pour un processus, on lui alloue de la place dans le tas (heap) si le SE le permet, ou bien de la mémoire supplémentaire contiguë à la partition du processus (agrandissement de celle-ci). Quand il n'y a plus de place, on déplace un ou plusieurs processus :

- soit pour récupérer par ce moyen des espaces inutilisés,
- soit en allant jusqu'au recouvrement. A chaque retour de recouvrement (swap), on réserve au processus une partition un peu plus grande que nécessaire, utilisable pour l'extension de la partition du processus venant d'être chargé ou du processus voisin.

Il existe trois façons de mémoriser l'occupation de la mémoire : les tables de bits (*bits maps*), les listes chaînées et les subdivisions (*buddy*).

2.2 Gestion de la mémoire par table de bits

On divise la MC en **unités d'allocations** de quelques octets à quelques Ko. A chaque unité, correspond un bit de la table de bits : valeur 0 si l'unité est libre, 1 sinon. Cette table est stockée en MC. Plus la taille moyenne des unités est faible, plus la table occupe de place.

A un retour de recouvrement (swap), le gestionnaire doit rechercher suffisamment de 0 consécutifs dans la table pour que la taille cumulée de ces unités permette de loger le nouveau processus à implanter en MC, avec le choix entre trois critères possibles :

- la **première zône libre** (*first fit*) : algorithme rapide
- le **meilleur ajustement** (*best fit*), mais on peut créer ainsi de nombreuses petites unités résiduelles inutilisables (fragmentation nécessitant un compactage ultérieur)
- le **plus grand résidu** (*worst fit*), avec le risque de fractionnement regrettable des grandes unités

2.3 Gestion de la mémoire par liste chaînée

On utilise une liste chaînée des zones libres en MC. On applique :

- soit l'un des algorithmes précédents,

- soit un algorithme de **placement rapide** (*quick fit*) : on crée des listes séparées pour chacune des tailles les plus courantes, et la recherche est considérablement accélérée.

A l'achèvement d'un processus ou à son transfert sur disque, il faut du temps (mise à jour des liste chaînées) pour examiner si un regroupement avec ses voisins est possible pour éviter une fragmentation excessive de la mémoire.

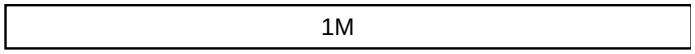
En résumé, les listes chaînées sont une solution plus rapide que la précédente pour l'allocation, mais plus lente pour la libération.

2.4 Gestion de la mémoire par subdivisions (ou frères siamois)

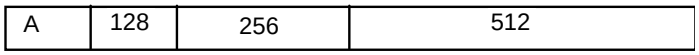
Cet algorithme proposé par Donald KNUTH en 1973 utilise l'existence d'adresses binaires pour accélérer la fusion des zones libres adjacentes lors de la libération d'unités.

Le gestionnaire mémorise une liste de blocs libres dont la taille est une puissance de 2 (1, 2, 4, 8 octets, ..., jusqu'à la taille maximale de la mémoire).

Par exemple, avec une mémoire de 1 Mo, on a ainsi 251listes. Initialement, la mémoire est vide. toutes les listes sont vides, sauf la liste 1 Mo qui pointe sur la zone libre de 1 Mo :

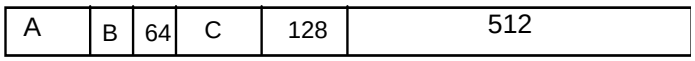


Un processus A demande 70 Ko : la mémoire est fragmentée en deux compagnons (buddies) de 512 Ko; l'un d'eux est fragmenté en deux blocs de 256 Ko; l'un d'eux est fragmenté en deux blocs de 128 Ko et on loge A dans l'un d'eux, puisque $64 < 70 < 128$:

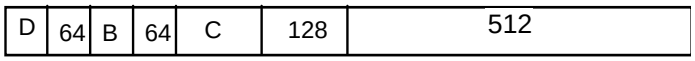


Un processus B demande 35 Ko : l'un des deux blocs de 128 Ko est fragmenté en deux de 64 Ko et on loge B dans l'un d'eux puisque $32 < 35 < 64$:

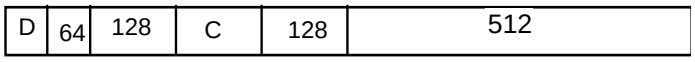
Un processus C demande 80 Ko : le bloc de 256 Ko est fragmenté en deux de 128 Ko et on loge C dans l'un d'eux puisque $64 < 80 < 128$:



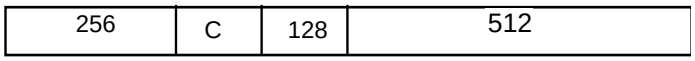
A s'achève et libère son bloc de 128 Ko. Puis un processus D demande 60 Ko : le bloc libéré par A est fragmenté en deux de 64 Ko, dont l'un logera D :



B s'achève, permettant la reconstitution d'un bloc de 128 Ko :



D s'achève, permettant la reconstitution d'un bloc de 256 Ko , etc...



L'allocation et la libération des blocs sont très simples. Mais un processus de taille $2^n + 1$ octets utilisera un bloc de 2^{n+1} octets ! Il y a beaucoup de perte de place en mémoire.

Dans certains systèmes, on n'alloue pas une place fixe sur disque aux processus qui sont en mémoire. On les loge dans un espace de va et vient (*swap area*) du disque. Les algorithmes précédents sont utilisés pour l'affectation.

3. GESTION AVEC RECOUVREMENT AVEC PAGINATION OU SEGMENTATION

La taille d'un processus doit pouvoir dépasser la taille de la mémoire physique disponible, même si l'on enlève tous les autres processus. En 1961, J. FOTHERINGHAM proposa le principe de la **mémoire virtuelle** : le SE conserve en mémoire centrale les parties utilisées des processus et stocke, si nécessaire, le reste sur disque. Mémoire virtuelle et multiprogrammation se complètent bien : un processus en attente d'une ressource n'est plus conservé en MC, si cela s'avère nécessaire.

La mémoire virtuelle fait appel à deux mécanismes : segmentation ou pagination. La mémoire est divisée en segments ou pages. Sans recours à la mémoire virtuelle, un processus est entièrement chargé à des adresses contiguës ; avec le recours à la mémoire virtuelle, un processus peut être chargé dans des pages ou des segments non contigus.

3.1 La pagination

L'espace d'adressage d'un processus est divisé en petites unités de taille fixe appelées **pages**. La MC est elle aussi découpée en unités physiques de même taille appelées **cadres**. Les échanges entre MC et disques ne portent que sur des pages entières. De ce fait, l'espace d'adressage d'un processus est potentiellement illimité (limité à l'espace mémoire total de la machine). On parle alors d'**adressage virtuel**.

Pour un processus, le système ne chargera que les pages utilisées. Mais la demande de pages à charger peut être plus élevée que le nombre de cadres disponibles. Une gestion de l'allocation des cadres libres est nécessaire.

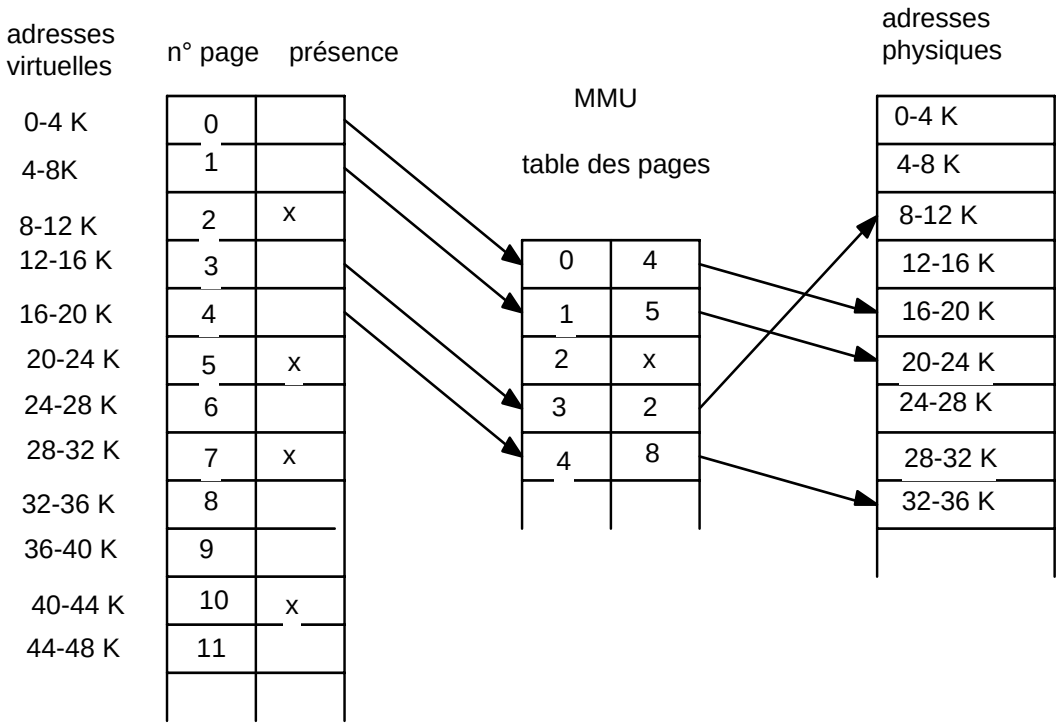
Dans un SE sans mémoire virtuelle, la machine calcule les adresses physiques en ajoutant le contenu d'un registre de base aux adresses relatives contenues dans les instructions du processus. Dans un SE à pagination, un sous-ensemble inséré entre l'UC et la MC, **la MMU** (Memory Management Unit ou unité de gestion de la mémoire) traduit les adresses virtuelles en adresses physiques.

La MMU mémorise :

- les cadres physiques alloués à des processus (sous forme d'une table de bits de présence)

- les cadres mémoire alloués à chaque page d'un processus (sous forme d'une table des pages)

On dira qu'une page est **mappée** ou **chargée** si elle est physiquement présente en mémoire.



Dans l'exemple précédent, les pages ont une taille de 4 Ko. L'adresse virtuelle 12292 correspond à un déplacement de 4 octets dans la page virtuelle 3 (car 12292 = 12288 + 4 et 12288 = 12*1024). La page virtuelle 3 correspond à la page physique 2. L'adresse physique correspond donc à un déplacement de 4 octets dans la page physique 2, soit : (8*1024) + 4 = 8196 .

Par contre, la page virtuelle 2 n'est pas mappée. Une adresse virtuelle comprise entre 8192 et 12287 donnera lieu à **un défaut de page**. Il y a défaut de page quand il y a un accès à une adresse virtuelle correspondant à une page non mappée. En cas de défaut de page, un déroutement se produit (trap) et le processeur est rendu au SE. Le système doit alors effectuer les opérations suivantes :

- déterminer la page à charger
- déterminer la page à décharger sur le disque pour libérer un cadre
- lire sur le disque la page à charger
- modifier la table de bits et la table de pages

3.2 La segmentation

Dans cette solution, l'espace d'adressage d'un processus est divisé en **segments**, générés à la compilation. Chaque segment est repéré par son numéro S et sa longueur variable L. Un segment est un ensemble d'adresses virtuelles contiguës.

Contrairement à la pagination, la segmentation est "connue" du processus : une adresse n'est plus donnée de façon absolue par rapport au début de l'adressage virtuel; une adresse est donnée par un couple (S , d), où S est le n° du segment et d le déplacement dans le segment, $d \in [0 , L [$.

Pour calculer l'adresse physique, on utilise une **table des segments** :

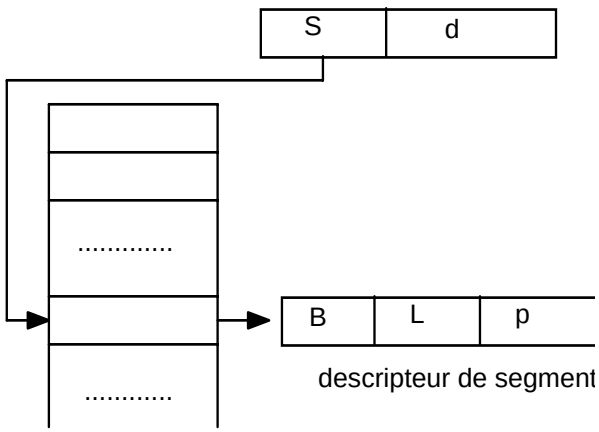


table des segments

B : adresse de base (adresse physique de début du segment)
L : longueur du segment ou limite
p : protection du segment

L'adresse physique correspondant à l'adresse virtuelle (S , d) sera donc B + d, si d <= L

La segmentation simplifie la gestion des objets communs (rangés chacun dans un segment), notamment si leur taille évolue dynamiquement.

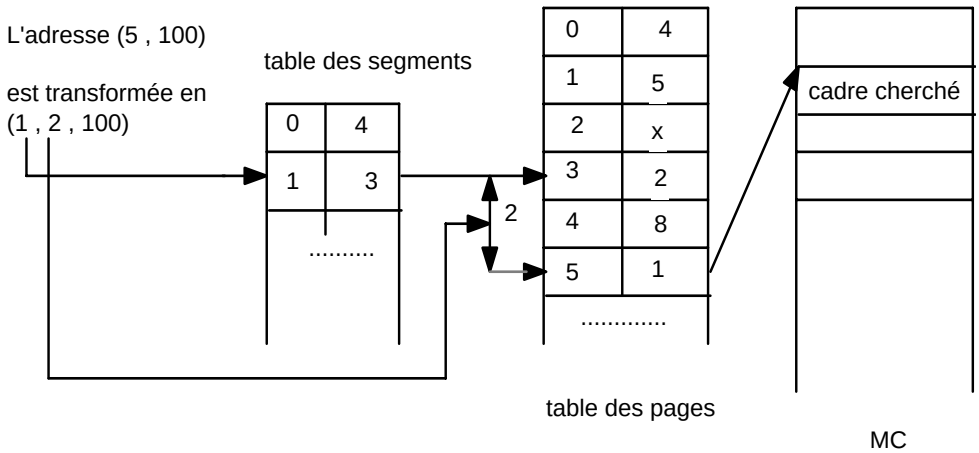
3.3 La pagination segmentée

Lorsque le système possède beaucoup de pages, la consultation de la table des pages peut être lente et inefficace s'il y a beaucoup de pages non chargées. On la segmente alors par processus.

Chaque processus possède une table des segments de la table des pages qui le concernent.

Une adresse (P , r) , avec P n° logique de page et r déplacement dans la page, est transformée à la compilation en adresse (S , P' , r), avec S n° d'un segment et P' .

Exemple :

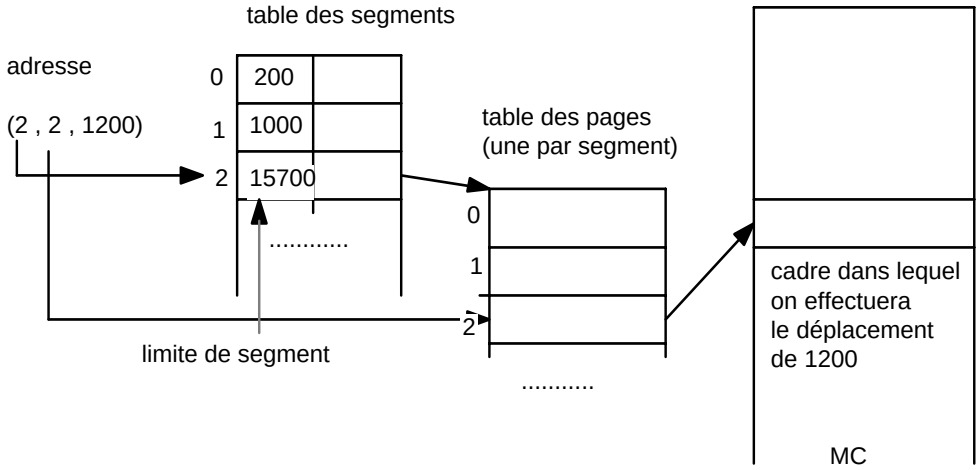


3.4 La segmentation paginée

La taille d'un segment peut être importante, d'où un temps de chargement long qui peut en résulter. La pagination des segments peut être une solution.

Une adresse virtuelle (S , d), avec S n° de segment et d déplacement dans le segment, est transformée en (S , P , d'), où P est un n° de page et d' un déplacement dans la page P.

Exemple : avec l'hypothèse de pages de 4 Ko, l'adresse logique (2 , 9200) est transformée en (2 , 2, 1200)



Dans tous les cas, l'utilisation d'une mémoire associative, stockant des triplets (S, P, adresse de cadre), gérée comme fenêtre au voisinage du dernier cadre accédé, peut accélérer la recherche des cadres.

4. ALGORITHMES DE REMPLACEMENT DE PAGES

Les méthodes reposent sur 3 stratégies :

- une stratégie de **chargement** qui choisit les pages à charger et l'instant de chargement
- une stratégie de **placement** qui choisit un cadre libre pour chaque page à charger
- une stratégie de r**emplacement** destinée à libérer certains cadres. La recherche se fait soit sur l'ensemble des pages (recherche globale), soit sur les pages "appartenant" au processus (recherche locale). Les stratégies de placement sont dépendantes des stratégies de remplacement : on charge nécessairement une page dans le cadre qui vient d'être libéré.

On peut distinguer deux catégories de méthodes :

- **pagination anticipée** : on charge les pages à l'avance; mais comment prévoir efficacement ?
- **pagination à la demande** : le chargement n'a lieu qu'en cas de défaut de page et on ne charge que la page manquante.

On appelle **suite de références** $w = p_1\ p_2\ ...\ p_n$ une suite de numéros de pages correspondant à des accès à ces pages. On dira qu'un algorithme A de pagination sur m cadres est **stable** si :

$$\forall\ m\ ,\ \forall\ w\ ,\ \text{on a : Coût}\ (A\ ,\ m\ ,\ w) <= \text{Coût}\ (\ A\ ,\ m+1\ ,\ w)$$

Cette notion est importante pour éviter l'écroulement du SE (thrashing) par une génération excessive de défauts de pages.

4.1 Remplacement de page optimal

Le SE indexe chaque page par le nombre d'instructions qui seront exécutées avant qu'elle ne soit référencée. En cas de nécessité, le SE retire la page d'indice le plus élevé, c'est à dire la page qui sera référencée dans le futur le plus lointain.

Cet algorithme est pratiquement impossible à appliquer (comment calculer les indices des pages ?). Toutefois, avec un simulateur, on peut évaluer les performances de cet algorithme et s'en servir comme référence pour les suivants. En outre, cet algorithme est stable.

4.2 NRU (not recently used)

Le SE référence chaque page par deux bits R (le plus à gauche) et M initialisés à 0. A chaque accès en lecture à une page, R est mis à 1. A chaque accès en écriture, M est mis à 1. A chaque interruption d'horloge, le SE remet R à 0.

Les bits R-M forment le code d'un index de valeur 0 à 3 en base 10. En cas de défaut de page, on retire une page au hasard dans la catégorie non vide de plus petit index. On retire donc préférentiellement une page modifiée non référencée (index 1) qu'une page très utilisée en consultation (index 2). Cet algorithme est assez efficace.

4.3 FIFO (first in, first out)

Le SE indexe chaque page par sa date de chargement et constitue une liste chaînée, la première page de la liste étant la plus ancienne ment chargée et la dernière la plus récemment chargée. Le SE replacera en cas de nécessité la page en tête de la liste et chargera la nouvelle page en fin de liste.

Deux critiques à cet algorithme :

- ce n'est pas parce qu'une page est la plus ancienne en mémoire qu'elle est celle dont on se sert le moins
- l'algorithme n'est pas stable : quand le nombre de cadres augmente, le nombre de défauts de pages ne diminue pas nécessairement (on parle de l'anomalie de BELADY : L.A. BELADY a proposé en 1969 un exemple à 4 cadres montrant qu'on avait plus de défaut de pages qu'avec 3).

Amélioration 1 : le SE examine les bits R et M (gérés comme ci-dessus en 4.2) de la page la plus ancienne en MC (en tête de la liste). Si cette page est de catégorie 0, il l'ôte. Sinon, il teste la page un peu moins ancienne, etc... S'il n'y a aucune page de catégorie 0, il recommence avec la catégorie 1, puis éventuellement avec les catégories 2 et 3.

Amélioration 2 : **Algorithme de la seconde chance** : R et M sont gérés comme ci-dessus en 4.2. Le SE examine le bit R de la page la plus ancienne (tête de la liste). Si R = 0, la page est remplacée, sinon R est mis à 0, la page est placée en fin de liste et on recommence avec la nouvelle page en tête. Si toutes les pages ont été référencées depuis la dernière RAZ, on revient à FIFO, mais avec un surcoût.

4.4 LRU (least recently used)

En cas de nécessité, le SE retire la page la moins récemment référencée. Pour cela, il indexe chaque page par le temps écoulé depuis sa dernière référence et il constitue une liste chaînée des pages par ordre décroissant de temps depuis la dernière référence.

L'algorithme est stable. Mais il nécessite une gestion coûteuse de la liste qui est modifiée à chaque accès à une page.

Variantes :

NFU (not frequently used) ou **LFU** (least frequently used) : le SE gère pour chaque page un compteur de nombre de références . Il cumule dans ce compteur le bit R juste avant chaque RAZ de R au top d'horloge. Il remplacera la page qui aura la plus petite valeur de compteur.

Vieillisement (aging) ou **NFU modifié** : avant chaque RAZ de R, le SE décale le compteur de 1 bit à droite (division entière par 2) et additionne R au bit de poids **fort**. En cas de nécessité, on ôtera la page de compteur le plus petit. Mais en cas d'égalité des compteurs, on ne sait pas quelle est la page la moins récemment utilisée.

Matrice : s'il y a N pages, le SE gère une matrice (N , N+1). A chaque référence à la page p, le SE positionne à 1 tous les bits de la ligne p et à 0 tous les bits de la colonne p. En outre, le 1er bit de la ligne p vaut 0 si la page est chargée, 1 sinon. On ôtera la page dont la ligne a la plus petite valeur.

4.5 Améliorations

Le dérobeur de pages : dès qu'un seuil minimal de cadres libres est atteint, le SE réveille un dérobeur de pages . Celui-ci supprime les pages les moins récemment utilisées, par algorithme d'aging, et les range dans la liste des cadres libres, ceci jusqu'à un seuil donné. Le SE vérifie qu'une page référencée sur défaut de page n'est pas dans la liste.

Problèmes d'entrée/sortie : on a intérêt, soit à verrouiller les pages concernées par les E/S pour éviter de les ôter, soit à effectuer les E/S dans des tampons du noyau et à copier les données plus tard dans les pages des utilisateurs.

Pages partagées : lorsque plusieurs processus exécutent le même ensemble de code, on a intérêt à utiliser des pages de code partagées, plutôt que des pages dupliquées. Mais on aura à prévoir une gestion spécifique des pages partagées pour éviter des défauts de pages artificiels.

4.6 Taille optimale des pages

Soit t la taille moyenne des processus et p (en octets) la taille d'une page. Soit e (en octets) la taille de l'entrée de chaque page dans la table des processus. Le nombre moyen de pages par processus est t/p. Ces pages occupent t*e/p octets dans la table des processus. La mémoire perdue en moyenne dans la dernière page d'un processus est p/2 (fragmentation interne). La perte totale q est donc :

$$q = t*e/p + p/2$$

q est minimal si : dq/dp = 0, c'est à dire : $p = (2\ t*e)^{0.5}$

Exemple : t = 64 Ko, e = 8 octets, donc p = 1 Ko
Les valeurs courantes sont 1/2 Ko, 1 Ko, 2 Ko ou 4 Ko

4.7 Le modèle du Working Set

Le modèle working-set de comportement d'un programme est une tentative proposée par P.J. DENNING en 1968 pour comprendre les performances d'un système paginé en multi-programmation. Il s'agit de raisonner non pas en terme de processus isolés, mais d'effet des autres processus présents sur chacun d'eux, de corréler allocation de mémoire et allocation du processeur.

Dans le cas d'un système mono-processeur, lorsque le **degré de multi-programmation** augmente (*nombre de processus présents en mémoire*), l'utilisation du processeur augmente : en effet, l'ordonnanceur a toutes les chances de trouver à chaque instant un processus à exploiter.

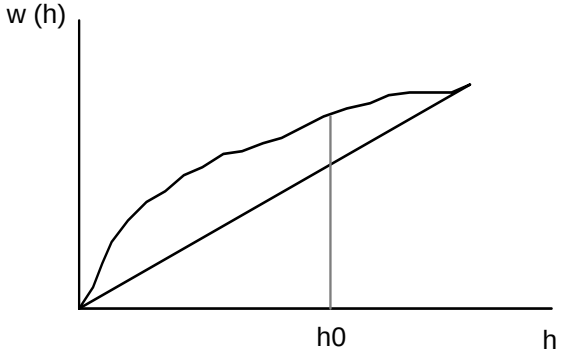
Toutefois, si le degré de multi-programmation franchit un certain seuil, il en résulte une forte croissance du trafic de pagination entre disques et mémoire centrale, accompagnée d'une **diminution brusque de l'utilisation du processeur**. Le haut degré de multi-programmation empêche chaque processus de garder suffisamment de pages en mémoire pour éviter un grand nombre de défauts de pages. Le canal de liaison avec le disque arrive à saturation et beaucoup de processus sont bloqués par attente de transfert de pages. On parle de ***thrashing*** (déconfiture) du processeur.

En d'autres termes, chaque processus exige un nombre minimal de pages présentes en mémoire centrale (appelé son working-set) pour pouvoir réellement utiliser le processeur. Le degré de multi-programmation doit être tel que le working-set de tous les processus présents en mémoire doit être respecté.

Conceptuellement, on peut définir le working-set à un instant t :

w (t , h) = { ensemble des pages apparaissant dans les h dernières références }

DENNING a montré que w(h) avait un graphe tel que celui-ci :



Plus h croît, moins on trouve de pages en excédent dans le working-set. Cela permet d'établir une valeur raisonnable de h (h_0) tel qu'une valeur de h supérieure à h_0 n'accroîtrait pas beaucoup le working-set.

5. PARTAGE DE CODE ET DONNEES EN MEMOIRE CENTRALE

Des procédures peuvent être partagées entre plusieurs processus. On a intérêt à ce qu'il n'y ait qu'une copie du code qui sera exécutée par chaque processus avec ses propres données.

Par exemple deux processus P1 et P2 sont susceptibles d'utiliser une procédure R. On ne peut pas prévoir l'ordre dans lequel ils vont entrer dans R ou sortir de R. Si l'un des deux processus est désalloué pendant qu'il exécute R et qu'il voit ses variables locales modifiées par l'exécution de R par l'autre processus, alors on dit que R n'est pas **ré-entrante**.

5.1 Cas de la pagination

Le partage du code et des données se fait par un double référencement aux pages contenant les informations communes.

5.1.1 Partage du code

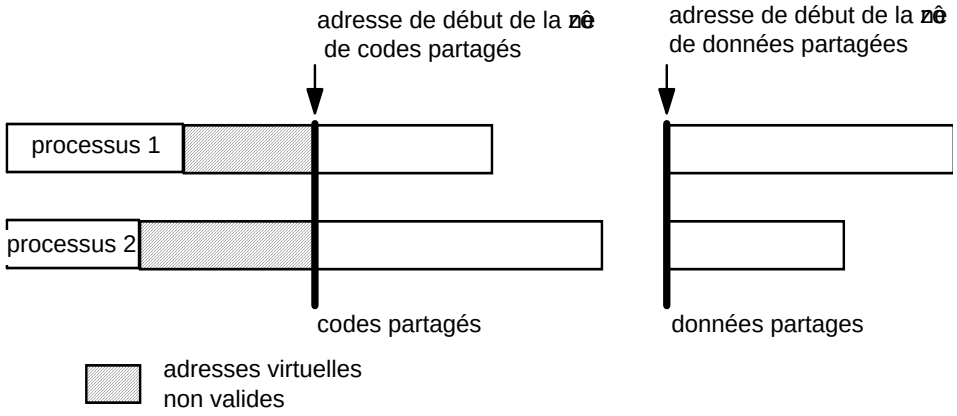
Chaque fois qu'un processus fait référence à son propre code, il utilise un numéro de page fixe qui le fait accéder toujours à la même page virtuelle. Donc, si un processus partage son code, il va toujours référencer la même page virtuelle, quelle que soit la table des pages utilisée. Chaque tâche doit donc mettre le code exécutable de ce processus à la même adresse virtuelle.

5.1.2 Partage de données

Pour partager des données, la zone commune peut être à des endroits différents dans la table des pages de chacune des tâches. Soit une page commune P contenant un pointeur q :

- si D pointe sur une adresse dans P, le numéro de cette page doit être le même dans tous les processus
- si l'objet pointé n'est pas dans P, alors il doit avoir la même adresse virtuelle dans tous les processus

5.1.3 Solution habituelle



5.2 Cas de la segmentation

Pour que deux processus référencent une partie commune, il suffit que cette partie se trouve dans un segment qui possède le même descripteur dans la table des segments de chacun des processus.

6. EXEMPLE : GESTION DE LA MEMOIRE PAR WINDOWS NT

Avec les adresses exprimées sur 32 bits, WINDOWS NT offre à chaque processus 2^{32} octets = 4 Go de mémoire virtuelle : 2 Go pour le SE (threads en mode noyau) et 2 Go pour les programmes (*threads* en mode utilisateur et en mode noyau) paginés en pages de 4 Ko.

WINDOWS NT protège sa mémoire grâce aux mécanismes suivants :

- espace d'adressage virtuel différent pour chaque processus

- fonctionnement en mode noyau pour l'accès au code et aux données du système

6.1 Le gestionnaire de la mémoire virtuelle

Il optimise l'utilisation de la mémoire selon plusieurs techniques :

- **évaluation différée** : plusieurs processus peuvent accéder aux mêmes données en lecture, sans les dupliquer. Si l'un des processus veut modifier des données, la page physique est recopiée ailleurs dans la mémoire; l'espace d'adressage virtuel du processus est mis à jour

- **protection de mémoire par page** : chaque page contient le mode d'accès autorisé en mode noyau et en mode utilisateur (lecture, lecture/écriture, exécution, pas d'accès, copie à l'écriture, page de garde)

Le gestionnaire de la mémoire virtuelle fournit des services natifs aux processus :

- lecture/écriture en mémoire virtuelle

- blocage des pages virtuelles en mémoire physique

- protection des pages virtuelles

- déplacement des pages virtuelles sur disque

- allocation de la mémoire en deux phases : réservation (ensemble d'adresses réservées pour un processus) et engagement (mémoire retenue dans le fichier d'échange avec le disque du gestionnaire de la mémoire virtuelle)

Le gestionnaire de pages utilise des techniques classiques :

- **pagination à la demande** : à la suite d'un défaut de pages, on charge les pages requises *plus quelques pages voisines*

- **placement des pages virtuelles en mémoire** : à la première page physique disponible

- **remplacement** selon la stratégie FIFO d'une page virtuelle d'un processus, suite à un défaut de page

- **table des pages à deux niveaux**

En outre, lorsqu'une page est requise suite à un défaut de page, et qu'elle est non utilisée, un *indicateur de transition* permet de déceler si elle peut devenir utilisable sans la faire transiter par le disque.

Toute page physique peut prendre l'un des **6** états suivants utilisés dans la base de données des pages physiques :

- **valide** : page utilisée par un processus
- **à zéro** : libre et initialisée avec des 0 pour des raisons de sécurité ("niveau C2")
- **modifiée** : son contenu a été modifié et n'a pas encore été recopié sur disque
- **mauvais** : elle a généré des erreurs au contrôle
- **en attente** : retirée d'un ensemble de pages de travail

Le gestionnaire des pages étant réentrant, il utilise un verrou unique pour protéger la base de données des pages physiques et n'en autoriser l'accès qu'à une tâche (thread) simultanée.

6.2 La mémoire partagée

Définition : c'est de la mémoire visible par plusieurs processus ou présente dans plusieurs espaces d'adressage virtuels

WINDOWS NT utilise des objets-sections pour utiliser des vues de fenêtres de la zone de mémoire partagée à partir de différents processus. La mémoire partagée est généralement paginée sur disque.

EXERCICES

1. On considère la table de segments suivante :

segment	base	longueur
0	540	234
1	1254	128
2	54	328
3	2048	1024
4	976	200

Représenter la mémoire. Calculer les adresses physiques correspondant à :
(0 , 128) , (1, 99) , (4 , 100) , (3 , 888) , (2 , 465) , (4 , 344)

2. Dans un système à pagination de 100 octets, donner la suite des pages correspondant aux adresses suivantes : 34 , 145 , 10 , 236, 510, 412, 789
Proposer une formule de calcul.

3. Pourquoi combine-t-on parfois pagination et segmentation ? Décrire un mécanisme d'adressage où la table des segments et les segments sont paginés.

Soit un processus dont la table des segments est :

segment	base	longueur
0	540	234
1	1254	128
2	54	328

La taille de page est 100 octets et les cadres sont alloués dans l'ordre suivant :
3, 4, 9, 0, 8, 7, 6, 11, 15, 5, 17

Représenter graphiquement la nouvelle table des segments et la mémoire (le processus étant entièrement chargé).
Traduire les adresses suivantes : (0 , 132) , (2 , 23) , (2 , 301)

4. Soit une machine à 3 pages dans laquelle le temps de traitement d'un défaut de page est 1 ms s'il n'y a pas de recopie, 3 ms s'il y a recopie. Le processus en cours effectue 5 ms de calcul entre deux références de page. Les bits R et M sont remis à 0 toutes les 25 ms de calcul, mais le traitement d'un défaut de page suspend l'horloge.
L'astérisque désignant un accès en écriture, w = 0 2* 4 1* 2 3* 0 4* 2* 4 3* 4* 5 3* 2, calculer le temps total d'exécution du processus avec les algorithmes :
   - FIFO seconde chance,
   - NFU avec un compteur sur 4 bits

Comparer avec le remplacement optimal.

5. Soit le programme C :

```
int A [1000], B [1000], C [1000], i;
main ()
{
      for i = 0; i < 1000; i++) C [i] = A [i] + B [i] ;
```

}

La MC est de 2 K mots et un entier occupe un mot. Le code du programme et la variable i occupent 256 mots.
Calculer le nombre et le taux (nombre de défauts de pages/nombre de référencements) des défauts de pages lorsque :

- une page = 128 mots, taille de la zone de données = 3 pages
- une page = 128 mots, taille de la zone de données = 6 pages
- une page = 256 mots, taille de la zone de données = 6 pages

pour les algorithmes FIFO, LRU et remplacement optimal.

Chapitre 8 SYSTEME DE FICHIERS

1. CONCEPTION D'UN SYSTEME DE FICHIERS

1.1 Fichiers

Un fichier est vu comme une suite d'**articles** ou d'**enregistrements logiques** d'un type donné qui ne peuvent être manipulés qu'au travers d'opérations spécifiques. L'utilisateur n'est pas concerné par l'implantation des enregistrements, ni par la façon dont sont réalisées les opérations. Généralement, elles sont traduites par le SE en opérations élémentaires sur les mémoires.

On dispose généralement des opérations suivantes :

fichiers : créer, ouvrir, fermer, détruire, pointer au début, renommer, copier dans un autre fichier, éditer le contenu

articles : lire, écrire, modifier, insérer, détruire, retrouver

Définition : un système de fichiers (SGF) est l'entité regroupant les fichiers mémorisés sur disque. Il contient les données des fichiers et un ensemble d'informations techniques.

Exemple : le SGF d'UNIX. Ses principales caractéristiques sont les suivantes :

- aucune structure interne : les fichiers sont de simples suites d'octets dont chacun peut être adressé individuellement. L'enregistrement logique a la taille d'un octet

- l'expansion des fichiers est dynamique

- les répertoires forment une structure hiérarchique

- l'utilisation est commune pour les fichiers de données et les fichiers spéciaux

Le **superbloc** d'UNIX contient notamment les informations suivantes :

- taille du système de fichier
- nombre de blocs libres
- pointeur sur le premier bloc libre dans la liste des blocs libres
- taille de la liste des i-nodes
- nombre et liste des i-nodes libres

UNIX utilise une technique de zones tampons en MC afin d'optimiser l'utilisation du disque. Une partie du superbloc et de la table des i-nodes est chargée en MC lors de l'exécution de la commande ***mount***. Toute opération d'E/S se fera à travers ces tables afin d'optimiser l'utilisation du disque. Lorsqu'un programme met à jour des informations d'un fichier, cela se fait à travers les zones tampons de la MC. Régulièrement, ces zones sont recopiées sur disque. Les zones concernées du superbloc sont mises à jour. cette situation est critique car la version du superbloc en MC et celle sur disque peuvent être différentes jusqu'à recopie sur disque de la partie du superbloc qui est en MC.

1.2 Organisation de l'espace disque

Il existe deux stratégies pour stocker un fichier de n octets sur disque :

- **allouer des secteurs contigus** totalisant une capacité d'au-moins n octets. Avec deux inconvénients :
  - le dernier secteur a toutes chances d'être sous-utilisé et ainsi, on gaspille de la place. Le pourcentage de place perdue est d'autant plus grand que la taille moyenne des fichiers est faible, ce qui est la réalité
  - si le fichier est agrandi, il faudra déplacer le fichier pour trouver un nouvel ensemble de secteurs consécutifs de taille suffisante

- **diviser le fichier en blocs** de taille fixe, insécables, que le SE alloue de façon non nécessairement contiguë à des secteurs.

Un **bloc** est défini comme une zone de mémoire secondaire contenant la quantité d'information qui peut être lue ou écrite par un périphérique en une seule opération. La taille d'un bloc est donc attachée à un périphérique d'E/S et fixée par le matériel. Cependant, si le SE n'a pas le choix de la taille d'un bloc, il peut grouper plusieurs article dans un même bloc (packing).

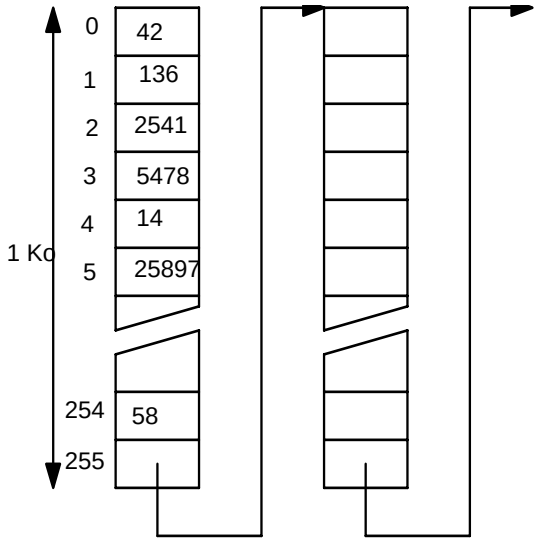
Pour des blocs de taille < 1 Ko, le taux de remplissage du disque est excellent, mais la vitesse de transfert des données est modeste (délai de rotation du disque et de recherche importants par rapport au temps de transfert lui-même). Au-delà de 1 Ko, le taux de remplissage se dégrade, mais la vitesse de transfert s'améliore.

1.3 Gestion des blocs libres

Dès qu'on a choisi une taille de blocs (souvent 1/2 Ko à 2 Ko), on doit trouver un moyen de mémoriser les blocs libres. Les deux techniques les plus répandues sont les suivantes :

- **table de bits :** on gère une table comportant autant de bits que de blocs sur le disque. A chaque bloc du disque, correspond un bit dans la table, positionné à 1 si le bloc est occupé, à 0 si le bloc est libre (ou vice versa). Par exemple, un disque de 300 Mo, organisé en blocs de 1 Ko, sera géré par une table de 300 Kbits qui occupera 38 des 307.200 blocs

- **liste chaînée de n° des blocs :** par exemple, un disque de 300 Mo, organisé en blocs de 1 Ko : supposons que chaque bloc soit adressé par 4 octets. Chaque bloc de la liste pourra contenir 255 adresses de blocs libres. La liste comprendra donc au plus 307.200/255 = 1205 blocs. Cette solution mobilise beaucoup plus de place que la précédente.



On peut imaginer deux variantes de cette dernière solution :

- **une liste chaînée de blocs :** chaque bloc libre pointe sur le bloc libre suivant

- **une liste chaînée de zônes libres :** chaque bloc de début d'une série de blocs libres (zone libre) contient la taille de la zone et pointe sur le premier bloc de la zone libre suivante.

1.4 Allocation de blocs pour le stockage des fichiers

Quatre méthodes sont utilisées selon les SE :

- **allocation contiguë :** on retrouve les solutions utilisées pour l'allocation de mémoire (algorithmes de la première zone libre, algorithme du meilleur ajustement, algorithme du meilleur résidu). L'implantation physique est proche de la vision logique. On limite également les déplacements de la tête le lecture/écriture, coûteux en temps. Toutefois, il y a deux inconvénients :
- le risque d'absence d'une zone libre de taille suffisante,
- le déplacement possible du fichier, lorsqu'il s'agrandit, vers une zone libre de taille suffisante.

- **allocation non contiguë chaînée :** chaque bloc du fichier contient un pointeur sur le bloc suivant. Lorsque le fichier change de taille, la gestion des blocs occupés est simple. Il n'y a aucune limitation de taille, si ce n'est l'espace disque lui-même. Deux inconvénients toutefois :

- la perte d'un pointeur entraîne la perte de toute la fin du fichier ou bien il faut un double chaînage par sécurité

- le mode d'accès est totalement séquentiel

- **allocation non contiguë indexée :** on peut envisager 3 solutions :

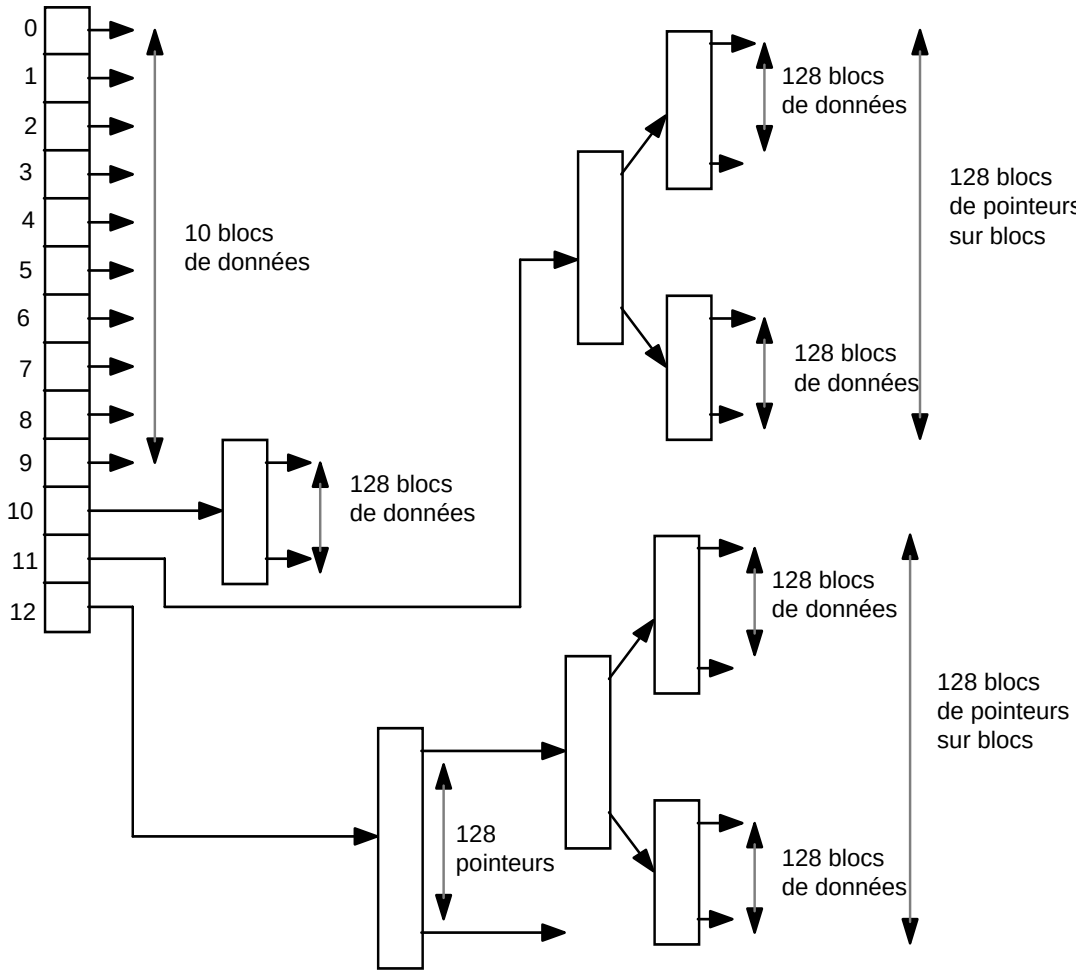
- une table globale d'index : elle évite la dispersion des pointeurs en rassemblant la liste des numéros de blocs utilisés par tous les fichiers. Cette table occupe beaucoup de place .

- une liste chaînée : dans le SE, donne la liste des blocs utilisés par les fichiers

- une table locale d'index : une table, en début de chaque fichier, contient l'ensemble des blocs utilisés. Il est donc aisé d'effectuer un accès direct à un bloc d'un fichier, ou de faire évoluer dynamiquement la taille d'un fichier. Quand cette table est petite, elle est stockée dans un bloc spécial ou **bloc d'index**. La taille des blocs d'index est une question difficile : s'ils sont trop petits, ils limiteront fortement la taille d'un fichier; s'ils sont trop grands, une grande partie de leur contenu est inutilisée ("fragmentation interne").

- **allocation non contiguë mixte :** on peut améliorer la solution précédente en utilisant plusieurs niveaux de tables d'index. Citons par exemple la méthode utilisée par UNIX :

- chaque fichier se voit affecter une table à 13 entrées
- les entrées 0 à 9 pointent sur un bloc de données
- l'entrée 10 pointe sur un bloc d'index qui contient 128 ou 256 pointeurs sur bloc de données (simple indirection)
- l'entrée 11 pointe sur un bloc d'index qui contient 128 ou 256 pointeurs sur bloc d'index dont chacun contient 128 ou 256 pointeurs sur bloc de données (double indirection)
- l'entrée 12 pointe sur un bloc d'index qui contient 128 ou 256 pointeurs sur bloc d'index dont chacun contient 128 ou 256 pointeurs sur bloc d'index dont chacun contient 128 ou 256 pointeurs sur bloc de données (triple indirection)



Pour les fichiers les plus longs, trois accès au disque suffisent pour connaître l'adresse de tout octet du fichier. Avec des blocs de 2 Ko, on peut aller jusqu'à 128 Go par fichier, mais UNIX limite la taille à 16 Go.

2. CORRESPONDANCE FICHIERS LOGIQUES-FICHIERS PHYSIQUES

Un **répertoire** est une table ou une liste chaînée dans laquelle le système range, pour chaque fichier :
- les informations nécessaires à son exploitation : droits, nom, dates, etc...
- les informations sur son implantation : taille, adresse de la table des index ou du premier bloc ou ...

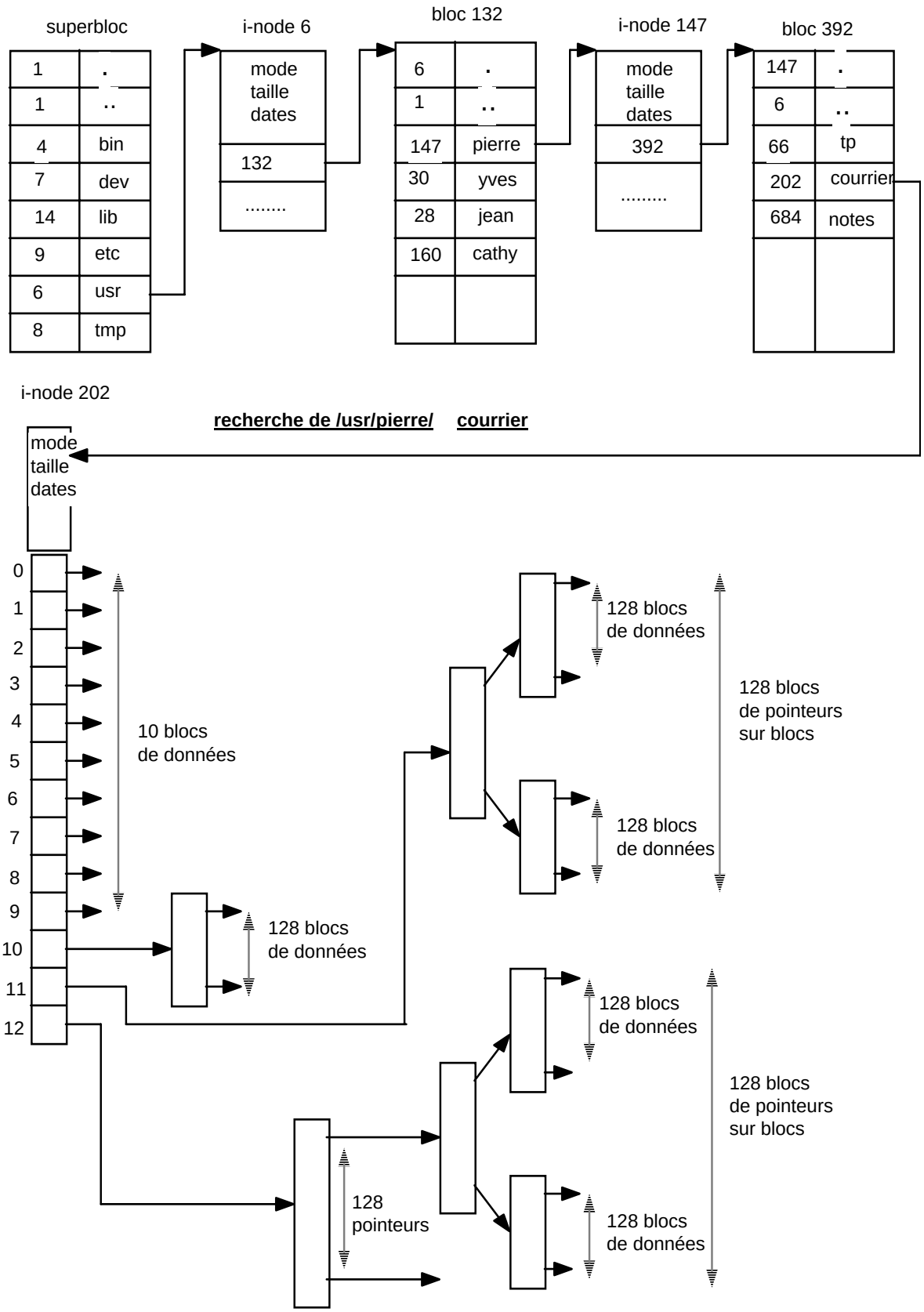
Il existe diverses solutions d'implémentation d'un répertoire :

- **répertoire à un niveau :** la table est gérée comme une liste. C'est une solution acceptable pour un SE mono-utilisateur, trop contraignant pour un SE multi-utilisateur (deux fichiers ne peuvent porter le même nom)

- **répertoire à deux niveaux :** chaque utilisateur possède son propre répertoire et il existe un répertoire de niveau supérieur qui associe à un nom d'utilisateur l'adresse de son répertoire.
Inconvénient : l'utilisation d'un fichier partagé (exemple : fichier de commande) nécessite soit sa recopie dans le répertoire de chacun de ses utilisateurs, soit l'énoncé de son chemin complet. Une autre solution est utilisée pour les fichiers système : on crée un utilisateur fictif qui contient tous les fichiers système, voire d'autres fichiers partagés. Quand un fichier est indiqué par un utilisateur, s'il n'existe pas dans son répertoire, le SE le cherche dans le répertoire fictif.

- **répertoire à structure d'arbre :** c'est le cas par exemple d'UNIX : *cf. schéma page suivante*

- **répertoire à structure de graphe sans cycle :** un fichier peut être dans les sous-arbres de plusieurs utilisateurs. Si un utilisateur le met à jour, il est modifié pour tous les autres. Un tel fichier possède plusieurs chemins d'accès, d'où un algorithme compliqué pour l'effacer. Au total, un répertoire peut contenir un pointeur sur un fichier ou un répertoire qui n'est pas dans le sous-arbre dont il est racine.


superbloc
1 .
1 ..
4 bin
7 dev
14 lib
9 etc
6 usr
8 tmp
i-node 6
mode
taille
dates
132
.....
bloc 132
6 .
1 ..
147 pierre
30 yves
28 jean
160 cathy
i-node 147
mode
taille
dates
392
.....
bloc 392
147 .
6 ..
66 tp
202 courrier
684 notes
i-node 202
recherche de /usr/pierre/ courrier
mode
taille
dates
0
1
2
3
4
5
6
7
8
9
10
11
12
10 blocs
de données
128 blocs
de données
128 blocs
de données
128 blocs
de pointeurs
sur blocs
128 blocs
de données
128 blocs
de données
128 blocs
de pointeurs
sur blocs
128
pointeurs
128 blocs
de données

3. COHERENCE D'UN SYSTEME DE FICHIERS

3.1 Cohérence au niveau des blocs

En parcourant la liste des blocs libres, on les note dans le tableau LIBRE. En parcourant tous les blocs d'information, on note les blocs utilisés dans le tableau UTILISE.

Si LIBRE [i] > 1, on supprime une des références au bloc i dans LIBRE
Si LIBRE [i] > 0 et UTILISE [i] > 0, on supprime la référence au bloc i dans LIBRE
Si UTILISE [i] > 1, on duplique le bloc i et on l'insère dans un des fichiers qui le référencient

3.2 Cohérence au niveau des fichiers

En parcourant tous les répertoires, on note pour chaque i-node le nombre d'entrées (n) qui pointent sur ce nœud.

Si n = 0, le fichier est mis dans un répertoire spécial (LOST+FOUND)

Si n = nombre L de liens déclarés dans l'i-node, c'est correct

Chapitre 9 LES INTERRUPTIONS

1. DEFINITIONS ET GENERALITES

Comment prendre en compte un événement, comment provoquer une rupture de séquence d'exécution d'un processus dans un délai très court ? Une solution : les interruptions.

1.1 Définitions

Def: Une interruption est un signal déclenché par un événement interne à la machine ou externe, provoquant l'arrêt d'un programme en cours d'exécution à la fin de l'opération courante, au profit d'un programme plus prioritaire appelé programme d'interruption. Ensuite, le programme interrompu reprend son exécution à l'endroit où il avait été interrompu.

Le système d'interruption est le dispositif incorporé au séquenceur qui détecte les signaux d'interruption. Ces signaux arrivent de façon asynchrone, à n'importe quel moment, mais ils ne sont pris en compte qu'à la fin de l'opération en cours.

Mécanisme général : Lorsqu'une interruption survient, le processeur achève l'exécution de l'instruction en cours pour les interruptions externes (cf. plus loin), puis il se produit :

1 Sauvegarde du contexte dans une pile :
- adresse de la prochaine instruction à exécuter dans le programme interrompu,
- contenu des registres qui seront modifiés par le programme d'interruption,
- contenu du mot d'état (registre de drapeaux) rassemblant les indicateurs

(tout cela forme le contexte sauvegardé)

2 Chargement du contexte du programme d'interruption (contexte actif) et passage en mode système (ou superviseur)

3 Exécution du programme d'interruption

4 Retour au programme interrompu en restaurant le contexte (commande Assembleur IRET) et en repassant en mode utilisateur.

1.2 Différents types d'interruptions

Les interruptions ne jouent pas seulement un rôle important dans le domaine logiciel, mais aussi pour l'exploitation de l'électronique.

On distingue donc :

• interruptions internes : protection du système et des processus, appelées par une instruction à l'intérieur d'un programme (overflow, erreur d'adressage, code opération inexistant, problème de parité…) *(hardware internes)*

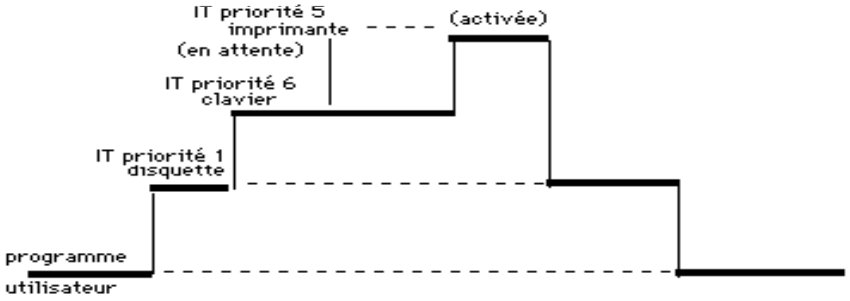
• interruptions logiques :permet à un processus utilisateur de faire un appel au système *(software)*

• interruptions matérielles déclenchées par une unité électronique (lecteur, clavier, canal, contrôleur de périphérique, panne de courant,....) ou par l'horloge *(hardware externes)*

1.3 Les priorités

A chaque interruption, est associée une priorité (système d'interruptions hiérarchisées) qui permet de regrouper les interruptions en classes. Chaque classe est caractérisée par un degré d'urgence d'exécution de son programme d'interruption.

Règle : Une interruption de priorité j est plus prioritaire qu'une interruption de niveau i si j > i.



L'intérêt de ce système est la solution de problèmes tels que :

• arrivée de plusieurs signaux d'interruption pendant l'exécution d'une instruction,

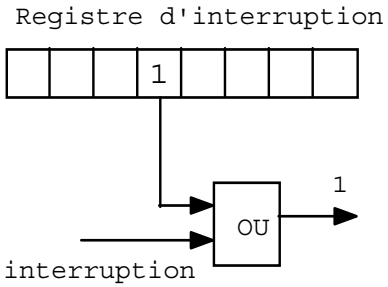
• arrivée d'un signal d'interruption pendant l'exécution du signal de traitement d'une interruption précédente.

On peut utiliser un contrôleur d'interruptions pour regrouper les fils d'interruptions, gérer les priorités entre les interruptions et donner les éléments de calcul d'adresse au processeur.

1.4 Masquage des interruptions

Certaines interruptions présentent tellement d'importance qu'il ne doit pas être possible d'interrompre leur traitement. On **masquera** alors les autres interruptions pour empêcher leur prise en compte. Certaines interruptions sont non-masquables : on les prend obligatoirement en compte. Une interruption masquée n'est pas ignorée : elle est prise en compte dès qu'elle est démasquée.

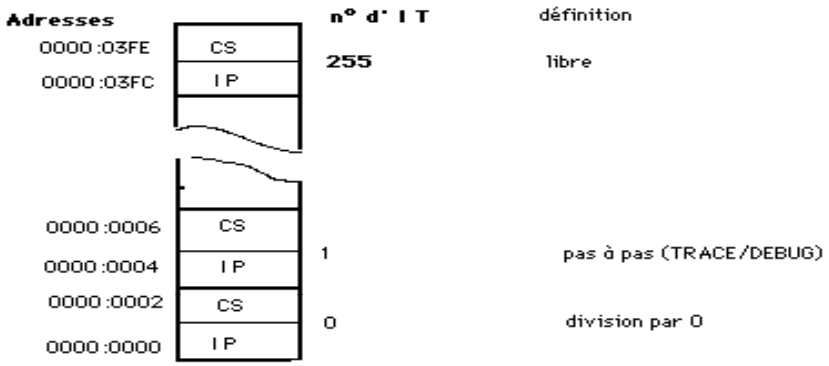
Au contraire, une interruption peut-être **désarmée** : elle sera ignorée. Par défaut, les interruptions sont évidemment armées.



Le bit correspondant à une interruption masquée est à 1 dans le registre d'interruption: sortie de OU à 1, donc masquage. Si le bit est à 0, pas de masquage.

2. INTERRUPTIONS VECTORISEES

Prenons l'exemple du microprocesseur 80386. Il peut comporter 256 interruptions numérotées de 0 à 255. Le lien entre chaque interruption et son programme est réalisé par une table ou vecteur d'interruptions.



Chaque élément contient l'adresse de début d'un programme d'interruption sur 4 octets (offset 2 octets, segment 2 octets). La taille de la table est donc de 4 x 256 = 1 Ko. Elle occupe les adresses 0h à 3FFh en mémoire centrale.

3. LES INTERRUPTIONS MATERIELLES : exemple du Intel 80386 et de ses successeurs

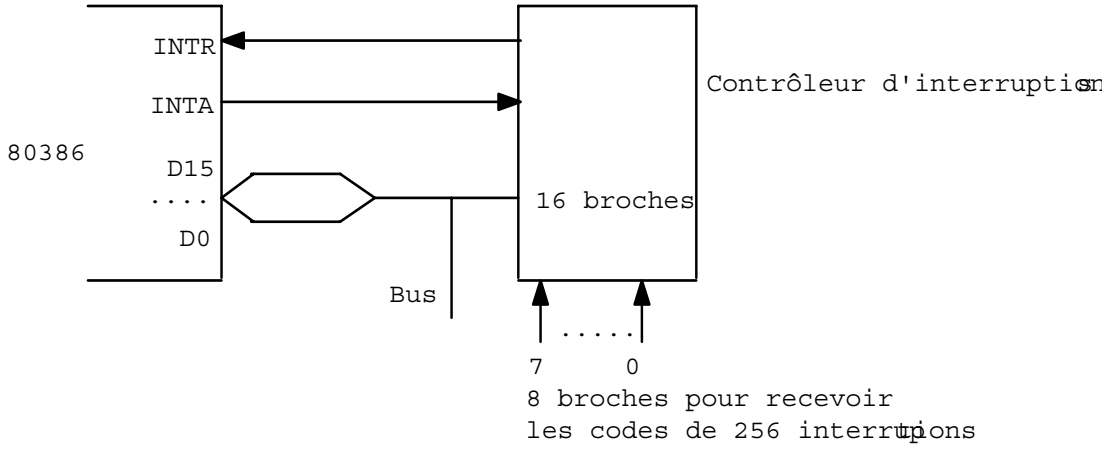
Elles sont déclenchées par une unité électronique. Par exemple, chaque fois qu'on presse ou relève une touche du clavier, l'IT clavier est déclenchée. Le programme d'interruption place le caractère entré dans le buffer à la suite des caractères entrés auparavant. Si le buffer est plein, il déclenche un bip ou une action convenue.

Les interruptions peuvent être masquées (exemple : clavier) grâce à l'instruction Assembleur CLI (CLear Interrupt flag) ou grâce à la fonction TURBO-C ® **disable ()**. Ainsi, après CLI ou disable (), aucun caractère ne peut être entré au clavier. Le flag IF (flag correspondant du registre d'état) est positionné à 0.

De même, l'instruction Assembleur STI (SeT Interrupt flag) ou la fonction TURBO-C ® **enable ()** démasque les interruptions en positionant le flag IF à 1.

La commande Assembleur NMI (No Mask Interrupt) permet de rendre une interruption non masquable. Les interruptions qui ne peuvent être masquables sont toujours exécutables, même après CLI ou disable ().

Dans le contexte Intel, les interruptions sont gérées par un **contrôleur d'interruptions** (circuit intégré 8259 ou circuit inclus dans le microprocesseur lui-même).Ce composant intercepte toutes les demandes d'interruption des périphériques et les communique à l'U.C. selon leurs priorités



INTR : interrupt request : le processeur est informé par le contrôleur de l'arrivée d'une demande d'IT

INTA : interrupt acknowledgement : le processeur a bien reçu par le bus de données le n° d'IT envoyé par le contrôleur. Le mécanisme détaillé de traitement d'une interruption non masquée est le suivant :

- l'U.C. termine l'exécution de l'instruction en cours

- elle lit sur le bus le numéro de l'interruption.

- elle sauvegarde le mot d'état dans la pile et l'adresse de retour CS : IP (adresse de l'instruction suivante)

- elle lit dans la table d'interruption à l'adresse :
[4 * n° d'IT] (mode calculé indirect)
Ainsi, à l'adresse [4*n°d'IT + 2] dans la table, elle trouve la partie segment CS de l'adresse du programme d'interruption et à l'adresse [4* n° d'IT] la partie offset de cette adresse.

- elle met à jour le compteur ordinal avec CS:IP

- elle exécute le programme s'achevant par IRET

Le **ROM BIOS** (Read Only Memory Input/Output System) et le **DOS** représentent le noyau du système d'un micro-ordinateur. Les routines du ROM BIOS permettent essentiellement le traitement des opérations d'entrée/sortie et d'interface avec les périphériques.

Citons notamment les principales interruptions du ROM BIOS :

IT 8 : tic horloge (timer) appelé toutes les 55 ms (18,2 fois/s.). Elle sert par exemple à arrêter le moteur de l'unité de disquette lorsque aucun accès à la disquette n'est exécuté. Cette interruption 08H, après avoir lancé le programme correspondant, appelle l'interruption 1CH. Cette dernière ne contient qu'un retour IRET afin de permettre aux programmeurs d'implanter leurs propres programmes d'interruption (objectifs : programmation concurrente, multitâche,....)

IT 10h à 12h : variable selon le matériel connecté. Souvent, IT 10 désigne l'IT vidéo IT 11 fournit la liste du matériel disponible et IT 12 le calcul de l'espace mémoire disponible

IT 13h : gestion des disques
IT 14h : gestion de la porte série
IT 15h : gestion d'un lecteur de cassettes
IT 16h : gestion du clavier
IT 17h : imprimante
IT 18h : activation du ROM BIOS
IT 19h : reset du système
IT 1Ah : gestion de la date et de l'heure

Les IT de 0 à 7 sont appelées directement par l'U.C. du microprocesseur . Il s'agit de :

IT 0 : division par 0
IT 1 : pas à pas
IT 2 : circuit RAM défectueux
IT 3 : point d'arrêt
IT 4 : débordement
IT 5 : copie d'écran
IT 6 et 7 : inutilisées

Les routines associées au DOS travaillent à un niveau supérieur. Elles peuvent parfois faire appel à des routines du ROM BIOS. Les principales interruptions associées au DOS sont :

IT 20h : fin normale d'un programme
IT 21h : appel de sous-routines système
IT 22h : gestion de l'adresse de saut après l'exécution d'un programme
Elle ne peut être activée directement. Elle précise la routine qui reprend le contrôle après la fin du programme
IT 23h : gestion de l'adresse de saut après un control-break. Idem à ci-dessus.
IT 24h : gestion de l'adresse de saut en cas d'erreur
IT 25h : lecture directe sur disque
IT 26h : écriture directe sur disque
IT 27h : programme résident (terminer un programme en le laissant résident). Un programme est dit ***résident*** si , une fois son exécution terminée, il rend le contrôle à l'environnement appelant mais reste chargé en mémoire. Il pourra alors être réactivé à tout moment , notamment à l'aide d'une IT. La fonction de Turbo-C void **keep** (int retour, in taille) , prototypée dans dos.h, suspend le programme en cours d'exécution, le rend résident. taille désigne sa taille qui peut être déterminée par un appel à MK_FP et code désigne le code retourné à l'environnement par ce programme.

4. ACTIVATION D'INTERRUPTIONS A PARTIR D'UN PROGRAMME C

Fonction **int86** (int num, union REGS * r_in, union REGS * r_out) : génère l'interruption num en fonction des contenus des registres passées en paramètres via r_in . La structure union REGS est définie dans dos.h

Fonction **int86x** (int num, union REGS * r_in, union REGS * r_out, union REGS * r_seg) : utilisée lorsque l'IT nécessite pour son traitement des zônes de mémoire. Pour cela, les registres d'adressage de segments sont passés comme paramètres

Fonction **geninterrupt** (int num) : déclenche l'IT num sans effectuer de copie de registres, mais en initialisant les registres à l'aide de pseudo variables définies dans dos.h : _ah, _al, _ch, etc...

Fonctions **intdos** (union REGS * r_in, union REGS * r_out) et **intdosx** (union REGS * r_in, union REGS * r_out, union REGS r_seg) : permettent d'appeler directement une routine de traitement de l'IT 21h

Fonction **biosequip** : détermine la liste des équipements disponibles (IT 11h)

Fonction **biosmemory** : calcule l'espace mémoire disponible (IT 12h)

Fonction **biosdisk** (int cmd, int face, int piste, int secteur, int n_secteur, void * buffer) : appelle la fonction cmd de l'IT 13h

Fonction **bioscom** (int cmd, char octet, int port) : appelle la routine cmd de l'IT 14h. octet représente l'octet à écrire si cmd = 1, port désigne le n° de la porte

Fonction **bioskey** (int cmd) : appelle la routine cmd de l'IT 16h

Fonction **biosprint** (int cmd, char octet, int port) : appelle la routine cmd de l'IT 18 h. octet représente l'octet à écrire si cmd = 0, port donne le n° de la porte parallèle

Fonction **biostime** (int cmd, long new_time) : appelle la fonction cmd de l'IT 1Ah. new_time contient la nouvelle heure en nombre de battements si cmd = 1

Il existe en Turbo-C le type **interrupt**. Lorsqu'une fonction est déclarée de ce type, TURBO-C va générer du code pour sauver les registres en début d'exécution et pour terminer la routine avec l'instruction IRET signalant le retour d'une interruption. La routine de service doit éviter un appel aux routines système car le DOS n'est pas réentrant.

La fonction **setvect** (int num, void interrupt (*f) ()) insère la fonction f en tant que routine de service de l'IT num. Pour éviter cette opération, il faut éviter d'être interrompu. Il faut donc faire précéder l'appel à setvect d'un appel à disable () qui masquera toutes les IT. Ensuite, un appel à enable () rendra à nouveau possible l'interruption du programme.

Chapitre 10 NOTIONS SUR WINDOWS NT

Le système d'exploitation WINDOWS NT *(New Technology)* a été conçu par Microsoft entre 1988 et 1993. Cinq objectifs étaient poursuivis en créant ce nouveau SE :

- offrir une interface homme-machine conviviale et standard pour tous ceux qui étaient familiers de l'environnement Windows sous MS-DOS, voire d'Apple

- conception modulaire du SE permettant son évolution

- portabilité du code sur toute machine utilisant un espace d'adressage sur 32 bits

- compatibilité **binaire** avec toutes les applications Microsoft existantes et compatibilité au niveau source avec la norme POSIX (Portable Operating System Interface for Computer Environment, 1988 et 1990). Cette norme est destinée à rendre les applications compatibles avec les différentes versions d'UNIX

- optimisation des services système et mécanisme d'échanges à grande vitesse pour accroître les performances

Outre la version de base pour postes de travail de gamme convenable, WINDOWS NT existe dans la version *Advanced Server* qui offre des services étendus de partage de ressources et d'administration.

1. CARACTERISTIQUES DE WINDOWS NT

4 caractéristiques principales permettent à WINDOWS NT d'être considéré comme un descendant évolué d' OS/2 d'IBM, de VMS de DEC et d'UNIX :

1.1 Utilisation du modèle client-serveur

Il assure l'accès transparent aux services de WINDOWS NT pour les clients hétérogènes WINDOWS, MS-DOS, OS/2 ou POSIX. Le dialogue entre clients et serveur se fait par échange de messages.

1.2 Modèle en couches

Il est utilisé notamment dans le système d'E/S et dans la partie de plus bas niveau : le noyau et la couche d'abstraction du matériel (Hardware Abstraction Layer, **HAL**)

1.3 Modèle de traitement symétrique

SMP (Symmetric MultiProcessing) permet de mieux exploiter la puissance du processeur physique, notamment tous les processeurs à architecture CISC (Complex Instruction Set Computer) tel que le 486 ou le Pentium d'INTEL ou à architecture RISC (Reduced Instruction Set Computer).

1.4 Utilisation des objets

L'utilisation des objets pour représenter les ressources du système permet de gérer les ressources de façon homogène et sécurisée.
Windows NT peut travailler selon deux modes :

- **le mode utilisateur** : mise en œuvre des sous-systèmes protégés ou serveurs, ainsi que des applications clientes

- **le mode moyau** : exécution des fonctions système ou exécutif NT

2. MODE UTILISATEUR

Dans ce mode, on accède à l'exécutif NT (code système) uniquement à travers ses services. Les sous-systèmes ou serveurs sont protégés.

Les communications entre sous-systèmes se font par échange de messages et plus rarement par partage de mémoire. Chaque serveur dispose d'une interface de programmation d'application (Application Programming Interface ou **API**) que les clients et autres serveurs peuvent appeler.

On distingue deux classes de sous-systèmes protégés ou serveurs :

2.1 Les sous systèmes d'environnement

Ils offrent un pseudo-environnement Windows, MS-DOS, OS/2 ou POSIX et proposent chacun une API spécifique avec un espace d'adressage privé pour être protégés les uns des autres :

- le sous-système **Win 32** offre l'API de Windows et l'interface graphique de WindowsNT (fenêtres). Il gère aussi les saisies utilisateur et les sorties de données de toutes les applications. Il est le lien entre l'utilisateur et le reste du SE. Lors du lancement d'une application, Win 32 crée un processus et en donne le contrôle au sous-système concerné

- le sous-système **POSIX** offre une API pour applications UNIX

- le sous-système **OS/2** offre une API pour applications OS/2

- le sous-système **MS-DOS** (Virtual DOS Machine ou **VDM**, client de Win 32) offre aux applications MS-DOS le contexte d'un processus appelé machine virtuelle DOS 5.0, avec un processeur virtuel Intel x86, sans son SGF propre , mais avec espace d'adressage privé et pilotes

- le sous-système **Windows 16 bits** (**WOW**, Windows on Win 32, client de Win 32) est une VDM multitâches dont chaque tâche fait tourner une application Windows 16 bits. Le code du noyau de Windows 3.1 est chargé au-dessus du code de MS-DOS. La gestion des fenêtres est faites par l'API Win 32

2.2 Les sous systèmes intégraux

Ils exécutent les fonctions de base du SE :

- les sous-systèmes **réseau** gèrent les demandes d'E/S sur le réseau

- le sous-système de **sécurité** maintient une base de données contenant des informations de sécurité sur les comptes utilisateurs locaux ou distants. Il est chargé de l'authentification des utilisateurs lors d'une tentative de connexion locale ou via le réseau.

Ainsi, dès qu'un utilisateur ouvre une session dans WINDOWS NT après avoir fourni un nom de compte et un mot de passe corrects, le sous-système de sécurité construit un objet appelé ***jeton d'accès*** et l'attache en permanence au processus utilisateur lancé à la connexion. Un jeton d'accès comprend l'identificateur personnel de sécurité de l'utilisateur (Security Identifier ou **SID)** et la liste des groupes dont il fait partie. Le jeton d'accès identifie tout processus et ses tâches (threads) auprès du SE.

Lorsqu'on crée un objet partageable par plusieurs processus, un descripteur de sécurité est créé dans l'en-tête de l'objet. Il s'agit d'un pointeur sur une liste de contrôle d'accès (Access Control List ou **ACL**) dont chaque élément (Access Control Entry ou **ACE**) comporte deux champs : un utilisateur ou un groupe, une liste de droits d'accès à l'objet (cf. exemple ci-dessus). Pour des raisons d'efficacité, les contrôles sont effectués à l'ouverture d'un objet et non pas à chaque utilisation.

3. MODE NOYAU

C'est le mode dans lequel tourne le code système ou **exécutif** de WINDOWS NT. On ne peut accéder que dans ce mode au matériel et à la mémoire.

L'exécutif est le moteur du SE. Il peut gérer un nombre quelconque de processus serveurs.

L'exécutif NT est composé de plusieurs éléments :

3.1 Le gestionnaire d'objets

gère les structures de données de l'exécutif, modélisant les ressources du SE. Elles sont utilisées par les programmes en mode utilisateur au travers des services.

3.2 Le moniteur de référence de la sécurité

surveille les ressources du SE en contrôlant l'accès aux objets par les processus. Quand un processus demande l'ouverture d'un objet, le moniteur consulte la liste de contrôle d'accès de l'objet et le jeton d'accès du processus.

3.3 Le gestionnaire de processus

crée, termine les processus et les tâches (**threads**), suspend et relance l'exécution des tâches, gère leurs données

3.4 L'appel de procédures locales

LPC (Local Procedure Call) transmet les messages entre client et serveur sur la même machine

3.5 Le gestionnaire de mémoire virtuelle

gère la pagination, fournit un espace privé d'adresses mémoire pour chaque processus et en protège l'accès

3.6 Le noyau

traite les interruptions, réalise l'ordonnancement des tâches, synchronise les processus, fournit des objets et des interfaces de bas niveau au reste de l'exécutif

3.7 Le système d'E/S

Il comprend :

- le gestionnaire d'E/S qui gère des modèles d'E/S indépendants des périphériques

- les pilotes de fichiers (drivers) qui traduisent les demandes d'E/S fichiers

- le redirecteur et le serveur du réseau : pilotes spécifiques qui transmettent et reçoivent des demandes d'E/S avec une autre machine du réseau

- les pilotes de périphériques (device drivers) gèrent la lecture ou l'écriture sur un périphérique physique ou un réseau

- le gestionnaire de mémoire cache

3.8 La couche d'abstraction du matériel (HAL)

masque à WINDOWS NT les détails d'implémentation de la plate-forme : contrôleurs d'interruption, mécanismes de communication multiprocesseur, interfaces d'E/S, etc...

Chapitre 11 L'EXECUTIF DE WINDOWS NT
Objets, processus et gestion de la mémoire

Comme il a été indiqué au chapitre 9, l'exécutif de WINDOWS NT met des sous-services standard à la disposition des applications clientes, gère les processus et les tâches (threads), gère la mémoire et les E/S.

Les ressources système sont représentées par des objets, mais WINDOWS NT est écrit en C de Microsoft et ne dispose pas vraiment de structures orientées objet.

1. GESTION DES OBJETS

Les objets de l'exécutif permettent trois fonctionnalités :

- offrir des noms explicites pour les ressources du système
- partager ressources et données entre les processus
- protéger les ressources

Seules les structures concernées par ces traitements sont modélisées par des objets.

Il existe deux sortes d'objets :

- **les objets de l'exécutif** : accessibles par les sous-systèmes protégés en mode utilisateur (processus, tâche, jeton d'accès, sémaphore, fichier).

- **les objets du noyau** : accessibles par les couches basses du système (interruption, tâche-noyau, processus-noyau, etc...)

Définition : on appelle *handle d'objet* un index dans une table de pointeurs sur les objets spécifique à un processus. Cette table comprend des pointeurs vers tous les objets pour lesquels le processus a ouvert un *handle*. Un *handle* est donc un pointeur indirect vers un objet. Seul le gestionnaire d'objets a le droit de créer un *handle*.

Pour un processus, un *handle* d'objets est un pointeur indirect sur des ressources du système. On évite ainsi que les applications manipulent directement les structures de données du système.

1.1 Structure des objets

Un objet comporte deux parties :

- **un en-tête** contrôlé par le gestionnaire d 'objets. Les services génériques (fermer un handle, dupliquer un handle, interroger l'objet, interroger la sécurité, changer la sécurité, attendre un objet) accèdent aux données de l'en-tête.

- **un corps** contrôlé par le sous-ensemble du SE qui a créé ce type d'objet. Le corps comprend plusieurs champs : type d'objet, attributs de l'objet, services de l'objet.

1.2 Gestionnaire d'objets

Il assure des fonctionnalités nombreuses :

- vérification des droits d'accès des tâches et processus sur les objets et réalisation de leur protection

- création et suppression des objets; surveillance des objets temporaires pour les supprimer dès qu'ils ne sont plus utilisés; suivi des objets en utilisant leur en-tête

- centralisation des opérations de contrôle des ressources

- attribution des noms aux objets (chaque nom est global sur une machine, mais pas dans le réseau) et implantation des objets nommés dans une mémoire globale gérée dynamiquement, attribution d'alias aux noms. Cet espace est structuré en arbre.

2. LES PROCESSUS ET LES TÂCHES (THREADS)

Chaque processus est découpé en une ou plusieurs unités d'exécution ou tâches ou *threads*. Le découpage en *threads* permet à un processus de paralléliser l'exécution d'une partie de son code ou d'une partie de celui-ci.

Processus et *threads* sont implémentés sous formes d'objets. Le gestionnaire de processus de l'exécutif crée ou supprime des processus et des *threads* et gère les communications interprocessus. Le gestionnaire de processus fournit des services pour la gestion des processus liés aux sous-systèmes d'environnement. Il n'y a pas de relation père-fils entre les processus natifs de WINDOWS NT.

2.1 Les processus

Un processus WINDOWS NT comprend :

- un programme exécutable (code et données)
- un espace d'adressage virtuel *privé*
- un ou plusieurs threads auxquels des ressources système sont affectées (sémaphores, fichiers, ports, ...)

La plupart des processus fonctionnent en mode utilisateur : ainsi, leur espace d'adressage est protégé des applications et des autres sous-systèmes.

A chaque processus est associé un ensemble de ressources : jeton d'accès, table d'objets, structure de données pour le gestionnaire de la mémoire virtuelle

La structure de processus de l'exécutif NT fournit des mécanismes de base qui permettent aux sous-systèmes d'environnement (POSIX, Win32, OS/2) de construire leur propre structure de processus et de la gérer.

2.2 Les tâches

Un thread appartient à un processus et à un seul. Sa structure comporte les éléments du contexte:

- un compteur ordinal
- une pile en mode utilisateur
- une pile en mode noyau
- une zone de données privées
- un mot d'état du processeur

Tous les threads d'un processus peuvent accéder à son espace d'adressage, aux handles d'objets et à ses autres ressources.

Lorsqu'un thread en mode utilisateur fait appel au SE, il est basculé en mode noyau et le service demandé est exécuté après vérification de ses paramètres. Puis, le thread est basculé à nouveau en mode utilisateur.

Il existe deux moyens de communications entre threads :

- les **objets de synchronisation** qui implémentent des fonctions d'attente et de signalisation. Un objet de synchronisation possède deux états : signalé et non signalé

- la **fonction d'alerte**, notifiée par l'APC (Asynchronous Procedure Call) permet d'interrompre l'exécution d'un thread pour lui faire exécuter une procédure (à comparer aux signaux d'UNIX)

3. COMMUNICATION INTER-PROCESSUS

Il existe 3 moyens de faire communiquer des processus avec WINDOWS NT : les appels de procédure locale, les appels de procédure distante et les canaux nommés.

3.1 LPC, Local Procedure Call

C'est un mécanisme rapide de transmission de message entre processus situés sur la même machine.

Le processus client envoie une demande de connexion sur le port de connexion du processus serveur. Le processus serveur crée deux objets-ports de communication pour établir un canal entre client et serveur et retourne au client l'un des handles correspondant.
A la mise en place du canal de communication, le processus client indique la méthode LPC qu'il veut utiliser parmi les trois suivantes :

- **envoi d'un message dans la file de message** de l'objet-port du processus serveur (1 bloc de message = 256 octets). Cette méthode, valable pour des messages courts, relie les espaces d'adressage des processus client et serveur par l'intermédiaire de la mémoire système allouée à l'objet-port

- **envoi, au port du serveur, d'un pointeur sur le message et passage du message en mémoire partagée**. Le processus client crée un objet de mémoire partagée : objet-section associé au port de communication du client, mais visible des deux processus. Lorsque le processus serveur répond, il crée aussi un objet-section associé à son propre port de communication

- **transmission d'un message à un thread spécifique d'un serveur à travers une mémoire partagée**. Ce mécanisme est appelé "LPC rapide". Il est utilisé par Win32 pour augmenter ses performances dans la gestion des fenêtres et des graphismes. Dès qu'un processus serveur reçoit une demande de transmission LPC rapide de messages sur son port de connexion, Win32 crée trois ressources pour le client :

  * un thread pour le traitement des requêtes
  * un objet-section de 64 Ko de mémoire partagée pour la transmission des messages
  * un objet-"paire d'événements" pour synchroniser le thread client et le thread serveur

3.2 RPC, Remote Procedure Call

L'objectif est de permettre à une application distribuée sur plusieurs machines d'appeler des services disponibles sur diverses machines du réseau. Ce mécanisme est conforme aux standards RPC de l'OSF.

Pour traiter des procédures distantes, une application utilise une bibliothèque locale de liaison dynamique (Dynamic Link Library ou **DLL**), comprenant des procédures relais qui portent le même nom que les procédures distantes et possèdent les mêmes interfaces.

Le relais récupère une copie des données pointées par un de ses paramètres, résout les références et appelle des procédures d'exécution de RPC qui s'adressent à la machine distante.

3.3 Les canaux nommés

Ce mécanisme fait abstraction des problèmes de routage et de transmission de données dans le réseau. Il est implémenté par un pseudo-SGF : le pilote du système de fichiers des canaux nommés.

Chaque canal nommé est implémenté par un objet-fichier et dispose des mêmes mécanismes de sécurité que les autres objets de l'exécutif. Un canal nommé est utile pour l'envoi de flots de données entre client et serveur d'une application distribuée.

4. Gestion des processus et threads par le noyau

Le noyau de WINDOWS NT est écrit en C de Microsoft et en Assembleur. Il assure le contrôle des processeurs et fournit des fonctions de bas niveau à l'exécutif NT. Pour l'essentiel, le noyau assure l'ordonnancement des processus et des threads, et gère les interruptions.

Le code du noyau n'est pas paginé hors de la mémoire et n'est pas interruptible. Il s'exécute toujours en mode noyau.

Le rôle du noyau peut se résumer à **4** éléments principaux :

4.1 Ordonnancement des threads

L'ordonnanceur des tâches est appelé "*module répartiteur du noyau*". Chaque tâche peut se trouver dans l'un des 6 états suivants :

- **prêt** : en attente d'exécution

- **standby** : un thread et un seul est dans cet état à un instant donné sur un processeur donné; c'est le thread dont l'exécution sera lancée à la prochaine commutation de tâches

- **en exécution** : état du thread en cours d'exécution. Si le noyau l'interrompt pour exécuter un thread de priorité plus élevée, il est placé en tête de la file d'attente, dans l'état **prêt**

- **attente** : le thread attend qu'un objet soit disponible ("mis en état signalé") . Le noyau l'informera que le répartiteur rendra disponible cet objet attendu

- **transitoire**: le thread pourrait être prêt, mais une ressource est indisponible. Dès que la ressource attendue est disponible, il passe à l'état **prêt**

- **terminé** : son exécution est achevée. Il est supprimé ou bien réinitialisé.

Les niveaux de priorité des threads sont numérotés de 1 (le plus bas niveau) à 31 (le plus élevé). Les niveaux 16 à 31 sont réservés aux threads temps réel et les niveaux 1 à 15 sont alloués aux threads à priorité variable. L'ordonnanceur gère une file d'attente par niveau de priorité.

4.2 Traitement des interruptions

Lorsque survient une interruption, le processeur transfère le contrôle au code de traitement des déroutements du noyau. Celui-ci transfère à son tour le contrôle au module chargé du traitement de l'interruption qui a été décelée.

Le noyau utilise un ensemble standard de priorités d'interruptions (Interrupt Request Level ou **IRQL**), portables d'un processeur à un autre.

Les exceptions (erreurs **synchrones** résultant de l'exécution d'une instruction en langage machine) sont traitées sont traitées en mode utilisateur ou en mode noyau selon un mécanisme analogue.

4.3 Synchronisation des processeurs

Sur une machine multiprocesseurs, le noyau assure la synchronisation des processeurs. Il dispose de primitives d'exclusion mutuelle pour la gestion de l'accès aux données globales à tous les processeurs (mécanisme dit de **verrou continu**).

Le noyau fournit aussi des mécanismes de synchronisation entre processeurs appelés **objets-répartiteurs**.

4.4 Reprise après interruption d'alimentation

Le noyau réserve à la coupure d'alimentation le deuxième niveau d'interruption par ordre décroissant (le niveau le plus élevé est alloué aux erreurs de bus et aux tests défectueux de la machine). L'objectif est de permettre la reprise des opérations d'E/S pour tout ordinateur muni d'une batterie de secours pour la mémoire, après une coupure d'alimentation. deux objets sont dédiés à la gestion de la reprise : objet-notification d'alimentation et objet-état d'alimentation.

5. GESTION DES ENTREES-SORTIES

Le système d'E/S de WINDOWS NT offre un accès aux périphériques sous forme de fichiers virtuels, manipulés par des *handles* de fichiers. Tous les pilotes de périphériques communiquent à l'aide de paquets de demandes d'E/S (Input/Output Request Packet ou **IRP)**. Tous les périphériques physiques, sources ou destinations, du système d'E/S sont modélisés par des ***objets-fichier*** . Dès le chargement d'un pilote de périphérique, l'objet-pilote associé et l'objet-périphérique associé permettent de traiter toute demande d'accès en occultant au maximum les caractéristiques physiques du périphérique.

Pour chaque opération d'E/S, le système d'E/S crée un IRP, le passe au pilote qui doit le traiter et récupère l'IRP lorsque l'E/S est achevée.

Les périphériques courants : clavier, souris, écran, imprimante, sont gérés par un **pilote monocouche**. Les unités de disques et les gestionnaires de réseaux ont des **pilotes multicouches** (pilote de fichiers, puis pilotes de périphériques).

Les principales caractéristiques du système d'E/S sont les suivantes :

- il admet plusieurs systèmes de fichiers : celui de MS-DOS, celui d'OS/2, CDFS (système de fichiers de CD-ROM), et bien sûr NTFS (celui de Windows NT)

- on peut ajouter ou enlever dynamiquement des pilotes; ceux-ci ont une structure uniforme et modulaire

- il protège ses ressources partageables par l'utilisation d'objets

- il supporte les interfaces d'E/S de Win32, OS/2 et POSIX

BIBLIOGRAPHIE SUCCINCTE

Helen CUSTER Au cœur de Windows NT, Microsoft Press, 1993
Bruno DARDONVILLE Architecture de Windows NT, Hermès, 1996
Paul YAO An Introduction to Windows NT Memory Management Fundamentals, Microsoft Development Library, 1992

Chapitre 12 LES SEMAPHORES SOUS UNIX

1. PRINCIPE DES SEMAPHORES SOUS UNIX

Pour créer un sémaphore, l'utilisateur doit lui associer **une clé.** le système lui renvoie un identificateur de sémaphore auquel sont attachés **n** sémaphores numérotés de 0 à n-1. Pour spécifier un sémaphore parmi les n, l'utilisateur indique l'identificateur et le numéro du sémaphore.

A chaque identificateur de sémaphore sont associés des **droits d'accès**. Ces droits sont nécessaires pour effectuer des opérations sur les sémaphores, mais inopérants pour :

- la destruction d'un identificateur de sémaphore,
- la modification des droits d'accès.

Seul le créateur, le propriétaire ou le super-utilisateur peuvent réaliser ces opérations.

1.1 Le fichier <sys/sem.h>

A chaque ensemble de sémaphores sont associées les structures semid_ds (une par ensemble de sémaphores), __sem et sembuf (une par sémaphore) décrites dans **<sys/sem.h>**

```
struct __sem        /* correspond à la structure d'un sémaphore dans le noyau */
{
        unsigned short  semval;         /* valeur du sémaphore */
        unsigned short  sempid;         /* pid du dernier processus utilisateur */
        unsigned short  semcnt;
                /* nombre de processus attendant l'incrémentation du sémaphore */
        unsigned short semzcnt;
                /* nombre de processus attendant que semval soit nul */
};

struct sembuf       /* correspond à une opération sur un sémaphore */
{
        unsigned  short sem_num;        /* n° de sémaphore : 0,.... */
        short sem_op;                   /* opération sur le sémaphore */
        short sem_flg;                  /* option */
 };

struct semid_ds
/* correspond à la structure d'une entrée dans la table des sémaphores */
{
        struct ipc_perm  sem_perm;      /* les droits */
        struct __sem * sem_base; /* pointeur sur le premier sémaphore de l'ensemble */
        time_t  sem_otime;              /* date dernière opération par semop     */
        ushort sem_nsems;/* nombre de sémaphores de l'ensemble */
        time_t sem_ctime;               /* date dernière modification par semctl */
};
```

1.2 La fonction semget

int semget (key_t cle, int nb_sem, int option)

nb_sem : nombre de sémaphores de l'ensemble

cle : si **cle** = IPC_PRIVATE, un nouvel ensemble de sémaphores est créé sans clé d'accès

sinon, il y a appel a **ftok**.

si **cle** ne correspond pas à un ensemble de sémaphores existant

si IPC_CREAT n'est pas positionné : erreur et retour de -1

sinon, un nouvel ensemble de sémaphores est créé et associé à cette clé. Les droits d'accès sont ceux mentionnés; le propriétaire et créateur est le propriétaire effectif du processus; le groupe propriétaire et créateur est le groupe propriétaire effectif du processus

sinon si IPC_CREAT **et** IPC_EXCL sont tous deux positionnés :

erreur et retour de -1

sinon la fonction retourne l'identificateur de l'ensemble de sémaphores

option : on peut combiner par | des droits d'accès et les constantes suivantes :

SEM_A : permission de modification

SEM_R : permission en lecture

IPC_CREAT , IPC_EXCL

La fonction : **key_t ftok (char * path , int id)** utilise une chaîne de caractères (en principe un nom de fichier unique dans le système), la combine à id pour générer une clé unique dans le système, sous forme d'un key_t (équivalent à un entier long).

Exemple : récupération de l'identificateur d'un ensemble de 5 sémaphores

```
num = semget (ftok (path, cle), 5, 0)
```

Exemple : /* création de 4 sémaphores associés à la clé 456 */

```
#include <errno.h>
#define CLE  456
main ()
{
int semid ;     /* identificateur des sémaphores */
char *path = "nom_de_fichier_existant";
if (( semid = semget (ftok (path, (key_t) CLE) , 4, IPC_CREAT | IPC_EXCL | SEM_R | SEM_A)) == -1)
{
        perror ("échec de semget\n");
        exit (1);
}
printf (" identificateur de l'ensemble : %d\n", semid);
printf ("clé de l'ensemble : %d\n", ftok (path, (key_t)  CLE));
}
```

1.3 La fonction semctl

int semctl (int semid, int semnum, int cmd, arg)

semid : identificateur de l'ensemble de sémaphores

semnum : numéro du sémaphore traité

```
union semun
{
 int val;
struct semid_ds *buf;
ushort array [ <nb de sémaphores>];
} arg;
```

cmd : paramètre de commande pouvant prendre 10 valeurs :
GETVAL : retourne la valeur du sémaphore de n° **semnum**
SETVAL : le sémaphore de n° **semnum** reçoit la valeur **arg.val**
GETPID : retourne le pid du processus qui est intervenu en dernier lieu
GETNCNT : retourne le nombre de processus en attente d'une incrémentation de la valeur du sémaphore de n° **semnum**
GETZCNT : retourne le nombre de processus en attente du passage à 0 du sémaphore de n° **semnum**
GETALL : range les valeurs de tous les sémaphores dans le tableau pointé par **arg.array**
SETALL : positionne les valeurs de tous les sémaphores à l'aide des valeurs du tableau pointé par **arg.array**
IPC_STAT : la structure associée à semid est rangée à l'adresse contenue dans **arg.buf**
IPC_SET : positionne les valeurs gid, uid et mode à partir de la structure pointée par **arg.buf**
IPC_RMID : détruit le sémaphore

1.4 La fonction semop

int semop (int semid, struct sembuf (* sops) [ ], int nbops)
retourne semval du dernier sémaphore manipulé ou -1 en cas d'erreur

semid : identificateur de l'ensemble de sémaphores

sops : pointeur vers un tableau de **nbops** structures de type sembuf (cf. page 1)

Pour chaque sémaphore de n° **sem_num** (0 à nbops-1), on indique dans le champ **sem_op** l'opération à effectuer avec un code entier :
si sem_op > 0 : semval = semval + sem_op
si sem_op = 0 : si semval = 0, rien n'est effectué
sinon le processus est endormi en attendant la nullité de semval
si sem_op < 0: si semval >= |sem_op|, alors semval ← semval - |sem_op|
sinon, le processus est endormi jusqu'à : semval >= |sem_op|
Pour le champ **sem_flg** :
s'il vaut IPC_NOWAIT, évite le blocage de processus et retourne un code d'erreur
s'il vaut SEM_UNDO, on évite de bloquer indéfiniment des processus sur des sémaphores après la mort accidentelle d'un processus. Très coûteux en temps CPU et en mémoire

2. LES OPERATEURS P ET V (WAIT et SIGNAL au sens de DIJKSTRA)

```
/* Appelons  ce fichier par exemple : "Dijkstra.h"
   Il est évidemment à inclure  dans le programme utilisateur*/
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
/*************************************************************/
int sem_create (key_t   CLEF,  int   initval)
/* crée un ensemble d'un seul sémaphore dont la clé est reliée à CLEF, de valeur initiale initval
    retourne l'identifiant associé au sémaphore si OK, sinon retourne -1
   Si l'ensemble associé à CLEF existe déjà, la fonction part en échec.
   Les droits sont positionnés de sorte que tout processus peut modifier le sémaphore   */
{
        union semun {
                int val;
                struct semid_ds *buf;
                ushort *array;
        }  arg_ctl;
        int semid;
        semid = semget ( ftok ("Dijkstra.h-<votre login>", CLEF), 1, IPC_CREAT | IPC_EXCL | 0666);
        if (semid == -1) return -1;
        arg_ctl.semval = initval;
        if (semctl (semid, 0, SETVAL, arg_ctl) == -1)  return -1;
        return semid;
}
/*************************************************************/
int sem_delete ( int semid)
/* supprime l'ensemble de sémaphores identifiés par semid */
{
        return semctl (semid , 0 , IPC_RMID , 0) ;
}
/*************************************************************/
int  P  (int  semid)
{
        struct sembuf sempar;
        sempar.sem_num = 0;
        sempar.sem_op =  -1;
        sempar.sem_flg = SEM_UNDO;
        return semop (semid , &sempar , 1);
}

/*************************************************************/
int  V (int semid)
{
        struct sembuf sempar;
        sempar.sem_num = 0;
        sempar.sem_op =  1;
        sempar.sem_flg = SEM_UNDO;
        return semop (semid , &sempar , 1);
}
```

En outre, la commande shell **ipcs** permet d'obtenir des informations sur les objets de type ensemble de sémaphores (et plus généralement tous les IPC), avec l'option :

- s pour se limiter aux sémaphores
- m pour se limiter aux segments de mémoire
- q pour se limiter aux files de messages
- c pour faire épparaître l'identité de l'utilisateur créateur

Elle fournit :

T : type de l'objet (q, m ou s)
ID : identification interne de l'objet
KEY : clé de l'objet en hexadécimal
MODE : droits d'accès à l'objet. Les 2 premiers des 11 caractères indiquent des informations spécifiques à une file de messages ou à un segment de mémoire partagée
OWNER : identité du propriétaire
GROUP : identité du groupe propriétaire

3. EXEMPLE DE PROGRAMMATION CONCURRENTE

```
main ()
{
        int  i, CLE = 33, mutex = 0;
        if ((mutex = sem_create ( CLE, 1)) == -1)
        {
                perror ("problème de création du sémaphore ");
                exit (-1);
        }
        printf ("création du sémaphore d'identifiant %d\n", mutex);
        if (fork () == 0)
        {
                printf ("je suis le processus %d, fils de %d\n", getpid (), getppid ());
                for (i = 0; i < 3 ; i++)
                {
                        P (mutex);
                        printf ("le fils entre dans sa section critique\n");
                        sleep (15);
                        printf ("le fils sort de sa section critique\n");
                        V (mutex);
                }
                exit (0);
        }
        else    {
                        for (i = 0 ; i < 4 ; i++)
                        {
                                P (mutex);
                                printf ("le pere entre dans sa section critique\n");
                                sleep (10);
                                printf ("le pere sort de sa section critique\n");
                                V (mutex);
                        }
                        wait (0);
                        printf ("FIN !\n");
                        sem_delete (mutex);
                }
}
```

Chapitre 13 LES TUBES

Nous avons vu dans les chapitres précédents deux moyens pour faire communiquer des processus sous UNIX : les signaux et les sémaphores. Les sémaphores, ainsi que les tubes, les files de messages et les segments de mémoire partagée constituent les **IPC** (Inter Process Communications), apparus avec UNIX System V. Le fichier **/usr/include/sys/ipc.h** contient la définition des objets et des noms symboliques utilisés par les primitives des IPC.

Qu'il s'agisse des sémaphores, des tubes, des files de messages ou des segments de mémoire partagée, il s'agit d'objets **externes** au SGF. Pour chacune de ces catégories, il existe une primitive (....get) d'ouverture/création (cf. ouverture des fichiers). Elle donne au processus une identification de l'objet interne (cf. descripteurs de fichiers).

1. DEFINITION ET CARACTERISTIQUES

Les tubes (pipes) constituent le mécanisme fondamental de communication entre processus sous UNIX. Ce sont des **files d'octets**. Il est conseillé de les utiliser de façon **unidirectionnelle**. Si l'on veut une communication bidirectionnelle entre deux processus, il faut ouvrir deux tubes, chacun étant utilisé dans un sens. Un tube est implémenté de façon proche d'un fichier : il possède un i-node, mais ne possède pas de nom dans le système ; son compteur de liens est nul puisque aucun répertoire n'a de lien avec lui.

Les tubes sont accessibles au niveau du shell pour rediriger la sortie d'une commande sur l'entrée d'une autre (symbole |).

Comme un tube ne possède pas de nom, il n'existe que le temps de l'exécution du processus qui le crée. Plusieurs processus peuvent lire ou écrire dans un même tube, sans qu'on puisse différencier l'origine des informations en sortie. La capacité d'un tube est limitée (PIPE_BUF octets).

Les processus communiquant par un tube doivent avoir **un lien de parenté** (par exemple descendance d'un ancêtre commun ayant créé ce tube : problème classique de l'héritage des descripteurs de fichiers par un fils).

2. CREATION ET UTILISATION D'UN TUBE

La création d'un tube correspond à celle de deux descripteurs de fichiers, l'un permettant d'écrire dans le tube et l'autre d'y lire par les opérations classiques **read** et **write** de lecture/écriture dans un fichier. Pour cela, on utilise la fonction prototypée par :

int pipe (int *p_desc)

où p_desc[0] désigne le n° du descripteur par lequel on lit dans le tube
et p_desc [1] désigne le n° du descripteur par lequel on écrit dans le tube.

La fonction retourne 0 si la création s'est bien passée et - 1 sinon.

La fonction C prototypée par :

FILE * fdopen (int desc, char *mode)

permet d'associer un pointeur sur FILE au tube ouvert ; mode doit être conforme à l'ouverture déjà réalisée.

3. SECURITES APPORTEES PAR UNIX

Un processus peut fermer un tube, mais alors il ne pourra plus le rouvrir. Lorsque tous les descripteurs d'écriture sont fermés, une fin de fichier est perçue sur le descripteur de lecture : read retourne 0.

Un processus qui tente d'écrire dans un tube plein est suspendu jusqu'à ce que de la place se libère. On peut éviter ce blocage en positionnant le drapeau O_NDELAY (mais l'écriture peut n'avoir pas lieu). Lorsque read lit dans un tube insuffisamment rempli, il retourne le nombre d'octets lus.

Attention : les primitives d'accès direct sont interdites; on ne peut pas relire les informations d'un tube car **la lecture dans un tube est destructive**.

Dans le cas où tous les descripteurs associés aux processus susceptibles de lire dans un tube sont fermés, un processus qui tente de lire reçoit le signal 13 SIGPIPE et se trouve interrompu s'il ne traite pas ce signal.

3.1 Ecriture dans un tube fermé en lecture

```
#include <errno.h>
#include <signal.h>
/*****************************************************************/
void it_sigpipe ()
{
        printf ("On a recu le signal SIGPIPE\n");
}
/*****************************************************************/
void main ()
{
        int p_desc [2];
        signal (SIGPIPE, it_sigpipe);
        pipe (p_desc);
        close (p_desc [0]);  /* fermeture du tube en lecture */
        if (write (p_desc [1], "0", 1) == -1)
                perror ("Erreur write : ");
}
```

Tester l'exécution de ce programme et expliquez son fonctionnement.

Autres exemples intéressants : Jean-Pierre BRAQUELAIRE, "Méthodologie de la programmation en Langage C", Masson, 1993, p. 344-345, 456-457

3.2 Lecture dans un tube fermé en écriture.

```
#include <errno.h>
#include <signal.h>
/*****************************************************************/
void main ()
{
int i , ret , p_desc [2];
char c;
pipe (p_desc);
write (p_desc[1], "AB", 2);
close (p_desc [1]);
for (i = 1; i <= 3; i++)
                    {  ret = read (p_desc [0], &c, 1);
                      if (ret == 1) printf ("valeur lue : %c\n", c );
                      else perror ("lecture impossible : ");
                    }
```

Tester l'exécution de ce programme et expliquez son fonctionnement

4. EXEMPLES DE PROGRAMMES.

4.1 Communication père-fils

Un fils hérite des tubes créés par le père et de leurs descripteurs.

```
/* exemple 1 */
#include <stdio.h>
#include <errno.h>
#define TAILLE  30          /* par exemple */
main ()
{
                    char envoi [TAILLE], reception [TAILLE];`int p [2], i, pid;
                    strcpy (envoi, "texte transmis au tube");
                    if (pipe (p) < 0)
                    {
                                perror ("erreur de création du tube ");
                                exit (1);
                    }
                    if ((pid = fork()) == -1)
                    {
                                perror ("erreur au fork ");
                                exit (2);
                    }
                    if (pid > 0)   /* le père écrit dans le tube */
                    {
                                for (i=0 ; i<5; i++) write (p[1], envoi, TAILLE);
                                wait (0);
                    }
                    if (pid == 0) /* le fils lit dans le tube */
                    {
                                for (i=0 ; i<5 ; i++) read (p[0], reception, TAILLE);
                                printf (" --> %s\n", reception);
                                exit (0);
                    }
}
```

Testez l'exécution de ce programme.

```
/* exemple 2  */
#include <stdio.h>
#include <errno.h>
#define NMAX 10              /* par exemple */
/*************************************************************/
void main ()
{
int pid , p_desc [2];
char c;
if (pipe (p_desc))
                    {  perror ("erreur en creation du tube ");
                       exit (1);
                    }
if ((pid = fork ()) == -1)
                    { perror ("echec du fork ");
                      exit (2);
                    }
if (pid == 0)
                    {  char t [NMAX];
                       int i=0;
                       close (p_desc [1]);
                       while (read (p_desc [0], &c, 1))
                                    if (i < NMAX)  t[i++] = c;
                       t [( i == NMAX) ? NMAX-1 : i ] = 0;
                       printf ("t : %s\n", t);
                    }
else                {   close p_desc [0]);
                       while ((c = getchar ()) != EOF)
                       if (c ≥ 'a' && c ≤ 'z')  write (p_desc [1], &c, 1);
                       close p_desc [1]);
                      wait (0);
                    }
}


/* exemple 3 */
#include <stdio.h>
#include <errno.h>
#define NMAX 10              /* par exemple */
/*************************************************************/
void main ()
{
int pid , p_desc [2] , m;
FILE *in, *out;
if (pipe (p_desc))
                    {  perror ("erreur en création du tube ");
                       exit (1);
                    }
out = fdopen (p_desc [0], "r");
in = fdopen (p_desc [1], "w");
if ((pid = fork ()) == -1)
                    { perror ("echec du fork ");
                      exit (2);
                    }

if (pid == 0)
                    {  int t[NMAX], i = 0;
                       fclose (in);
                       while (fscanf (out, "%d", &m) != EOF)
```

```
                        if (i <NMAX)  t[i++] = m;

                    t ( [ i == NMAX)  ?  NMAX-1 : i ] = 0;
                    for (m=0 ; m < i ; m++)  printf ("%d \n", m);
                }
..... cf. exemple 2 .....
}
```

Testez l'exécution de ces programmes et expliquez leur fonctionnement.

Autres exemples intéressants : Braquelaire p. 465-467.

4.2 Héritage des descripteurs lors d'un fork

```
#include <errno.h>
#include <stdio.h>
/*****************************************************************/
void code_fils (int numero)
{
        int fd, nread;
        char texte [100];
        fd = numero;
        printf ("le descripteur est : %d\n", fd);
        switch (nread = read (fd, texte, sizeof (texte)))
        {  case  -1 :   perror (" erreur read : ");
                        break;
           case 0 : perror ("erreur EOF");
                        break;
          default : printf ("on a lu %d caractères : %s\n", nread, texte);
        }
}
/*****************************************************************/
void main ()
{
        int p_desc  [2],
        char chaine [10];
        if (pipe (p_desc ))
                        {  perror ("erreur en creation du tube ");
                           exit (1);
                        }
        switch (fork())
        {   case  -1 :   perror (" erreur fork : ");
                        break;
           case 0 :if (close (p_desc [1]) == -1) perror ("erreur close : ");
                        code_fils (p_desc [0]);
                        exit (0);
        }
        close (p_desc [0]);
        if (write (p_desc [1], "hello", 6) == -1) perror ("erreur write :");
}
```

Testez l'exécution de ce programme et expliquez son fonctionnement.

4.3 Les redirections d'E/S standards : exemple de who | wc

```
#include <errno.h>
```

```
#include <stdio.h>
/*****************************************************************/
void main ()
{
int p [2], n;
if (pipe (p ))
                { perror ("erreur en création du tube ");
                  exit (1);
                }
if ( n= fork ()) == -1)
                { perror ("erreur avec fork : ");
                  exit (2);
                }
if (n == 0)             /* processus correspondant à who */
                {  close (1);           /* on ferme la sortie standard   */
                   dup (p [1]);     /* fournit un descripteur ayant les mêmes caractéristiques physiques que p[1], et ayant la valeur la plus faible possible. Ici, nécessairement : 1  */
                   close (p [1]); /* la sortie standard est ainsi redirigée sur l'entrée du tube  */
                  close (p [0]);
                  execlp ("who", "who", 0);
                /* on recouvre le processus par le texte de who */
                }
else            {               /* processus correspondant à wc  */
                  close (0);       /* on ferme l'entrée standard  */
                  dup (p [0]);     /* fournit un descripteur ayant les mêmes caractéristiques physiques que p[0], et ayant la valeur la plus faible possible. Ici, nécessairement : 0  */
                  close (p [0]);   /* l'entrée  standard est ainsi redirigée sur l'entrée du tube  */
                  close (p [1]);
                  execlp ("wc", "wc", 0);  /* on recouvre le processus par le texte de wc */
                }
}
```

Autre exemple intéressant : Braquelaire p. 472-478.

4.4 Communication entre deux fils

```
#include <errno.h>
#include <stdio.h>
/*****************************************************************/
void main ()
{
int pid1, pid2, ret, p_desc [2];
char *arg1, *arg2;
ret = pipe (p_desc);
sprintf (arg1, "%d", p [0]); /* conversion de p[0] en chaîne de caractères */
sprintf (arg2, "%d", p [1]); /* conversion de p[1] en chaîne de caractères */
pid1 = fork ();
if (pid1 == 0) execl ("fils1", "fils1", arg1, 0);
pid2 = fork ();
if (pid2 == 0) execl ("fils2", "fils2", arg2, 0);
……
/* suite du père */
……
}
```

La communication est bien : fils1 -> fils2. Avec 3 tubes, on réaliserait une communication circulaire : père -> fils1 -> fils2 -> père.

tube p1 : père -> fils1 (fils1 doit connaître p1[0])
tube p2 : fils1 -> fils2 (fils 1 doit connaître p2[1], fils 2 doit connaître p2 [0])

tube p3 : fils2 -> père (fils 2 doit connaître p3 [1])

Autre exmple intéressant : Braquelaire p. 467-472.

5. LES TUBES NOMMES

Un tube nommé possède un nom dans le système. S'il n'est pas détruit, **il persiste dans le système après la fin du processus** qui l'a créé. Tout processus possédant les autorisations appropriées peut l'ouvrir en lecture ou en écriture. Un tube nommé conserve toutes les propriétés d'un tube : taille limitée, lecture destructive, lecture/écriture réalisées en mode FIFO. Un tube nommé permet à des processus sans lien de parenté dans le système de communiquer.

Pour créer un tube nommé, on utilise la fonction : **int mknod (char *ref, int mode)**
qui retourne 0 si elle a pu créer un i-node dont la première référence est pointée par ref, avec les protections mode); en cas d'échec, le retour est -1.

```
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
/*****************************************/
void main ()
{
        printf ("on va créer un tube de nom fifo1 et un tube de nom fifo2\n");
        if (mknod ("fifo1", S_FIFO | 0666) == -1) /* S_FIFO obligatoire */
                { perror ("erreur de création fifo 1 ");
                 exit (1);
                }
        if (mknod ("fifo2", S_FIFO | 0666) == -1)  /* S_FIFO obligatoire */
                { perror ("erreur de création fifo2  ");
                  exit (1);
                }
        sleep (30);
        printf ("on efface le tube fifo1\n");
        unlink ("fifo1");
}
```

Lancez ce programme en arrière-plan et testez avec **ls** les créations et la suppression, ainsi que la présence de fifo2 après la fin du processus. Dans le catalogue, les tubes nommés apparaissent avec **p** en tête des droits d'accès.

Pour utiliser un tube nommé créé, il est nécessaire de l'ouvrir en lecture ou en écriture avec open.

Des précautions d'ouverture et d'utilisation sont à connaître. On se reportera avec profit à Jean-Marie RIFFLET, "La programmation sous UNIX", Mac-Graw-Hill, 1992, pages 263 à 267.

```
/* exemple 1  : l'exécutable sera nommé ex1 */
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <errno.h>
main (int n, char ** v)
{
        int p;
        if (mknod ( v[1], S_IFIFO | 0666) == -1)
        {
```

```
                        perror ("création impossible ");
                        exit (1);
                }
                if ((p = open (v[1], O_WRONLY, 0) )== -1)
                {
                        perror ("ouverture impossible ");
                        exit (2);
                }
                write (p, "ABCDEFGH", 8);
}

/* l'exécutable sera nommé ex2 */
#include <stdio.h>
#include <fcntl.h>
#include <errno.h>
main (int n, char ** v)
{
        int p;
        char c;
        if ((p = open (v[1], O_RDONLY, 0)) == -1)
        {
                perror ("ouverture impossible ");
                exit (2);
        }
        read (p, &c, 1);
        printf ("%c\n", c);
}
```

On lance : $ ex1 fifo&
(il s'affiche .....)
$ ex2 fifo
A