Systèmes d'exploitation

Michel Meynard

UM2

Univ. Montpellier 2 - 2009

Table des matières

Plan

Déroulement

- Cours 16.5h, TD 16.5h, TP 18h ;
- Contrôle : examen écrit (80/100), note CC (20/100) ;
- Prérequis : architecture, programmation C ;
- Conseils : voir et faire les examens précédents.

Type de systèmes étudiés : monoprocesseur, multitâche.

Contenu du cours

Les points suivants seront étudiés en théorie puis en pratique en utilisant la programmation en C sous Unix.

1. Besoins et rôles d'un système d'exploitation ; composantes d'un système ; noyau ; prérequis matériels ; contexte d'exécution ; pile.
2. Gestion des Entrées-Sorties
3. Processus : constituants, vie, ordonnancement, génération.
4. Gestion de l'espace disque, gestion des fichiers : système de gestion des fichiers ; représentation des fichiers et répertoires ; inodes, partitions et parcours. Gestion des fichiers ouverts.
5. Communications entre processus : moyens de communication simples et évolués ; synchronisation de processus et protections d'accès.
6. Gestion de la mémoire : principes, allocation, mémoire virtuelle, pagination.
7. Gestion des entrées-sorties : pilotes, controleurs.

Bibliographie

Andrew Tanenbaum, *Systèmes d'exploitation*, 2ème éd., Campus Press, 2003

J.M. Rifflet, J.B. Yunès *Unix, Programmation et communication*, Dunod, 2003

Gary Nutt, *Operating systems, a modern perspective*, 3rd edition, Addison-Wesley, 2003

Samia Bouzefrane, *Les systèmes d'exploitation, Unix, Linux et Windows XP avec C et Java*, Dunod, 2003

D.P. Bovet, M. Cesati, *Le Noyau Linux*, O'Reilly, 2001

M. J. Bach, *The design of the Unix operating system*, Prentice-Hall, 1986.

Plan

Composantes d'un Système d'exploitation

Un système d'exploitation doit gérer les ressources, c'est-à-dire partager les ressources communes de la machine et les allouer *au mieux* aux processus des utilisateurs.

gestion des processus

- allocation de la ressource UC (Unité Centrale) : plusieurs processus demandeurs, mais un seul élu à la fois ;
- illusion de parallélisme : temps partagé par **quantum** de temps ;
- Assurer l'isolement et la communication entre processus ;
- éviter la **famine** : un pus attend indéfiniment ;
- **ordonnancement** (scheduling) des pus ;

Composantes d'un Système d'exploitation - suite

gestion de l'espace disque

- arborescence de répertoires et fichiers ;
- répondre aux demandes d'allocation et libération de l'espace : création, suppression, modification de fichiers et répertoires ;
- **protection** par les droits d'accès des fichiers entre utilisateurs ;

gestion de la mémoire

- répondre aux demandes d'allocation et libération d'espace ;
- protéger les accès ;
- masquer l'espace physique (*mémoire virtuelle*) ;

gestion des Entrées/Sorties E/S

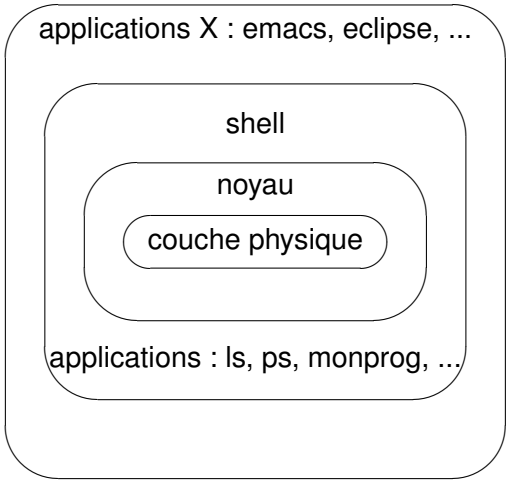
- entrées/sorties généralisées : lire depuis le clavier ou depuis un fichier de la même façon ;
- efficacité pour le système : réaliser les transferts, au moment *le plus opportun* ;
- gestion **tamponnée** (buffer) des E/S ;

Interface de Programmation d'Apllication (API)

Le système Unix offre une API naturelle en C puisque le système est écrit en C !
Cette API se décompose en 2 niveaux :

- **appels systèmes** : fonctions de bas niveau appelant directement le noyau ; décrits dans la section 2 du manuel (`man 2 write`)
- **fonctions de la bibliothèque standard C** : opérations de plus haut niveau ; décrits dans la section 3 du manuel (`man 3 printf`)

Structure en couches d'un système



Plan

Programme et processus

Un programme peut être écrit dans un langage :

- **interprété** : *script* bash ou python exécuté par l'interprète ;
- **compilé** : le *source* est compilé en *objet* puis il est *lié* avec d'autres objets et des *bibliothèques* afin de produire un *binaire* ;

Dans les deux cas, le *script* et le *binaire* doivent être *exécutables*, c'est à dire qu'ils doivent posséder le droit d'exécution `x` pour l'utilisateur.
Un processus est un programme *exécutable* en cours d'exécution. Par la suite, on s'intéressera principalement aux binaires C.

Programme C : hello.c

```
#include <stdio.h>/* printf */
#include <stdlib.h>/* malloc */
#include <string.h>/* strcat */

int global=0;
char *concat(const char *s1, const char *s2){ /* concat */
  char *s=(char*)malloc(sizeof(char)*(strlen(s1)+strlen(s2)+1));
  return strcat(strcat(s,s1),s2);
}
int nbappels(){
  static int nb=0;
  return ++nb;
}
int main(int n, char *argv[], char *env[]){
  nbappels();nbappels();nbappels();
  printf(concat("Bonjour ","le monde !\n"));
  printf("Nb appels : %d",nbappels());
}
```


Fabrication du binaire

En deux étapes : *compilation* puis *édition de liens* :

```
gcc -c -g -ansi -Wall -std=c99  hello.c
gcc -o hello hello.o
```

Toujours en deux étapes mais avec 1 seule commande :

```
gcc -g -ansi -Wall -std=c99 -o hello hello.c
```

Options : **C**ompil seulement, **g** debug, **ansi** norme du C, **W**arning all, **std** c99 pour les commentaires à la C++ ...

Déclaration et définition en C

- **déclaration** d'objet : indication du type de l'identificateur ; par exemple : `extern int i; void f();`
- **définition** d'objet : réservation d'espace mémoire ; par exemple : `int i; void f()return;`

Un même objet peut être déclaré plusieurs fois mais doit être **défini une seule fois**. Dans cette définition, il peut être **initialisé**.
Les fichiers d'**en-tête** (headers ou .h) ne contiennent généralement que des déclarations. Il sont de 2 sortes :

- `#include <stdio.h>` en-tête système dans `/usr/bin/include/` ;
- `#include "mon.h"` en-tête personnel dans • ;

Durée de vie et visibilité

- une variable définie en dehors de toute fonction est **globale** : sa durée de vie égale celle du processus et sa visibilité est totale ;
- une variable définie en dehors de toute fonction et précédée de `static` a une durée de vie égale celle du processus et sa visibilité est locale au fichier ;
- une définition de **fonction** précédée de `static` limite sa visibilité au fichier ;
- une variable définie dans une fonction et précédée de `static` a une durée de vie égale celle du processus et sa visibilité est locale à la fonction : elle n'est initialisée qu'une seule fois !
- une variable définie dans une fonction est **locale** a une durée de vie égale celle de la fonction et sa visibilité est locale à la fonction ;

Pointeurs et objets dynamiques

- Un objet dynamique (par opposition à `static`) peut être créé pendant l'exécution du processus grâce à la fonction `malloc(taille)` qui retourne un pointeur sur la zone du **tas** réservée.
- Ensuite, l'objet peut être modifié dans n'importe quelle fonction qui a connaissance de son adresse.
- Enfin, l'objet devra être détruit par `free(ptr)` afin de désallouer l'espace qui avait été réservé et d'éviter toute **fuite mémoire**.

Les objets dynamiques epuvent être de n'importe quel type (tableau, structure, entier, ...) et sont gérés dans le segment de **tas**.

Segments de mémoire d'un processus

On distingue dans un processus quatre segments de mémoire :

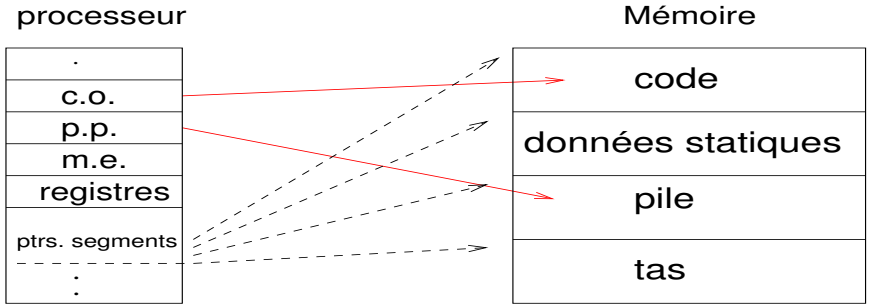
1. **Code** contenant la suite des instructions machine (le programme binaire).
2. **Données statiques** contenant les données définies hors de toute fonction (dites *globales*) ou définies *static* dans une fonction.
3. **Pile** contenant :
   - les adresses de retour des fonctions appelantes,
   - les paramètres d'appels et le résultat,
   - les variables locales.
4. **Tas** contenant les objets dynamiques (*new*, *malloc*, ...)

Parfois un cinquième segment de données statiques non initialisées existe ...

Durée de vie des objets dans les segments

1. **Code** : durée de vie égale à celle du processus ; accessible en lecture seule.
2. **Données statiques** : durée de vie égale à celle du processus ; les données statiques sont créées dès le lancement du processus et sont détruite avant la destruction du processus lui-même ;
3. **Pile** : une variable locale est créée à l'exécution même de l'instruction de définition ; elle est détruite à la fin du bloc (accolade fermante dans les langages comme C, C++, java, etc.).
4. **Tas** : la durée de vie d'un objet dans le tas va de sa création explicite (malloc) jusqu'à sa destruction explicite (free).

Contexte d'exécution d'un processus

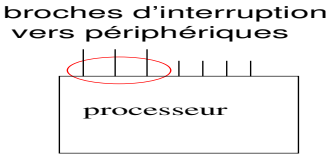


Le contexte d'exécution d'un processus est composé :

- le contenu des registres du processeur (compteur ordinal, pointeur de pile, mot d'état, ...) ;
- les segments de code, de pile et de données ;
- les informations systèmes relatives au processus : utilisateur(s), groupe, identifiant de pus, identifiant du parent de pus, priorité, ...

Plan

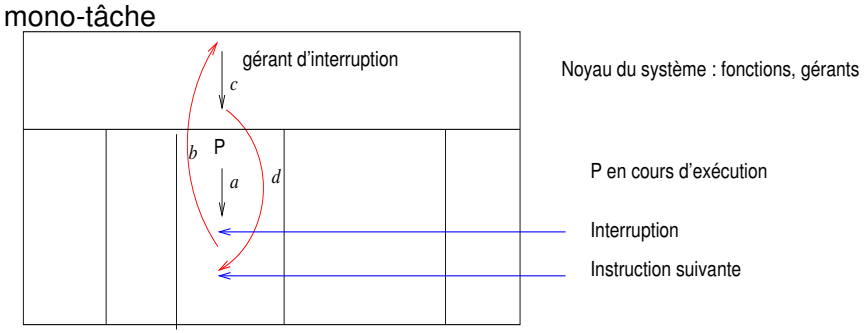
Interruption - système mono-tâche



Mécanisme matériel, permettant la prise en compte d'un événement extérieur, **asynchrone**. Fonctionnement :

1. l'instruction en cours est terminée ;
2. sauvegarde du contexte minimal (CO, PP, Mot d'État (PSW)) ;
3. une adresse prédeterminée est forcée dans CO ; à partir de là tout se passe comme tout appel de fonction ;
4. la fonction exécutée s'appelle *gérant d'Interruption* ; on dit que le gérant correspondant à l'interruption est lancé ;
5. fin du gérant et retour à la situation avant l'interruption : restauration du contexte précédent et suite de l'exécution.

Schéma de fonctionnement d'une interruption



Interruption - Utilité

Le mécanisme d'interruption est utilisé intensivement :

- par les divers périphériques, afin d'alerter le système qu'une entrée-sortie vient d'être effectuée ;
- par le système d'exploitation lui-même ; c'est la base de l'allocation de l'UC aux processus. Lors de la gestion d'une interruption, le système en *profite* pour déterminer le processus qui obtiendra l'UC. Les concepteurs du système ont écrit les gérants des interruptions dans ce but.

On en déduit que les systèmes multi-tâches vont détourner la dernière étape du déroulement de la gestion d'interruptions décrit précédemment.

Interruptions - Priorités

- plusieurs niveaux de priorité des interruptions ;
- un gérant d'interruption ne peut être interrompu que par une IT de plus haut niveau ;
- une interruption de même niveau est masquée jusqu'au retour du gérant ;
- ce mécanisme doit être assuré par le matériel : par exemple, `IRET` permet de retourner d'un gérant donc de récupérer le contexte de l'interrompu !

Interruption - Exemple de priorités

Un exemple dans l'ordre décroissant de priorité

1. horloge
2. contrôleur disque
3. contrôleur de réseau
4. contrôleur voies asynchrones
5. autres contrôleurs...

Exemples d'interruptions :

- le contrôleur disque génère une interruption lorsqu'une entrée-sortie vient d'être effectuée,
- le clavier génère une interruption lorsque l'utilisateur a fini de saisir une donnée,

Exceptions détectées et programmées

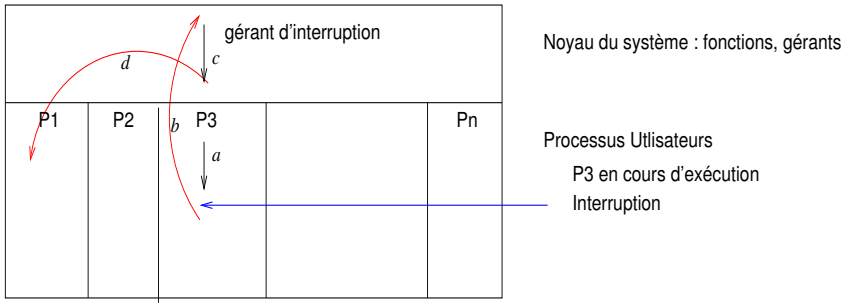
Une **exception** génère un comportement semblable à celui des interruptions (gérant d'exception) mais elle est **synchrone** ; 2 types d'exception :

- Exception détectée par le processeur : division par 0, overflow, défaut de page, ... appelé aussi *déroutement, trappe* ;
- Exception programmée : elle permet d'implémenter un appel système grâce à une instruction privilégiée (*interruption logicielle*) ;

Horloge - Timer

- Le **temps partagé** permet aux processus de se partager l'UC. Ce mécanisme est réalisé grâce aux interruptions Horloge qui ont lieu périodiquement à chaque **quantum de temps** ;
- En quelque sorte, une minuterie qui est enclenchée au début de chaque processus et qui se déclenche toutes les $\approx 10ms$;
- Le gérant d'interruption de l'horloge va alors appeler l'**ordonnanceur** (*scheduler*) qui est chargé d'élire le prochain processus !
- Les autres gérants d'interruption (E/S disque, ...) font de même !

Ordonnancement des processus



Mode d'exécution

2 modes d'exécution *alternatifs* d'un même processus :

1. **mode utilisateur** : mode normal dans lequel les instructions machines du programme binaire sont exécutées ;
2. **mode noyau** : mode système ou privilégié où certaines instructions *sensibles* peuvent être exécutées ;

Exemples d'instructions privilégiées : masquer les interruptions, manipuler l'horloge ou les tables systèmes, sortir de son espace de mémoire, . . .

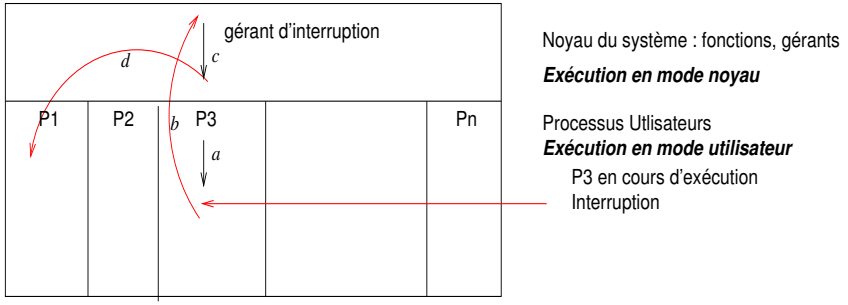
Raisons du passage en mode noyau :

- Exception (détectée (déroutement), ou programmée (appel système)) ;
- Interruption ;

Changement de mode d'exécution

Il effectue simultanément deux opérations :

1. bascule un bit dans le processeur (bit du mode d'exécution) ;
2. exécute le gérant d'interruption ou d'exception ;



Plan

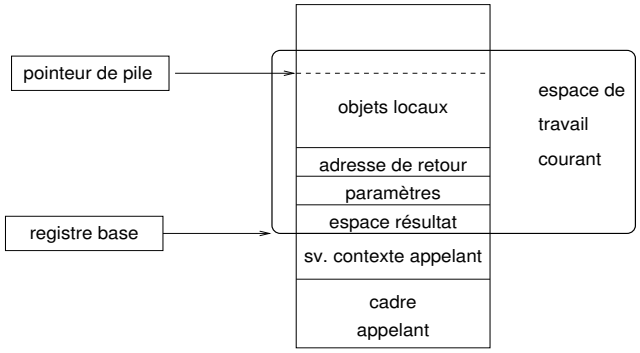
Petite histoire de la programmation

- programme : séquence d'instructions contenant des sauts conditionnels ou non (JMP, JC, ...) ;
- invention du registre de retour et des routines : limitation à un niveau d'appel : 1 main et des routines ;
- invention de la pile et de la programmation **procédurale** (CALL et RET) !
- programmation évoluée : fonctions, passage de paramètres, variables locales, ...
- programmation par objets, exceptions (traversée de la pile) ...

Fonctionnement de la pile

- à chaque occurence de fonction C, correspond un **cadre** (frame) contenant :
  - résultat éventuel (void)
  - paramètres passés par l'appelant
  - adresse de retour à l'appelant empilé par CALL
  - valeur du registre BP de l'appelant
  - variables locales de cette occurence de fonction
- ces différentes données sont accédées via le registre de base de pile BP, et le registre de sommet de pile SP
- au fur et à mesure des appels emboîtés, les cadres s'empilent
- au fur et à mesure des retours, les cadres sont dépilés

Espace de travail d'une fonction ou cadre



- l'appelant empile résultat et paramètres, puis réalise le CALL
- l'appelé sauve BP, le modifie, puis installe les var locales
- pendant l'exécution, l'appelé accède aux param et var locales par adressage indexé (BP+x) ou (BP-x)
- avant de RET, l'appelé nettoie les var locales et restaure BP
- après le CALL, l'appelant dépile les paramètres et récupère le résultat

Exemple : fact.c

```
#include <stdio.h>/* printf */
#include <stdlib.h>/* atoi */
int fact(int n){ int res=1;
  if (n<=1) return res;
  else {
    res=n*fact(n-1);
    return res;
  }
}
int main(int n, char *argv[], char *env[]){
  if (n!=2){
    fprintf(stderr,"Syntaxe : %s entier !\n", argv[0]);
    return 1;
  }
  int i=atoi(argv[1]);
  printf("factorielle de %s : %d\n",argv[1],fact(i));
  return 0;
}
```


Cadre : fact 2

type	commentaire
int	résultat du main
int	n nombre d'arguments
char**	argv
char**	env
void*	adrs retour
void*	BP de l'appelant
int	i==2 var locale du main
int	résultat de fact(2)==2 au retour
int	n==2 paramètre de fact(2)
void*	adrs retour au main
void*	BP du main
int	res==1 (puis 2) var locale de fact(2)
int	résultat de fact(1)==1 au retour
int	n==1 paramètre de fact(1)
void*	adrs retour à fact(2)
void*	BP de fact(2)
int	res==1 var locale de fact(1)

TABLE: 3 cadres de travail : main, fact(2), fact(1)

Pile - Remarques et Questions

Deux piles par processus :

- une pile utilisateur active en mode utilisateur ;
- une pile système active en mode noyau. La pile système est vide lorsqu'un processus est en mode utilisateur.

Un appel récursif se déroule comme tout appel emboîté

Plan

Noyau

Noyau d'un système d'exploitation : les fonctions du système d'exploitation qui **résident** en mémoire centrale dès le lancement du système jusqu'à son arrêt.

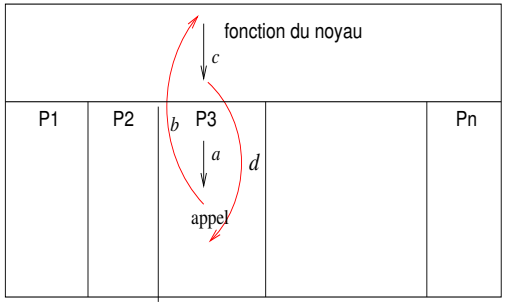
Certaines parties doivent obligatoirement rester en mémoire, sous peine de casse assurée ; d'autres peuvent temporairement être mises à l'écart, sur disque, puis ramenées en mémoire si nécessaire.

Exemples : la gestion de la mémoire centrale ne peut être délocalisée sur disque, sinon, on ne saurait ni comment lui reallouer de l'espace, ni où la réinstaller... Par contre, la gestion des files d'attente d'impression peut être absente temoprairement de la mémoire.

Discussion noyau modulaire ou noyau monolithique ?

Appel système

Un *appel système* est la demande d'exécution d'une fonction du noyau faite par un processus quelconque. On parle aussi d'*appel noyau*.



Noyau du système : fonctions, gérants

Exécution en mode noyau

Processus Utlisateurs

Exécution en mode utilisateur

Appels système sous Unix

Exemples :

- générer un processus (*fork()*) ou le terminer (*exit()*) ;
- créer des répertoires, lire, écrire dans des fichiers (*mkdir(), read(), write()*),
- demander de l'espace mémoire (*brk(), sbrk()*) ...

Noter les parenthèses dans tous les exemples afin de ne pas confondre **commande Unix** et **fonction du noyau**.

Les commandes du système, une grande majorité des programmes, système ou utilisateurs, sont construits à partir des appels système. **Exemples** : la commande *mkdir* utilise l'appel système *mkdir()*, les fonctions d'entrées-sorties connues dans les langages évolués font appel à *read(), write()*, les demandes d'allocation de mémoire (*new, malloc()* font appel à *brk(), sbrk()*.

Appels système et manuel

Le manuel Unix regroupe l'ensemble des appels noyau dans le volume 2 du manuel.
Lorsqu'on les visualise, on les reconnaît au chiffre 2 inscrit entre parenthèses après le nom de l'appel, alors que les commandes habituelles se reconnaissent au chiffre 1.

Exemple : mkdir(1), touch(1), mkdir(2), open(2).

volume 3 : fonctions de bibliothèque

Plan

Introduction

Les circuits mémoires et de calcul (électroniques et magnétiques) permettent de stocker des données sous forme **binaire**.

bit abréviation de binary digit, le bit constitue la plus petite unité d'information et vaut soit 0, soit 1.

mot séquence de bits numérotés de la façon suivante :

```
bn-1 bn-2 ... b2 b1 b0
 1    0   ... 0  1  1
```

quartet $n = 4$, **octets** $n = 8$ (*byte*), **mots** de n bits (*word*).

Poids fort, unités

La longueur des mots étant paire (n=2p), on parle de demi-mot de **poids fort** (ou le plus significatif) pour les p bits de gauche et de demi-mot de **poids faible** (ou le moins significatif) pour les p bits de droite.

Unités multiples

1 Kilo-octet = 2^{10} octets = 1024 octets noté 1 Ko
1 Méga-octet = 2^{20} octets = 1 048 576 octets noté 1 Mo
1 Giga-octet = 2^{30} octets $\approx 10^9$ octets noté 1 Go
1 Téra-octet = 2^{40} octets $\approx 10^{12}$ octets noté 1 To

Plan

Représentation des entiers positifs

Représentation en base 2 Un mot de n bits permet de représenter 2^n configurations différentes. Ces 2^n configurations sont associées aux entiers positifs compris dans l'intervalle $[0, 2^{n-1}]$ de la façon suivante :

$x = b_{n-1} * 2^{n-1} + b_{n-2} * 2^{n-2} + ... + b_1 * 2 + b_0$

Exemples

```
00...0 représente 0
0000 0111 représente 7 (4+2+1)
0110 0000 représente 96 (64+32)
1111 1110 représente 254 (128+64+32+16+8+4+2)
0000 0001 0000 0001 représente 257 (256+1)
```

100...00 représente 2^{n-1}
111...11 représente $2^n - 1$
Par la suite, cette convention sera notée Représentation Binaire Non Signée (RBNS).

Représentation en base 2^p

Plus compact en base 2^p : découper le mot $b_{n-1}b_{n-2}...b_0$ en tranches de p bits à partir de la droite (la dernière tranche est complétée par des 0 à gauche si n n'est pas multiple de p). Chacune des tranches obtenues est la représentation en base 2 d'un chiffre de x représenté en base 2^p.
p=3 (représentation **octale**) ou p=4 (représentation **hexadécimale**).
En représentation hexadécimale, les valeurs 10 à 15 sont représentées par les symboles A à F. On préfixe le nombre octal par 0, le nombre hexa par `0x`.

Exemples

```
x=200 et n=8
en binaire :     11 001 000 (128+64+8) : 200
en octal :        3   1   0 (3*64+8)   : 0310
en hexadécimal :     C   8 (12*16+8)   : 0xC8
```


Opérations

Addition binaire sur n bits Ajouter successivement de droite à gauche les bits de même poids des deux mots ainsi que la retenue éventuelle de l'addition précédente. En RBNS, la dernière retenue ou report (**carry**), représente le coefficient de poids 2^n et est donc synonyme de **dépassement de capacité**. Cet indicateur de Carry (Carry Flag) est situé dans le registre d'état du processeur.

exemple sur 8 bits

```
    1         1  1  1  1
    1  1  1   0  0  1  0  1        0xE5
+   1  0  0   0  1  0  1  1    +   0x8B
-----------------------------    --------
(1) 0  1  1   1  0  0  0  0        0x170      368 > 255
```

Les autres opérations dépendent de la représentation des entiers négatifs.

Le complément à 1 (C1)

Les entiers positifs sont en RBNS. Les entiers négatifs -|x| sont obtenus par inversion des bits de la RBNS de |x|. Le bit de poids n-1 indique le signe (0 positif, 1 négatif).
Intervalle de définition : $[-2^{n-1} + 1, 2^{n-1} - 1]$

Exemples sur un octet

```
3     0000 0011     -3    1111 1100
127   0111 1111     -127  1000 0000
0     0000 0000     0     1111 1111
```

Inconvénients :

- 2 représentations distinctes de 0 ;
- opérations arithmétiques peu aisées : 3 + -3 = 0 (1111 1111) mais 4 + -3 = 0 (00...0) !

Le second problème est résolu si l'on ajoute 1 lorsqu'on additionne un positif et un négatif : 3+1+ -3=0 (00...0) et 4 +1+ -3=1 (00...01)
D'où l'idée de la représentation en Complément à 2.

Le complément à 2 (C2)

Les entiers positifs sont en RBNS tandis que les négatifs sont obtenus par C1+1. Le bit de poids n-1 indique le signe (0 positif, 1 négatif). Une autre façon d'obtenir le C2 d'un entier relatif x consiste à écrire la RBNS de la somme de x et de 2^n.
Intervalle de définition : $[-2^{n-1}, 2^{n-1} - 1]$
Exemples sur un octet [-128, +127] :

```
3     0000 0011    -3    1111 1101
127   0111 1111    -127  1000 0001
0     0000 0000    -128  1000 0000
```

Inconvénient :

- Intervalle des négatifs non symétrique des positifs ;
- Le C2 de -128 est -128 !

Le complément à 2 (C2) (suite)

Avantage fondamental :

- l'addition binaire fonctionne correctement !
  -3+3=0 : 1111 1101+0000 0011=(1) 0000 0000
  -3 + -3 = -6 : 1111 1101+1111 1101=(1) 1111 1010

Remarquons ici que le positionnement du Carry à 1 n'indique pas un dépassement de capacité !
Dépassement de capacité en C2 : pas le CF mais l'Overflow Flag (OF).

```
127+127=-2   : 0111 1111 + 0111 1111=(0) 1111 1110
-128+-128=0  : 1000 0000 + 1000 0000=(1) 0000 0000
-127+-128=1  : 1000 0001 + 1000 0000=(1) 0000 0001
```

$Overflow = Retenue_{n-1} \oplus Retenue_{n-2}$

L'excédent à $2^{n-1} - 1$

Tout nombre x est représenté par $x + 2^{n-1} - 1$ en RBNS. Attention, le bit de signe est inversé (0 négatif, 1 positif).
Intervalle de définition : $[-2^{n-1} + 1, 2^{n-1}]$
Exemples sur un octet :

```
3     1000 0010    -3    0111 1100
128   1111 1111    -127  0000 0000
0     0111 1111
```

Avantage :

- représentation uniforme des entiers relatifs ;

Inconvénients :

- représentation des positifs différente de la RBNS ;
- opérations arithmétiques à adapter !

Opérations en RBNS et C2

Le C2 étant la représentation la plus utilisée (`int`), nous allons étudier les opérations arithmétiques en RBNS (`unsigned int`) et en C2.
Addition en RBNS et en C2, l'addition binaire (ADD) donne un résultat cohérent s'il n'y a pas dépassement de capacité (CF en RBNS, OF en C2).
Soustraction La soustraction x-y peut être réalisée par inversion du signe de y (NEG) puis addition avec x (ADD). L'instruction de soustraction SUB est généralement câblée par le matériel.
Multiplication et Division La multiplication x*y peut être réalisée par y additions successives de x tandis que la division peut être obtenue par soustractions successives et incrémentation d'un compteur tant que le dividende est supérieur à 0 (pas efficace $O(2^n)$).
Cependant, la plupart des processeurs fournissent des instructions MUL et DIV efficaces en $O(n)$ par décalage et addition.
Exemples : $13 * 12 = 13 * 2^3 + 13 * 2^2 = 13 << 3 + 13 << 2$
$126/16 = 126/2^4 = 1216 >> 4$

Codage DCB

Décimal Codé Binaire DCB ce codage utilise un quartet pour chaque chiffre décimal. Les quartets de 0xA à 0xF ne sont donc pas utilisés. Chaque octet permet donc de stocker 100 combinaisons différentes représentant les entiers de 0 à 99.
Le codage DCB des nombres à virgule nécessite de coder :

- le signe ;
- la position de la virgule ;
- les quartets de chiffres.

Inconvénients

- format de longueur variable ;
- taille mémoire utilisée importante ;
- opérations arithmétique lentes : ajustements nécessaires ;
- décalage des nombres nécessaires avant opérations pour faire coincider la virgule.

Avantage Résultats absolument corrects : pas d'erreurs de troncatures ou de précision d'où son utilisation en comptabilité.

Virgule flottante

notation scientifique en virgule flottante : $x = m * b^e$

- m est la mantisse,
- b la base,
- e l'exposant.

Exemple : pi = $0,0314159 * 10^2 = 31,4159 * 10^{-1} = 3,14159 * 10^0$

Représentation **normalisée** : positionner un seul chiffre différent de 0 de la mantisse à gauche de la virgule . On obtient ainsi : $b^0 \leq m < b^1$.

Exemple de mantisse normalisée : $pi = 3,14159 * 10^0$

En binaire normalisé

Exemple : $7,25_{10} = 111,01_2 = 1,1101 * 2^2$
4+2+1+0,25=(1+0,5+0,25+0,0625)*4

Virgule Flottante binaire

Remarques :

- $2^0 = 1 \leq m < 2^1 = 2$
- Les puissances négatives de 2 sont : 0,5 ; 0,25 ; 0,125 ; 0,0625 ; 0,03125 ; 0,015625 ; 0,0078125 ; ...
- La plupart des nombres à partie décimale finie n'ont pas de représentation binaire finie : (0,1 ; 0,2 ; ...).
- Par contre, tous les nombres finis en virgule flottante en base 2 s'expriment de façon finie en décimal car 10 est multiple de 2.
- Réfléchir à la représentation en base 3 ...
- Cette représentation binaire en virgule flottante, quel que soit le nombre de bits de mantisse et d'exposant, ne fait qu'**approcher** la plupart des nombres décimaux.

Algorithme de conversion de la partie décimale On applique à la partie décimale des multiplications successives par 2, et on range, à chaque itération, la partie entière du produit dans le résultat.

Virgule Flottante en machine

Exemples de conversion 0,375*2=**0**,75*2=**1**,5 ; 0,5*2=**1**,0 soit 0,011
0,23*2=0,46*2=0,92*2=1,84*2=1,68*2=1,36*2=0,72*2=1,44*2=0,88 ...
0,23 sur 8 bits de mantisse : 0,00111010

Standardisation

- portabilité entre machines, langages ;
- reproductibilité des calculs ;
- communication de données via les réseaux ;
- représentation de nombres spéciaux (∞ NaN, ...) ;
- procédures d'arrondi ;

Norme IEEE-754 flottants en simple précision sur 32 bits (float)

- signe : 1 bit (0 : +, 1 : -) ;
- exposant : 8 bits en excédent 127 [-127, 128] ;
- mantisse : 23 bits en RBNS ; normalisé sans représentation du 1 de gauche ! La mantisse est arrondie !

Virgule Flottante simple précision

4 octets ordonnés : signe, exposant, mantisse.

Valeur décimale d'un float : $(-1)^s * 2^{e-127} * 1, m$

Exemples 33,0 = +100001,0 = $+1,000012^5$ représenté par : 0 1000 0100 0000 100... c'est-à-dire : 0x 42 04 00 00

-5,25 = -101,01 = $-1,01012^2$ représenté par : 1 1000 0001 0101 000... c'est-à-dire : 0x C0 A8 00 00

Nombres spéciaux

- 0 : e=0x0 et m=0x0 (s donne le signe) ;
- infini : e=0xFF et m=0x0 (s donne le signe) ;
- NaN (Not a Number) : e=0xFF et m qcq ;
- dénormalisés : e=0x0 et $m \neq$0x0 ; dans ce cas, il n'y a plus de bit caché : très petits nombres.

Intervalle : $]-2^{128}, 2^{128}[= [-3.4 * 10^{38}, 3.4 * 10^{38}]$ avec 7 chiffres décimaux **significatifs** (variant d'une unité)

Virgule Flottante (fin)

Remarques

- Il existe, entre deux nombres représentables, un intervalle de réels non exprimables. La taille de ces intervalles (pas) croît avec la valeur absolue.
- Ainsi l'**erreur relative** due à l'arrondi de représentation est approximativement la même pour les petits nombres et les grands !
- Le nombre de chiffres décimaux significatifs varie en fonction du nombre de bits de mantisse, tandis que l'étendue de l'intervalle représenté varie avec le nombre de bits d'exposant :

double précision sur 64 bits (double)

- 1 bit de signe ;
- 11 bits d'exposant ;
- 52 bits de mantisse (16 chiffres significatifs) ;

Plan

Représentation des caractères

Symboles

- alphabétiques,
- numériques,
- de ponctuation et autres (semi-graphiques, ctrl)

Utilisation

- entrées/sorties pour stockage et communication ;
- représentation interne des données des programmes ;

Code ou jeu de car. : Ensemble de caractères associés aux mots binaires les représentant. Historiquement, les codes avaient une taille fixe (7 ou 8 ou 16 bits).

- ASCII (7) : alphabet anglais ;
- ISO 8859-1 ou ISO Latin-1 (8) : code national français (é, à, ...) ;
- UniCode (16 puis 32) : codage universel (mandarin, cyrillique, ...) ;
- UTF8 : codage de longueur variable d'UniCode : 1 caractère codé sur 1 à 4 octets.

ASCII

American Standard Code for Information Interchange Ce très universel code à 7 bits fournit 128 caractères (0..127) divisé en 2 parties :

- 32 caractères de fonction (de contrôle) permettant de commander les périphériques (0..31) ;
- 96 caractères imprimables (32..127).

Codes de contrôle importants

```
0 Null   9 Horizontal Tabulation    10 Line Feed
13 Carriage Return
```

Codes imprimables importants

```
0x20 Espace            0x30-0x39 '0'-'9'
0x41-0x5A 'A'-'Z'      0x61-0x7A 'a'-'z'
```


Code ASCII

Hexa	MSD	0	1	2	3	4	5	6	7
LSD	Bin.	000	001	010	011	100	101	110	111
0	0000	NUL	DLE	espace	0	@	P	`	p
1	0001	SOH	DC1	!	1	A	Q	a	q
2	0010	STX	DC2	"	2	B	R	b	r
3	0011	ETX	DC3	#	3	C	S	c	s
4	0100	EOT	DC4	$	4	D	T	d	t
5	0101	ENQ	NAK	%	5	E	U	e	u
6	0110	ACK	SYN	&	6	F	V	f	v
7	0111	BEL	ETB	'	7	G	W	g	w
8	1000	BS	CAN	(	8	H	X	h	x
9	1001	HT	EM	)	9	I	Y	i	y
A	1010	LF	SUB	*	:	J	Z	j	z
B	1011	VT	ESC	+	;	K	[	k	{
C	1100	FF	FS	,	<	L	\	l	|
D	1101	CR	GS	-	=	M	]	m	}
E	1110	SO	RS	.	>	N	^	n	~
F	1111	SI	US	/	?	O	_	o	DEL

ISO 8859-1 et utf-8

MSD\ LSD	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
C	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
D	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	Œ	Ø	Ù	Ú	Û	Ü	Ý	Þ	SS
E	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
F	ð	ñ	ò	ó	ô	õ	ö	œ	ø	ù	ú	û	ü	ý	þ	ß

TABLE: ISO Latin-1

Représentation binaire UTF-8	Signification
0xxxxxxx	1 octet codant 1 à 7 bits
0011 0000	0x30='0' caractère zéro
110xxxxx 10xxxxxx	2 octets codant 8 à 11 bits
1100 0011 1010 1001	0xC3A9 caractère 'é'
1110xxxx 10xxxxxx 10xxxxxx	3 octets codant 12 à 16 bits
1110 0010 1000 0010 1010 1100	0xE282AC caractère euro €
11110xxx 10xxxxxx 10xxxxxx 10xxxxxx	4 octets codant 17 à 21 bits

TABLE: utf-8

Plan

Structuration

Les structures de données disponibles en MC utilisent 2 principes :

- la contiguïté (tableau, struct) ;
- le chaînage par les pointeurs (liste, arbre, graphe) ;

Les pointeurs peuvent pointer sur des données statiques, dynamiques (malloc) ou locales (pile). Attention, les Systèmes d'exploitation utilisent principalement la contiguïté (tables systèmes) pour son efficacité.

Exemples

- `int global; ... int* pglobal=&global;`
- `int* pi=(int*)malloc(sizeof(int));`
- `{int i=5; int *pi=&i;};`

Tableaux C

Tableau SD homogène permettant l'accès indexé grâce à la contigüité des cases et à leur taille identique.

Exemples

- `char *s="toto";` chaîne de 4 caractères stockés sur 5 octets (caractère nul `'\0'` terminateur) ;
- `char *s="é";` chaîne de 1 caractère stocké sur 2 (ISO Latin-1) ou 3 (utf-8) octets ;
- `int tab[]={1,2,3};` tableau de 3 entiers ;
- `int *t=(int *)malloc(10*sizeof(int));` tableau dynamique de 10 entiers non initialisés ;
- `t[0]=tab[2];` pointeur ≈ tableau
- Se souvenir de la taille d'un tableau (argc, argv) ou mettre une balise à la fin (env) ;

Struct et Union

Syntaxiquement semblable, ces 2 outils de structuration permettent :

- soit d'agglomérer des champs de différents types ;
- soit de représenter un parmi différents types ;

Exemple

```
struct {
  char *nom;          // nom du fichier
  char type;          // soit 'f', soit 'd'
  union {
    char **liste;     // ls
    char *contenu;    // cat
  } choix ;
} x ;                 // une variable
```

Si x est un fichier (x.type=='f'), x.choix.contenu contiendra une chaîne décrivant son contenu, sinon si f est un répertoire (x.type=='d'), x.choix.liste contiendra la liste du répertoire.

Plan

Rôle et Organisation

La composante du système s'occupant de la *Gestion des Entrées-Sorties* (**I/O management**) est chargée de la communication avec les périphériques.

L'interface qu'offre le système d'exploitation pour l'accès aux périphériques cherche à uniformiser, à **banaliser** l'accès, c'est-à-dire à rendre la syntaxe de l'appel système indépendant du périphérique.

Quand on écrit `read(descripteur, donnée,taille)`, on ne dit rien du périphérique concerné (disque, clavier, ...).

Le système d'exploitation fait la liaison entre cette demande et le périphérique grâce à la demande d'ouverture.

Du logiciel au matériel on trouve la liaison entre appels système, pilotes et contrôleurs.

Pilote, Contrôleur, etc.

Plusieurs couches interviennent dans la communication avec le périphérique :

1. L'*appel système* lui-même détermine en fonction de l'entrée-sortie demandée le type de support (faut-il transférer un bloc ? une ligne ?, ...) et par conséquent le périphérique concerné.
2. Le *pilote*, logiciel du système d'exploitation, établit la liaison entre le système d'exploitation et les actions à demander au matériel du périphérique.
3. Le *contrôleur* est une partie intégrante du matériel indépendante de l'UC. Son rôle est de recevoir les demandes du pilote et de les réaliser.

On rappelle que les données sont transférées entre la mémoire et les périphériques par l'intermédiare de *bus*, ensemble matériel de fils **et** un protocole. Le protocole gère l'exclusivité d'accès et permet à plusieurs périphériques d'utiliser le même bus.

Exemple

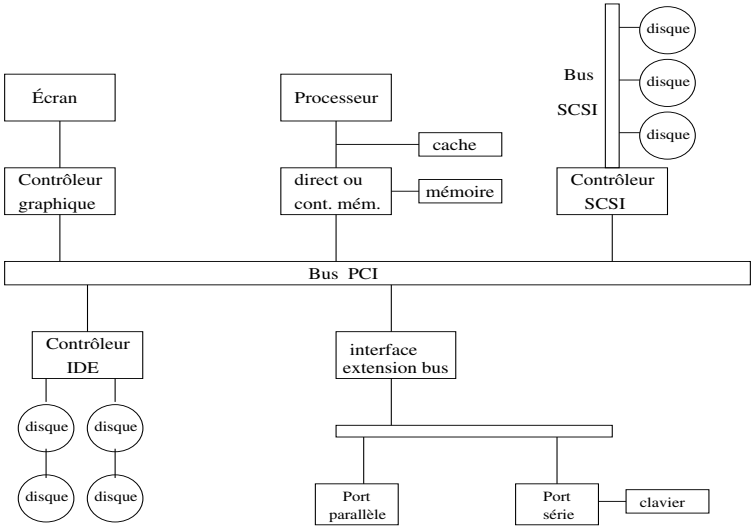
Un processus demande une lecture sur un descripteur de fichier.

- l'appel système passe en mode noyau et vérifie si on peut satisfaire directement, sans accès au périphérique, la demande. Si oui, la demande est satisfaite et le retour au mode utilisateur du processus immédiat. Sinon, on détermine de quel périphérique il s'agit, ici disque, puis la demande est expédiée au pilote concerné.
- Le pilote calcule et convertit la demande en adresse de disque et de secteur, puis invoque le contrôleur. Le processus est alors bloqué en attendant l'interruption, sauf cas spécifique d'attente active.
- Le contrôleur reçoit un numéro de secteur et une instruction (lecture, écriture) ; il recopie le contenu du secteur dans un tampon système en MC. Puis, il lance une interruption de fin d'E/S.

Exemple - suite

- Le gérant d'interruption réveille le processus demandeur qui exécute le pilote.
- Le pilote recopie une partie du tampon système vers l'espace utilisateur (adresse de la donnée).
- Puis, le pilote retourne un résultat à l'appel noyau permettant au processus de repasser en mode utilisateur.

Exemple - fin : Structure Classique



Plan

Bloc ou Caractère

La disctinction majeure en termes de types de périphériques provient de l'unité de transfert :

- **caractère** pour les périphériques dont l'unité est un caractère ou une suite de caractères de taille non déterminée à l'avance ; par exemple le clavier.
- **bloc** pour les périphériques dont l'unité de transfert est un ensemble de taille fixe par catégorie de périphérique ; disque par exemple.

Sous Unix, l'ensemble des périphériques est représenté dans le répertoire `/dev`. On peut y voir que les types de fichiers sont représentés par les lettres b pour bloc et c pour caractère.

Constatons qu'on vient d'enrichir les types de fichiers connus : on avait juque là les fichiers réguliers (-), les répertoires (d), les liens symboliques (l), les tubes (p) et on ajoute les deux nouveaux.

Plan

Contrôleur

Un contrôleur

- reçoit une commande, lire ou écrire, et une donnée dans un tampon local,
- il met en route le matériel si besoin (contrôle de la rotation du disque par exemple), réalise la commande , vérifie qu'elle est faite (test de relecture après écriture),
- avertit qu'elle est faite (interruption).

Il existe des contrôleurs très simples, contrôleurs de hauts-parleurs par exemple et très complexes, contrôleurs disque SCSI, qui disposent de leur propre processeur et sont capables de gérer plusieurs disques simultanément.

Pilote

Un pilote transcrit la demande de l'utilisateur en demande compréhensible par un contrôleur déterminé.

De plus en plus fréquemment, il n'est plus nécessaire de recompiler le noyau du système d'exploitation lorsqu'un nouveau pilote est mise en place pour un nouveau matériel.
Il reste parfois nécessaire de *retoucher* la configuration, ou d'ajouter le nouveau pilote s'il était absent.
Ceci est dû à la création de pilotes génériques permettant de faire la liaison dynamiquement entre un pilote et le système.

Extension : On peut gérer sous forme de pseudo-périphériques des matériels divers : la mémoire, le noyau du système, disques virtuels...

Exemple - Configuration du Clavier

Le fonctionnement classique, pour toutes les lectures au clavier, consiste à rentrer une suite de caractères et la valider par la touche *entrée*.

Ce fonctionnement est dit **bloquant, canonique**.

bloquant car l'instruction de lecture ne sera débloquée que lorsque la donnée sera disponible en mémoire,

canonique car la chaîne saisie est validée par *entrée*, et avant cette validation, on peut effacer, revenir sur ce qui est déjà frappé autant que nécessaire.

Le mode **canonique** est spécifique du clavier alors que le mode **bloquant** s'applique à tout fichier.

Modification du Comportement

On peut modifier ce comportement bloquant canonique :

- L'appel système *fcntl()* permet de modifier le comportement habituel de tout fichier, en particulier de basculer les comportements bloquant et non-bloquant.
- Les fonctions *tcgetattr()* et *tcsetattr()* permettent de modifier **tous** les paramètres de gestion du clavier.

Attention : passer au comportement non-canonique, implique que toutes les touches saisies sont validées immédiatement, sans possibilité de correction ! Par exemple, la touche d'effacement devient un caractère normal (0x08).

Plan

Flots

Un flot d'E/S est une séquence de caractères que l'on peut :
- soit lire (read) ;
- soit écrire (write) ;
- soit lire et écrire ;

Tout processus démarre avec 3 flots ouverts :
- `stdin`, 0 le flot d'entrée standard ;
- `stdout`, 1 le flot de sortie standard ;
- `stderr`, 2 le flot de sortie d'erreur standard ;

Ouverture d'un flot

Un flot doit être ouvert (open) sur un fichier ou un périphérique afin que le système réserve des tampons en MC : `int open(char* path, int mode, int droits)`
- path : désigantion du fichier ou du périphérique ;
- mode : `O_RDONLY, O_WRONLY, O_RDWR, O_APPEND, O_CREAT` ;
- droits : seulement pour la création de fichier ;
- résultat : entier **descripteur** du flot ou -1 si erreur ;

Un **pointeur courant** est positionné en début de flot lors de l'ouverture (sauf pour APPEND), et il est automatiquement incrémenté au fur et à mesure des lectures ou écritures. Lors d'une lecture en fin de flot, le résultat indique une "fin de fichier" (End Of File).

Algorithmique

Afin de généraliser notre discours à d'autres systèmes qu'Unix, on utilisera une notation algorithmique afin de décrire des mécanismes systèmes puis on traduira en C sur Unix.

comptage des octets d'un fichier

Données : chemin chaîne désignant le fichier
Résultat : entier nombre d'octets
Fonction `compte(`*chemin*`)` : entier ;
entier nb=0,fd;
fd=ouvrir(chemin, LECTURE);
tant que *EOF != lireCar(fd)* **faire**
  nb++;
fermer(fd);
return *nb*;

Traduction en C : compte.c

```
int compte(char *chemin){
  int nb=0,fd;
  fd=open(chemin,O_RDONLY);
  if (fd<0){
    fprintf(stderr,"Impossible d'ouvrir le fichier %s !\n",chemin);
    return 2;
  }
  char c; /* tampon de lecture */
  while(1==read(fd,&c,1)){
    nb++;
  }
  close(fd);
  return nb;
}
int main(int n, char *argv[], char *env[]){
  if (n!=2){
    fprintf(stderr,"Syntaxe : %s chemin !\n", argv[0]);
    return 1;
  }
  printf("%s contient %d octets !\n",argv[1],compte(argv[1]));
  return 0;
}
```


Accès direct

L'intérêt de l'**accès séquentiel** est son universalité : quel que soit le périphérique de mémorisation ou de communication, ce mode d'accès est possible.
Son inconvénient est sa lenteur dans la recherche d'une information particulière : il faut en moyenne lire la moitié du fichier pour trouver l'information recherchée.

Pour les fichiers sur dique, l'**accès direct** permet de rechercher un article parmi d'autres grâce à une **expression d'accès** (parfois nommée clé). Par exemple, le numéro d'étudiant (unique), ou le nom (homonymes) d'un étudiant sont des expressions d'accès.
L'appel système lseek permet de se déplacer dans un fichier ouvert :

```
off_t lseek(int fildes, off_t offset, int whence);
whence : SEEK_SET|SEEK_CUR|SEEK_END
```

Voir aussi `fseek() (3)`

Plan

schéma général système

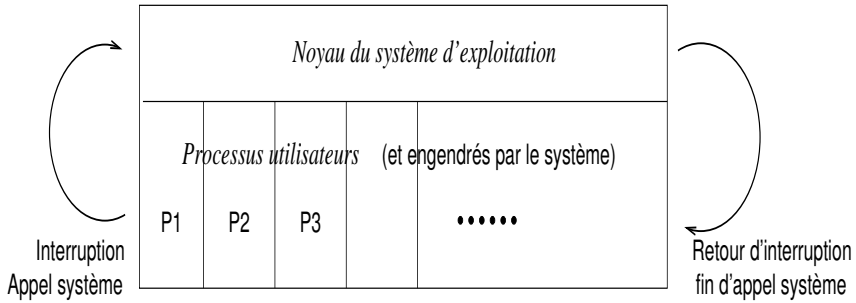


Table des processus

La table des processus contient, par processus, un ensemble d'informations relatives à chaque composante du système. Un tout petit extrait :

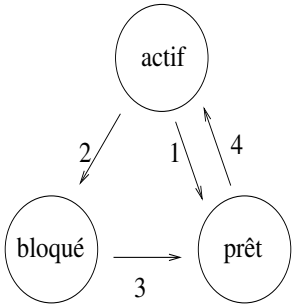
G. processus	G. mémoire	G. fichiers
CO, PP, PSW	ptrs segments	descripteurs
Temps UC	gest. signaux	masque
identités	esp. virtuel	répertoire travail
état	processus liés	propriété

Attention : Le système gère beaucoup d'autres tables : table des fichiers ouverts, de l'occupation mémoire, des utilisateurs connectés, des files d'attente, etc.

Plan

États d'un processus

On commence par les états de base :



actif tient la ressource UC

prêt seule la ressource UC lui manque

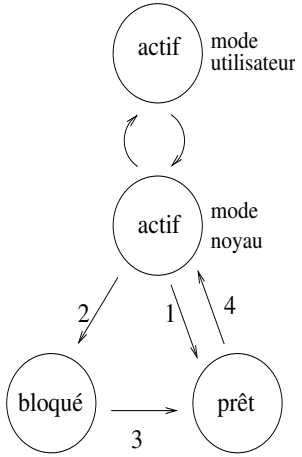
bloqué manque au moins une autre ressource

Important : le passage à l'état actif ne peut se faire que par l'état prêt.

Exercice : citer un exemple pour chaque cas de changement d'état.

Un peu plus

On complète (un peu) ces états de base :



Déjà vu : les appels système, les gérants d'interruption font passer du mode utilisateur au mode noyau ; les retours de ces appels font le passage inverse. Compléments plus loin.

En dehors des changements entre modes *actif utilisateur* et *actif noyau*, tous les autres changements d'état ne peuvent se produire qu'en mode noyau. Heureusement...
question : pourquoi ?

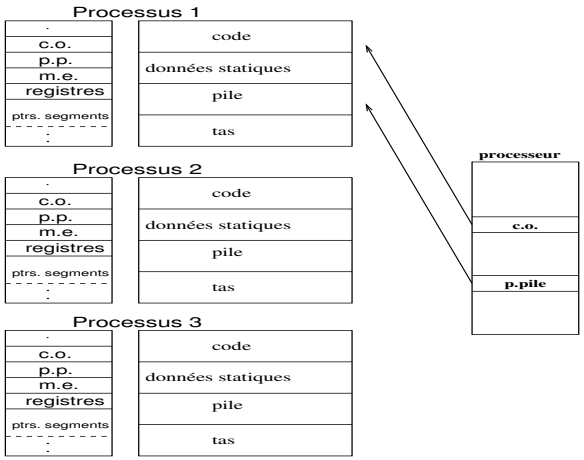
Plan

Principe du changement de contexte

- **le *système* s'exécute dans le contexte du processus actif** ;
- on reconnait le processus actif car c'est le seul processus vers qui pointent le CO et le pointeur de pile -SP- du processeur (système mono-processeur) ; il est aussi désigné par l'état *actif* dans la table des processus ;
- si ce processus doit s'interrompre temporairement, il faut sauvegarder tout les éléments qui risquent de disparaître et les restituer lorsqu'il pourra continuer.

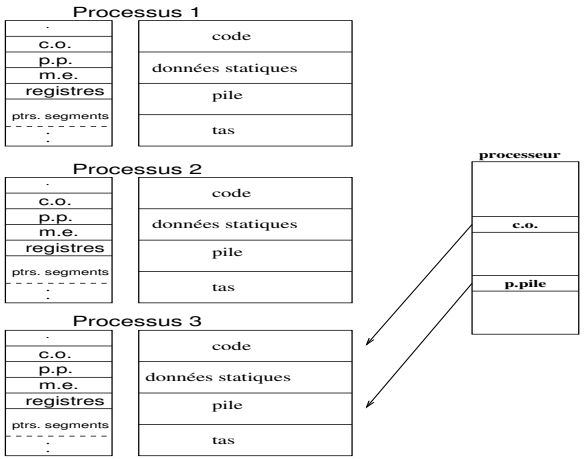
On dit que le système effectue un *changement de contexte*, ou un *basculement de contexte* (*context switch*).
Ce changement se produit lorsque le processus actif passe à l'état bloqué ou prêt et qu'un autre processus devient actif.

Un processus actif



Un autre processus actif

Sauvegarde du contexte de P1, restauration du contexte de P3.



Réalisation des changements de contextes

Lorsque le processus actif passe au mode noyau, il y aura exécution d'une fonction du noyau : soit un gérant d'interruption, soit la fonction appelée directement par ce processus.
la fonction du noyau appelée s'exécute dans le contexte du processus actif courant. Cette exécution va invariablement finir par l'appel à l'**ordonnanceur** (*scheduler*).

Déroulement d'un appel noyau

On peut décrire le déroulement d'un appel noyau :

1. passage du processus actif en mode noyau ;
2. exécution de l'appel noyau (appelé aussi *routine*) ;
3. cet appel modifie l'état du processus actif, sauvegarde son contexte ;
4. appel de l'ordonnanceur qui procède à l'élection ; **remarque** : la fonction noyau appelée n'est pas terminée ;
5. l'ordonnanceur restaure le contexte du processus élu et le marque *actif* ;
6. fin de l'ordonnanceur (*return*) dans le contexte du nouvel élu ;
7. fin de l'appel ayant provoqué précédemment la suspension de l'élu ;
8. passage de l'élu en mode utilisateur et suite de son exécution.

Interrogation orale

Questions :

1. Expliquer pourquoi la fonction noyau appelée n'est pas terminée ;
2. Pour tous les processus en attente, c'est-à-dire tous sauf le processus actif, quel est l'état de leur pile ? quelle est la dernière fonction empilée ?
3. Dans quel mode d'exécution se trouvent tous les processus en attente ?

Rôle de l'ordonnanceur

Règles à respecter :

- élire parmi les processus **prêts** celui qui deviendra actif ;
- l'élu est celui de plus haute priorité compte tenu de la **politique** d'allocation de l'UC du système (vaste programme...) ;
- effectuer un changement de contexte : sauvegarder celui du processus courant et restituer celui de l'élu.

Schéma algorithmique ordonnanceur

tant que *pas de processus élu* **faire**
    consulter table processus ;
    chosir celui de plus haut priorité parmi les prêts;
    si *pas d'élu* **alors**
        attendre ;
        //jusqu'à nouvelle interruption (processeur à l'état *latent*)
    marquer ce processus actif ;
    basculer le contexte ;
    return ;
    //le processus continue son exécution

Plan

Scénario - Étape Initiale

On suppose trois processus, P1, P2 et P3, tels que P1 est *actif*, P2 et P3 sont dans l'état *prêt*

P1 possède donc la ressource UC. On suppose qu'il ne consomme pas entièrement son quantum de temps, car il fait une demande de lecture d'une donnée sur disque.
Les étapes suivantes vont se dérouler :

1. P1 passe en mode privilégié en faisant l'appel système *read()* ;
2. l'exécution de *read()* va commencer, puis lancer la demande de lecture physique qui sera prise en charge par une entité extérieure dépendant du périphérique concerné (contrôleur disque, contrôleur clavier, . . .) ;
3. l'état de P1 sera passé à *bloqué* et son contexte sauvegardé ;
4. enfin, *read* va appeler l'ordonnaceur.

Scénario - Étape 2

Choix possibles pour l'ordonnanceur : P2 ou P3 ; on suppose que c'est P2 qui est élu. Les étapes suivantes sont :

1. l'état de P2 est passé à *actif* ; on rappelle qu'il est en mode privilégié d'exécution ;
2. le contexte de P2 est restauré ;
3. l'horloge programmabe allouant les quantum de temps est réinitialisée à la valeur fixée dans le système ;
4. fin de l'ordonnaceur : le CO est restitué à partir de la pile de P2 (adresse de retour) ;
5. fin de l'appel noyau ou de l'interruption qui avait provoqué l'arrêt précédent de P2 et P2 repasse alors en mode utilisateur.

Suite Étape 2

6. suite de l'exécution de P2 ; on suppose que P2 ne fait aucune opération d'entrée-sortie et qu'aucun événement ne vient le perturber ; P2 consomme ainsi entièrement son quantum de temps ;
7. il y a interruption d'horloge ;
8. passage en mode noyau et exécution du gérant d'interruption d'horloge qui passe P2 à l'état prêt ;
9. sauvegarde du contexte de P2 par le gérant ;
10. fin du gérant (presque) : appel ordonnanceur.

Remarque : On dit dans cette situation que P2 a été *préempté* et que le sytème d'exploitation qui opère fait de la *préemption*.

Scénario - Étape 3

Choix possibles pour l'ordonnanceur : P2 ou P3 ; on suppose P3 élu ; **rappel rapide** : P3 passe à l'état actif, son contexte est restauré, il est en mode noyau et l'horloge est reinitialisée. Suite du scénario :

1. P3 est en cours d'exécution ; on suppose que la lecture demandée par P1 est (enfin) prête ; alors,
2. P3 est interrompu par une interruption disque ;
3. il y a passage en mode noyau et exécution du gérant d'interruption disque ; la donnée lue est donc disponible en mémoire, dans un espace tampon du système ; d'autres situations sont possibles ici, selon la gestion des transferts entre disques et mémoire centrale, mais le principe de l'interruption reste ;
4. P1 est passé à l'état *prêt* (on dit que P1 est *réveillé*) ;
5. P3 est **aussi** passé à l'état *prêt* ;
6. après la sauvegarde du contexte de P3, appel de l'ordonnanceur.

Remarques

Noter l'exécution de la routine d'interruption disque sur le compte et dans le contexte de P3 qui n'est pas concerné et se voit interrompu et délogé.

Noter aussi l'instant où se produit l'interruption lors d'une entrée-sortie :
en **entrée** lorsque la donnée (le secteur lu, la ligne entrée par l'utilisateur, le clic souris) est disponible en mémoire,
en **sortie** lorsque la donnée est transférée de l'espace du processus vers le tampon système (on peut modifier son contenu dans l'espace du processus)

Scénario - Étape 4

Choix possibles pour l'ordonnanceur : P1, P2 ou P3.
On suppose que P1 est élu ; pour sa mise en place, voir le rappel rapide ci-avant. Déroulement de la suite :

1. *read()* continue son exécution pour P1 et amène le contenu du tampon disque dans la mémoire de P1 ; **exemple** : si P1 a fait *read(monfich,&erlude,sizeof(int))* la donnée *erlude* sera remplie à partir du tampon disque ;
2. fin d'exécution de *read()* entraînant le passage de P1 en mode utilisateur ;
3. on suppose que P1 continue en faisant quelques instructions, puis se termine ; il fait donc appel à *exit()*, qui est un appel noyau ;

Suite Étape 4

4. passage en mode noyau et réalisation de *exit()* ; un ensemble d'opérations de nettoyage est lancé : appel de destructeurs éventuels, fermeture des fichiers encore ouverts, opérations comptables, restitution de l'espace mémoire occupé par P1, signalement de sa fin à ses descendants (voir plus loin, la descendance des processus), nettoyage de l'entrée P1 dans la table des processus, . . .
5. appel ordonnanceur : P2 et P3 sont prêts.

D'autres soucis ?

Il est temps d'ajouter quelques nouveautés : des problèmes non encore traités.

- Comment se fait la génération des processus ? Voir paragraphe suivant.

ou des questions :

- Que se passe-t-il s'il n'y a aucun processus prêt ? Voir *état latent* du processeur dans la bibliographie,
- Quel est le lien entre les appels *kill()* et *exit()* ? Voir la communication entre processus plus loin.

Plan

Génération de processus

Objectif : obtenir un nouveau processus à l'état *prêt*. Il faut :

- vérifier l'existence de l'exécutable,
- réserver un élément dans la table des processus,
- réserver l'espace nécessaire en mémoire,
- charger le code et données statiques dans les segments correspondants,
- initialiser les divers éléments des tables du système,
- mettre en place les fichiers ouverts par défaut,
- initialiser le contexte (compteur ordinal, pointeur pile en particulier).

Important : noter que c'est forcément un processus (le processus actif) qui demande cette création !

Sous Unix

Sous Unix, deux phases distinctes :

- mise en place d'un clône, par une copie de l'ensemble des segments du processus demandeur,
- mise en place de nouveaux segments de code et données statiques, réinitialisation de la pile et du tas, **si nécessaire**.

Le clône est réalisé par l'appel noyau *fork()* ; le remplacement des segments par *execve()* (cet appel est décliné en plusieurs variantes).

Exemple : on lance dans une fenêtre de l'interprète de langage de commande (le *shell*) ***belotte***.
Pour le réaliser, l'interprète se duplique d'abord. Il y a donc un deuxième processus interprète et dans ce deuxième il y a appel à *execve()* afin de charger le code de ***belotte*** à la place du celui de l'interprète.

Principe de fonctionnement de *fork()*

- Créer une copie des segments de l'appelant ;
- chacun des deux processus aura donc le même code exécutable et continuera son exécution indépendamment de l'autre ;
- permettre au parent de reconnaître l'enfant créé parmi tous ceux qu'il a créés en lui restituant le numéro du nouvellement créé.

On peut noter que l'enfant aura un moyen de reconnaître son générateur ; en effet, si un parent peut avoir plusieurs enfants, un enfant ne peut avoir qu'un seul parent. Quoique, en cas de décès prématuré du générateur...

Schéma algorithmique fork()

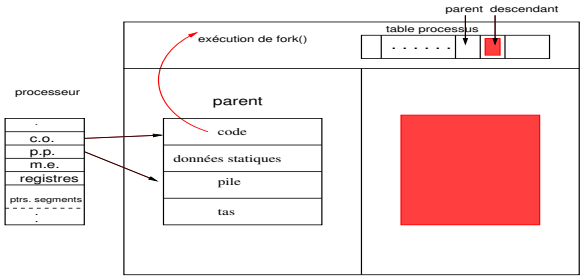
```
//résultat : dans parent : numéro de l'enfant ; dans enfant : 0
si (ressources système non disponibles) alors
    retourner erreur ; exit(0) ;
créer nouvel élément dans table processus ;
obtenir nouveau numéro processus ;
marquer état de ce processus en cours création ;
initialiser table processus[enfant] ;
copier segments de l'appelant dans l'espace mémoire du nouveau ;
incrémenter décompte fichiers ouverts ;
marquer état enfant prêt ;
si (processus en cours est le parent) alors
    retourner numéro enfant ;
sinon
    retourner 0 ;
```


Exemple *fork()*

```
pid_t louche = fork () ;
switch (louche)
    case (-1 :)
        //erreur à la génération
        cout«"erreur système en votre faveur"«endl ;
        break ;
    case (0 :)
        //partie exécutée dans l'enfant
        cout«"un vieux me ressemblant ? impossible !"«endl ;
        break ;
    default
        :
    //partie exécutée dans le parent
    cout «"qui suis-je ? où cours-je ?"«endl ;
```

Déroulement

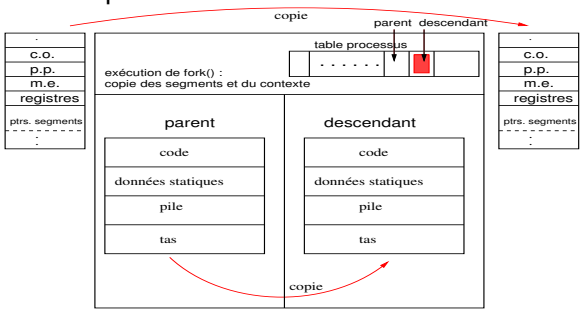
Le processus exécute ce code ; il y a appel noyau : *fork()*.



Il y a vérification de disponibilité des ressources : dans la table des processus, dans l'espace mémoire, etc, puis réservation d'espace pour l'enfant.

Suite et fin Déroulement

Cet appel fait une copie :



Avant la fin de *fork()* deux processus existent et tous les deux vont exécuter la fin de l'appel noyau, chacun dans son contexte.
La pile de chacun contient un résultat différent de l'exécution de *fork()*.
Donc chacun déroulera un cas différent de l'exécution du même code.

Questions et Exercices

1. En reprenant l'exemple précédent, où vont avoir lieu les affichages respectifs ?
2. Dans le schéma précédent représentant la suite et fin de déroulement, dans quelle partie de la mémoire sont localisées les deux « *oreilles* » représentant le contexte d'exécution de chaque processus ?
3. Peut-on prévoir l'ordre dans lequel les deux processus vont s'exécuter ? Justifier. Autrement dit, dans l'exemple peut-on dire dans quel ordre s'afficheront les lignes écrites par les processus ?
4. Donner deux exemples dans lesquels on aura un résutlat négatif à *fork()*.

Plan

Unix : Recouvrement de Processus

Le recouvrement consiste à demander dans un processus l'exécution d'un autre code exécutable que celui en cours d'exécution.

Principe :

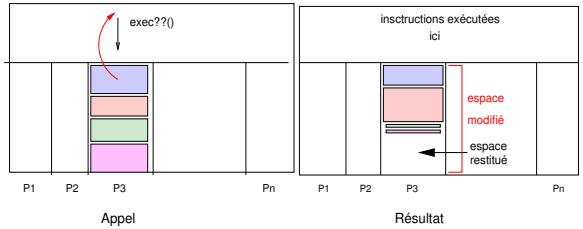
- vérifier l'existence et l'accessibilité (droits) du fichier exécutable ;
- écraser son propre segment de code par le nouvel exécutable ;
- passer éventuellement à ce code des paramètres déxécution ;
- générer un nouveau segment de données statiques ;
- vider le segment de pile ;
- vider le tas ;
- faire quelques modifications dans la table des processus (espace mémoire alloué, compteur ordinal, etc).

Déroulement

Difficulté : se rendre compte qu'on fait de l'auto-destruction sans risque, car

- les instructions exécutées sont dans le noyau ; on ne risque pas de les écraser ;
- la partie écrasée est dans une autre partie de l'espace mémoire et les données ne sont pas utiles ;

C'est donc une copie disque → mémoire qui est faite, avec une réinitialisation ou rechargement des segments de données.



Recouvrement - Syntaxe

Il y a un et un seul appel noyau, `execve()`, décliné en plusieurs formes <<confortables>>, plus faciles à appréhender.

Deux formes simples :

```
#include <unistd.h>
extern char **environ;

int execl(const char *path, const char *arg, ...)
int execlp(const char *file, const char *arg, ...)
```


Exemples

Exemple 1 : Un processus `aloeil` dérive d'un programme contenant l'instruction

```
int rigue=execl("/auto_home/amoi/bin/oculaire",
        "oculaire","prise","pousse","garde",NULL);
```

- L'exécution de `aloeil` va provoquer, si les ressources sont disponibles, le lancement de l'exécutable `oculaire` **à la place de `aloeil`**.
- `oculaire` recevra les paramètres indiqués, `oculaire` en position 0, `prise` en position 1, `pousse` en position 2, `garde` en position 3.

Exemples - suite

Exemple 2 : on suppose que la ligne d'appel devient

```
int rigue=execlp("oculaire","oculaire",
"prise","pousse","garde",NULL);
```

- Ici, le fichier exécutable `oculaire` va être recherché dans l'ensemble des répertoires cités dans la variable d'environnement `PATH`.
- Si le fichier n'est pas trouvé, `execlp` rend un résultat négatif.
- Noter que seul un défaut de terminaison de `exec()` engendre un résultat retourné ; autrement dit, pas de résultat positif à attendre.

Exercices

1. Trouver deux cas d'erreurs possibles.
2. Est-ce qu'un nombre de paramètres incohérent entre ceux passés et ceux attendus par l'exécutable est un cas d'erreur de `exec()` ?
3. Comment faire pour que ses propres programmes soient pris en compte, par `execlp()` (deux solutions) ?

Unix : Suite recouvrement

L'**appel noyau** :

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

Rapprocher cette syntaxe de celle de `main()`. Il est possible de passer un environnement.

Autres formes :
Voir le manuel pour les détails, `int execle()`, `int execv()`, `int execvp()`

Résumé :

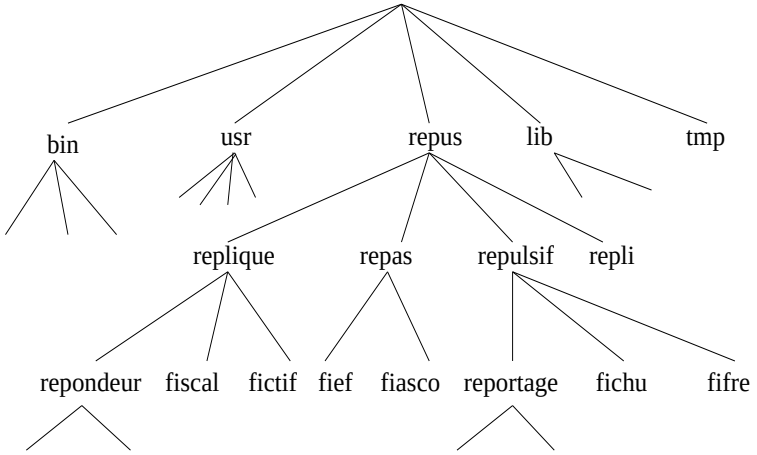
- **l** liste explicite des paramètres,
- **v** liste des paramètres selon pointeur (`char *const argv[]`)
- **p** chemin exécutable selon `PATH`,
- **e** environnement à passer selon pointeur (`char *const envp[]`)

Plan

Les Problèmes

- Gérer l'espace disque, répondre aux demandes d'allocations et de libération d'espace ;
- Donner à l'utilisateur une vision cohérente, indépendante du mode de gestion de l'espace.

Vision utilisateur



L'utilisateur face au système

L'utilisateur veut :

- des fichiers possédant (au moins un) nom,
- organisés de sorte à les retrouver <<facilement>>,
- sur lesquels il a le droit de propriété absolu et le droit de laisser faire les autres selon son gré,
- extensible l'infini ;
- que les modifications sur un fichier soient faites sur toutes les copies du même fichier (vaste programme) ;
- et d'autres caractéristiques qu'il faudra ajouter à ce document.

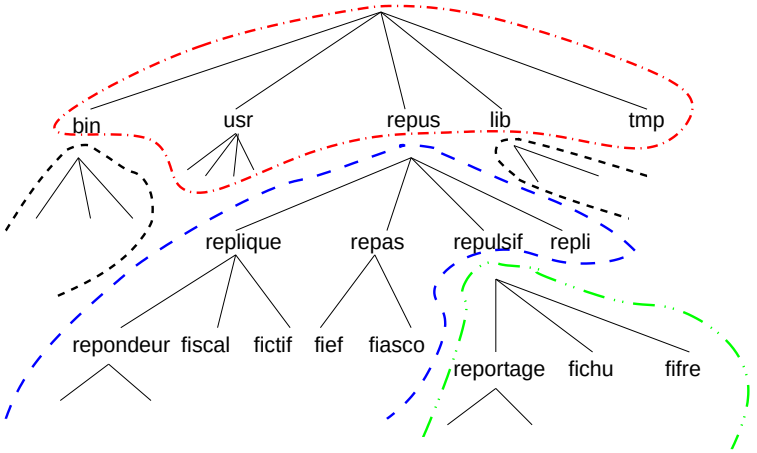
Réponse du système

Le système répond qu'il propose ce qu'il a de mieux, mais que :

- les noms des fichiers sont insuffisants car pas discriminatoires,
- l'espace est forcément limité et il doit le gérer pour tous,
- qu'il y a plusieurs solutions pour masquer ou montrer des fichiers et qu'il ne peut appliquer une solution différente à chacun.

Plan

Partitions et sous-arbres



Système de fichiers *simple* ou Partition

Une partie de l'espace disque. En toute généralité, peut aller d'une partie d'un disque à plusieurs disques. Doit contenir tout le nécessaire pour la (bonne) tenue de l'espace.
Se compose de deux parties créées avant la première utilisation.

espace de gestion	allocation blocs
espace des données	contenus effectifs fichiers

Composantes d'une Partition

Espace de gestion allocation/récupération des **blocs**, unités de transmission périphérique ↔ mémoire centrale.
Comporte :
- une table des *inodes* (ou *i-nœuds*) : matricules des fichiers,
- un moyen de connaître les blocs libres (table complète, partielle, ...),
- éventuellement bloc de lancement (*bootstrap*).

Espace des données contient les ...contenus des fichiers et répertoires ; éventuellement quelques blocs <<volés>> par l'espace de gestion.

Plan

Table des inodes

num.	type	droits	liens	prop.	taille	dates	pointeurs

- **num.** numéro d'inode
- **type** type de l'inode : fichier, répertoire, autre (il y en a),
- **droits** droits habituels : utilisateur, groupe, autres,
- **prop.** identité (numéro) du propriétaire,
- **taille** taille du fichier en nombre d'octets,
- **pointeurs** numéros des blocs dans l'espace des données,
- **dates** il y en a plusieurs ; étudié plus loin,
- **liens** étudié plus loin.

Plan

Représentation Fichiers et Répertoires

num.	type	droits	liens	prop.	taille	dates	pointeurs
1450	-	rwxr-xr-x		470 ; 47001	125	*	8003475
795	d	rwxr-x--		470 ; 47001	2048	? ?	101472
2	d	rwxr-xr-x		0 ; 0	2048	? ?	10

Répertoires

repas

num.	nom
2	••
795	•
1450	fief

racine sous-arbre

num.	nom
0	••
2	•
795	repas
7654	repli

repus
autre partition

num.	nom

Plan

Extension d'un fichier

Remplir le dernier bloc ; demander une nouvelle allocation s'il est plein.

si *(demande écriture)* **alors**
- **si** *(dernier bloc plein)* **alors**
  - demander allocation un bloc ;
  - marquer adresse obtenue dans pointeurs table-inodes ;
- remplir dernier bloc ;
- modifier taille fichier dans table-inodes ;

Attention : Il n'y a aucune marque de fin de fichier dans le fichier. La seule information est la taille du fichier.
Remarque : Les blocs alloués à un même fichier ne sont pas contigus.
Exercice : écrire l'algorithme de réduction d'un fichier.

Droits - un début

Question : quels sont les droits nécessaires pour effectuer l'opération d'extension ?
Réponse :
- écrire sur le fichier, évidemment,
- pouvoir l'atteindre !

Atteindre un fichier c'est indiquer un chemin d'accès (absolu ou relatif) ; il faut pouvoir pacourir ce chemin, donc traverser les répertoires composant le chemin.

Droits sur répertoires :
- le droit **r** sur un répertoire permet de lire son contenu,
- le droit **x** permet de le traverser.

Suppression Fichier

Algorithme 1 : supprimer(fichier)

si *(droits de traversée jusqu'au répertoire contenant acquis **et** droits d'écriture dans répertoire contenant)* **alors**
- restituer espace désigné par pointeurs ;
- effacer entrée dans table-inodes ;
- effacer entrée dans répertoire ;

sinon
- afficher suppression impossible

Remarque : constater qu'aucun droit sur le fichier lui-même n'est nécessaire !
Question : analyser ce qui se passe dans `/tmp` et déduire que ce comportement est insuffisant. Y a-t-il une réponse ?

Copie de Fichiers

copier(source,destination)

si *(droits d'atteindre source* ***et*** *droits lecture fichier source* ***et*** *droits d'atteindre destination* ***et*** *droits d'écriture dans répertoire destination)*
alors
  si *(demande allocation blocs pour tout le fichier positive)* **alors**
    demander allocation d'un élément dans table-inodes ;
    créer cet inode (allouer numéro, marquer droits, proprio...) ;
    créer entrée dans répertoire de destination ;
    copier ;
  sinon
    afficher ("erreur allocation") ;
sinon
  afficher ("erreur accès") ;

Déplacer, Renommer

L'opération de déplacement inclut le renommage (voir la syntaxe de `mv` sous Unix).

- Elle peut se faire sur fichier ou répertoire.
- Si les éléments *source* et *destination* sont dans le même SGF simple, l'opération consiste uniquement à déplacer (ou renommer) une entrée du répertoire contenant la source vers le répertoire contenant la destination. Cette opération est donc nettement plus économique qu'une copie suivie d'une suppression.
- Lorsque source et destination se trouvent dans deux SGF simples différents et que le système d'exploitation accepte de faire l'opération, il réalise alors une copie suivie d'une suppression.

Déplacement - Algorithme

deplacer(source,destination)

si *((droits d'atteindre* ***et*** *droit d'écriture répertoire contenant source)* ***et*** *(droits d'atteindre* ***et*** *droit d'écriture répertoire contenant destination))*
alors
  si *(source et destination dans même SGF simple)* **alors**
    créer (nouveau nom, n^o inode source)
          dans répertoire contenant destination ;
    effacer entrée source du répertoire contenant ;
  sinon
    //Attention : l'opération ci-dessous peut concerner un répertoire
    copier (source, destination) ;
    effacer (source) ;

Opérations sur Répertoires

Un répertoire contient la table des correspondances n^o *inode* $\Leftrightarrow$ *nom*

La **longueur des noms des fichiers** est variable, avec une limite administrable, fixée par le système ; par exemple 256 octets.

Les éléments autres que le *numéro d'inode* et le *nom* inclus dans un répertoire permettent la **gestion de la table** ; par exemple, la longueur de l'élément courant.

Ceci explique pourquoi des données de l'inode comme la taille d'un répertoire est gérée autrement que celle d'un fichier.

Ceci explique aussi pourquoi les opérations sur les répertoires se font **indirectement**, à travers des appels système : on ne peut admettre que l'utilisateur puisse modifier directement (ouvrir, écrire) un répertoire ; le SGF pourrait être corrompu.

Voir plus loin dans ce chapitre les appels système liés au SGF.

Plan

Notion de lien

On peut voir une entrée dans un répertoire comme un *pointeur* sur un élément de la table des inodes.

Question : peut-on avoir plusieurs pointeurs sur le même inode ?

Réponse : pourquoi pas ? On étudie la réalisation et les conséquences de cette opération.

num.	type	droits	liens	prop.	taille	dates	pointeurs
1450	–	rwxr-xr-x	2	470 ; 47001	125	*	8003475

Répertoires

repas

num.	nom
1450	fief

replique

num.	nom
1450	fictif

Il faudra mémoriser dans l'inode le **nombre** de références.

Lien Dur

Une telle référence s'appelle *lien dur* ou *physique*, par opposition à un *lien symbolique* qu'on étudiera plus tard.

Attention : on peut dire que toute référence à un inode représentant un fichier est un *lien dur* ! En effet, une fois l'opération effectuée, on ne peut établir un ordre de précédence parmi les références.

Caractéristiques :

- Un lien dur est forcément dans une même partition (SGF simple) ; on dit qu'un lien dur ne peut *traverser* les partitions.
- Cette notion n'a aucun sens sur un répertoire - penser aux modifications dans les répertoires et voir la suite.
- Noter qu'il y a un seul inode, donc un seul propriétaire, un seul fichier, un seul ensemble de droits, . . . **MAIS** des utilisateurs différents, propriétaires de répertoires différents pourraient faire des liens sur un même inode, selon les droits du fichier.

Problèmes Induits par les Liens Durs

Une correction à faire : L'algorithme de suppression de fichier décrit précédemment devient faux.

En effet, la suppression d'une référence supprimera l'inode et tout le contenu du fichier. Il restera alors dans des répertoires des entrées référençant des objets non identifiés (pas des OVNI, à cause du V).

Principe de la solution :

- à chaque création d'un lien incrémenter le nombre de liens dans la table des inodes ;
- le décrémenter à chaque suppression ;
- ne supprimer l'inode et le fichier que lorsqu'il n'y a plus de références.

Nouvel algorithme de Suppression

Algorithme 2 : supprimer(fichier)

si *(droits de traversée jusqu'au répertoire contenant acquis* ***et*** *droits d'écriture dans répertoire contenant)* **alors**
- nombreDeLiens - - ;
- effacer entrée dans répertoire ;
- **si** *(nombreDeLiens == 0)* **alors**
  - restituer espace désigné par pointeurs ;
  - effacer entrée dans table-inodes ;

sinon
- afficher suppression impossible

Exemple de Suppression

num.	type	droits	liens	prop.	taille	dates	pointeurs
1450	–	rwxr-xr-x	2	470 ; 47001	125	*	8003475

Répertoires

repas

num.	nom
1450	fief

replique

num.	nom
1450	fictif

Avantages et Manques

Avantages
- inutile de faire des copies de fichiers, donc une seule version du fichier, donc mises à jour cohérentes,
- permet d'assurer la compatibilité entre emplacements différents prévus pour un fichier, par exemple entre versions d'un système d'exploitation (fichiers dans /bin, /usr/bin, . . .).

Manques
- la limite au SGF simple est très réductrice,
- pas de référence à un répertoire.

Remarque : la donnée *liens* dans la table des inodes est utilisée et a un sens différent pour les répertoires.

Plan

Un peu de Repos - Appels Système du SGF

On établit une liste non exhaustive des appels systèmes permettant d'accéder ou manipuler soit des *fichiers*, soit des *répertoires*, soit des *inodes*.

Accès à des fichiers

open(), close(), creat()	comme leur nom l'indique
read(), write()	comme leur nom l'indique
lseek()	déplacement avant/arrière
unlink()	supprimer (algorithme 2)
link()	créer un lien
rename()	renommer, déplacer (répertoires aussi)

Exemples : la commande `rm` fait appel à *unlink()*, la commande `ln`, peut faire appel à *link()*, selon les options passées.

Appels Système pour Répertoires

Noter que les commandes connues utlisent forcément ces appels.

rename()	renommer, déplacer
mkdir(), rmdir()	création suppression
opendir(), closedir()	ouverture, fermeture
readdir()	lecture d'une entrée (n^o inode, nom)
chdir()	parcourir, se déplacer

Remarque : pas de *writedir()* ; pourquoi ?

Appels Système pour Inodes

Toujours : noter que les commandes connues utlisent forcément ces appels.

stat(), lstat(), fstat()	accès à la table des inodes
chmod(), fchmod()	modification droits
access()	vérifier droits fichier
touch(), utime(), utimes()	accès et modification dates
chown()	modifier propriétaire

Accès à l'Inode

À partir du nom ou d'un descripteur sur un fichier ouvert, on peut accéder à l'inode, mais pas aux adresses de blocs (pointeurs dans la table des inodes)... Syntaxe :

```
NAME
stat, fstat, lstat - get file status
SYNOPSIS
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
int lstat(const char *file_name, struct stat *buf);
```

Structure Récupérée

```
struct stat{
  dev_t st_dev; /* device */
  ino_t st_ino; /* inode */
  umode_t st_mode; /* protection */
  nlink_t st_nlink; /* number of hard links */
  uid_t st_uid; /* user ID of owner */
  gid_t st_gid; /* group ID of owner */
  dev_t st_rdev; /* device type (if inode device) */
  off_t st_size; /* total size, in bytes */
  unsigned long st_blksize; /* blocksz for filesyst I/O */
  unsigned long st_blocks; /* number of blocks allocated
  */
  time_t st_atime; /* time of last access */
  time_t st_mtime; /* time of last modification */
  time_t st_ctime; /* time of last change */
  }
```


Exemple d'Accès à l'Inode

On constate qu'on peut récupérer toutes les informations présentées précédemment dans la table des inodes.

Reste à connaitre les quelques masques nécessaires pour extraire une information consistant en une suite de bits de longueur différente de 32 ou 8.

```
struct stat istique;
int arissable = stat ("fichu", &istique);
if (arissable == 0){
      cout << "le fichier fichu a pour num. inode"
           << istique.st_ino;
      if (S_ISREG(istique.st_mode))
           cout << "et c'est un fichier
                régulier"<< endl;
      }
```


Plan

Le Problème de la Taille

Constat : dans la table des inodes, les pointeurs sur les blocs de données sont les adresses de ces blocs. Il faut gérer avec ces pointeurs des fichiers de taille extrêmement variable (euphémisme).

Problème : Comment gérer cette taille avec un nombre fixe de pointeurs dans la table des inodes ?

Pourquoi un nombre fixe ? parce que les systèmes d'exploitation les adorent et cherchent aussi une solution efficace permettant de minimiser le nombre d'accès disque. Par exemple, le nombre de pointeurs est fixe et limité à 13 jusqu'à 16 pointeurs selon les version d'Unix. Et pourtant, il va bien falloir gérer de très gros fichiers comme des tout petits.

Blocs Directs et Indirects

Principe : la gestion de la taille se fera avec quelques adresses pointant directement sur les blocs de données, comme vu jusque là. Ensuite, dès que les fichiers grossissent, on construit des blocs dits indrects, dont le contenu est lui-même un ensemble d'adresses, qui permettent donc d'étendre les adresses directes.

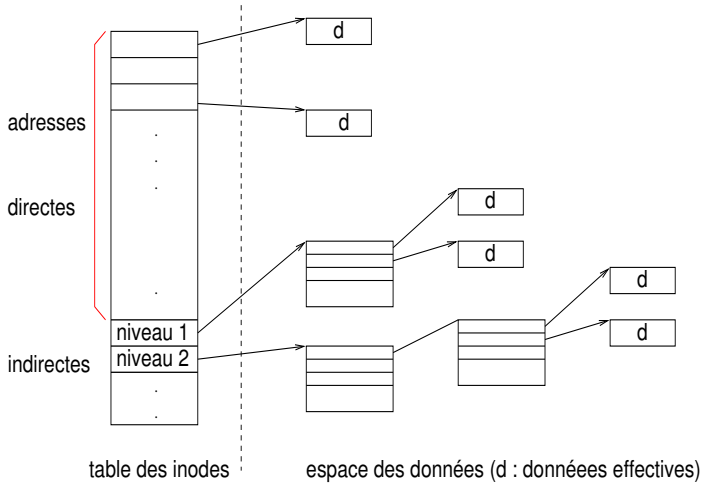
Méthode :
Adressage de tout fichier : un arbre dont les feuilles sont les blocs de données et les nœuds internes des blocs d'adresses.

Chaque niveau de l'arbre autre que les feuilles est une liste des adresses des enfants.

Méthode en Détails

- pointeurs directs : contiennent l'adresse de blocs de données.
- pointeurs indirects : contiennent l'adresse de blocs contenant des adresses d'autres blocs, de données ou d'adresses, selon de niveau d'indirection.
  - indirects à n niveaux : contiennent les adresses de blocs, eux-mêmes contenant des adresses de niveau $n-1$. Une adresse de niveau 0 est une adresse de bloc de donnée.
- le premier niveau de l'arbre est dans la table des inodes ; il est de taille fixe : n_0 pointeurs directs, n_1 pointeurs de niveau 1, n_2 de niveau 2, etc.
- les divers unix : $10 \leq n_0 \leq 13$, $n_1 = 1$, $n_2 = 1$, $n_3 = 1$

Arbre d'Allocation



Exemple - Taille Maximale d'un Fichier

On prend une table d'inodes classique avec 10 pointeurs directs et un seul pointeur pour chacun des 3 niveaux suivants.

Données de base : on suppose que

- la taille des blocs de données est $4K$ octets,
- les adresses sont codées sur 32 bits (4 octets).

1. Taille maximale atteinte par les blocs directs :
   10 blocs $\times 4K$ octets $= 40K$ octets
2. Pour les blocs indirects, il faut d'abord calculer le nombre d'adresse contenues dans un bloc de données, c'est-à-dire *taille bloc*/*taille adresse*,
   ici $4K$ octets $/4$ octets $= 1K$ adresses.

Taille Maximale d'un Fichier - suite

3. Taille maximale atteinte par le 1^{er} niveau :
   1 pointeur $\times 1K$ adresses $\times 4K$ octets $= 4K^2$ octets $= 4M$ octets.
4. Taille maximale atteinte par le $2^{ème}$ niveau :
   1 ptr $\times \underbrace{1K \times 1K}_{1M \text{ adr}}$ adr $\times 4K$ oct $= 4K^3$ oct $= 4G$ oct.
5. Taille maximale atteinte par le $3^{ème}$ niveau :
   1 ptr $\times \underbrace{1K \times 1K \times 1K}_{1G \text{ adr}}$ adr $\times 4K$ oct $= 4K^4$ oct $= 4T$ oct.

On peut donc approximer la taille maximale d'un fichier **atteinte par l'adressage des blocs**, ici $40Ko + 4Mo + 4Go + 4To$, au dernier niveau d'indirection, ici $4T$ octets.

Exercices

1. Si la taille des blocs de données est doublée (à $8Ko$) calculer le facteur de multiplication de la taille maximale d'un fichier (approximation).
2. Avec des blocs de $4Ko$, sur quelle longueur faudrait-il coder la taille du fichier dans la table des inodes, afin de satisfaire la taille atteinte par l'adressage des blocs ?
3. On considère que les blocs non terminaux (ceux contenant des adresses de blocs) sont <<volés>> à l'espace des données. Combien de blocs sont ainsi subtilisés avec les blocs de $4Ko$ et le fichier de $4Go$?

Plus difficile :

4. Situation de départ : blocs de $4Ko$, taille du fichier codée sur 32 bits. On veut agrandir la capacité maximale d'un fichier et passer à $32Go$. Analyser les solutions possibles ; Que peut-on proposer pour assurer la compatibilité entre les deux situations ?

Mais, En Fin de Compte

Mais il faudrait prendre en compte aussi :

- le codage de la taille du fichier dans la table des inodes,
- l'espace physique réel disponible sur la partition disque.

Exemple :

- lorsque la taille du fichier dans la table des inodes est codée sur 32 bits, la taille maximale d'un ficher est de 2^{32} octets, soit $4Go$;
- si l'espace des données de la partition est supérieur à $4Go$, alors $4Go$ reste la taille maximale réelle ;
- dans ce cas, avec les blocs de $4Ko$ de l'exemple, le $3^{ème}$ niveau d'indirection est non utilisé.

Plan

Lien Symbolique

Objectif : dépasser les limites des liens durs, à savoir, la limitation à une même partition et l'impossibilité de pointer sur un répertoire.

Méthode : un fichier de type nouveau permettant de désigner (pointer sur) un *chemin* ; on parlera encore de *pointeur* et *pointé* ; il n'y a pas de tireur.

Exemple : créer un lien symbolique `/repulsif/fiacre` (pointeur) sur `/repus/repas/fief` (pointé).

Point de vue du système : toute action demandée sur le pointeur agira sur le pointé, sauf quelques exceptions.

Exemple de Lien Symbolique

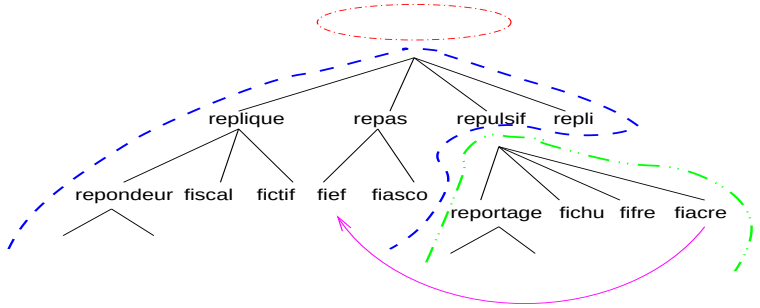


Table des inodes partition verte

num.	type	droits	liens	prop.	taille	dates	pointeurs
3232	`l`	`rwxr-xr-x`			17		99999

Attention :

- noter le type `l` : ni fichier, ni répertoire, mais nouveau type,
- noter la longueur : exactement celle de la chaîne de caractères `/repus/repas/fief` (contenu du bloc 99999)

Caractéristiques des Liens Symboliques

Les liens symboliques peuvent se faire sur des répertoires, dans une même partition ou dans des partitions différentes.

Attention : on peut créer des incohérences (boucles, . . .).

Syntaxe de la **commande** permettant de créer un lien symbolique :
`ln -s [chemin pointé] [chemin pointeur]`

L'**appel système** est *symlink()*.

Constater la différence : la même commande, `ln` crée des liens symboliques ou durs, en fonction de l'option passée.
Les appels système sont différents (*link()* et *symlink()*).

Opérations sur des Liens Symboliques

Exemple sur répertoire : Supposons l'opération
`ln -s /repus/repas /repus/repulsif/reportage/reponse`
L'utilisateur fait : `cd /repus/repulsif/reportage/reponse`

Question : Où est-il réellement ?

Réponse : ça dépend des interprètres de langage de commande ! Penser à `cd` •• pour voir la difficulté.

Suppression : c'est une des opération qui s'effectue directement sur son paramètres. On peut donc supprimer le pointeur, ouf ...

Affichage : `ls -l /repus/repulsif/reportage` donnera pour `reponse` la ligne
`lrwxr-xr-x ... ... ... 12 reponse -> /repus/repas`
les droits et le propriétaire sont ceux relatifs au pointeur.

Remarques Globales sur les Liens

Liens symboliques :

- **Rien** dans la table des inodes ne permet de savoir si on pointe vers un fichier ou répertoire.
- **Aucune** vérification n'est faite lors de la création d'un lien symbolique : le pointé peut ne pas exister, on a pu créer une incohérence, ... Les vérifications seront faite uniquement lors des demandes d'ouverture qui agissent sur les pointés.
- L'appel système *stat()* agit sur le pointé. Utiliser *lstat()* si on veut obtenir les caractéristiques du pointeur.

Tous types de liens :
Le seul moyen permettant de connaître l'ensemble des pointeurs est de parcourir l'arborescence. On ne peut pas *remonter* d'un inode vers les liens durs, ou les liens symboliques qui le référencent. En fait, comme pour les pointeurs, on ne peut *remonter* d'un objet vers ses pointeurs.

Plan

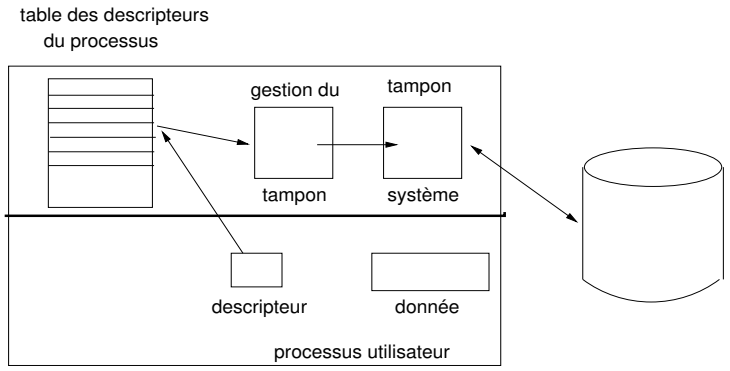
Quels Problèmes

Problèmes traités dans ce chapitre :

- Comment est-ce que le système gère les fichiers ouverts par les processus ? Que se passe-t-il losque plusieurs processus partagent le même fichier ?
- Pourquoi est-il nécessaire de vérifier la cohérence entre le contenu des répertoires et la table des inodes ? Que peut-on vérifier ? Peut-on réparer ?
- Comment est réalisée l'association des systèmes de fichiers simple en un système de fichiers global ?
- Des questions sont restées sans réponse sur les droits des fichiers et répertoires.

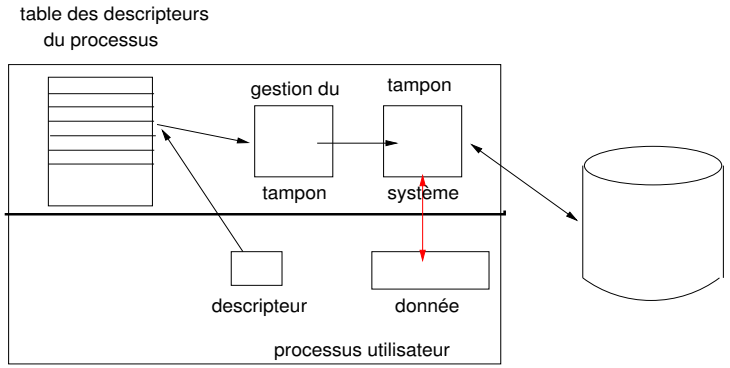
Plan

Ouverture d'un Fichier



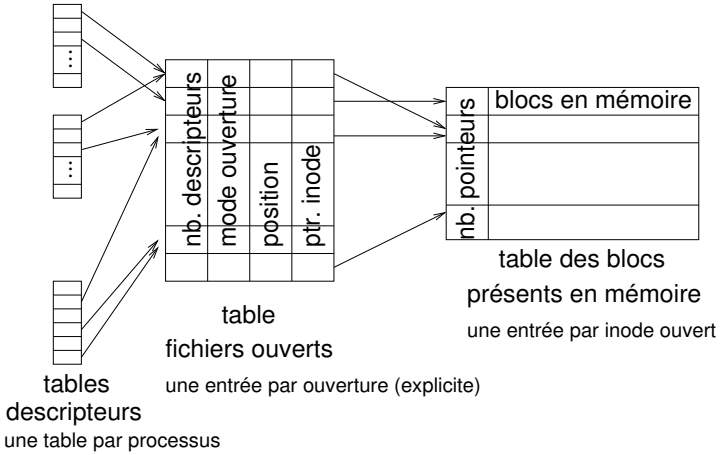
- Le descripteur est un indice vers un élément de la table des descripteurs de ce processus. Il y une table par processus.
- Le tampon système contient au moins un bloc du fichier ouvert.
- Noter que la gestion de ce bloc dépend du type de périphérique.

Accès à une donnée



- La demande de l'utilisateur provoque un transfert tampon système ⇔ espace utilisateur.
  Elle ne provoque pas toujours un transfert disque ⇔ tampon système.

Table des Fichiers Ouverts du Système



Le système gère toutes les demandes d'ouverture de fichiers par la *table des fichiers ouverts du sytème*.

Questions Soulevées

Plusieurs questions se posent à la vue de cette table des fichiers ouverts et les tables associées :

- Quel rapport entre la table des blocs présents en mémoire (i.e. des inodes ouverts) et le tampon système vu précédemment ?
- À quelle situation correspond le fait d'avoir des descripteurs de processus différents pointant sur la même entrée dans cette table ?
- Dans quelles conditions peut-on avoir pour un même processus deux descripteurs pointant vers la même entrée de cette table ?
- À quelle situation correspond le fait d'avoir plusieurs pointeurs de la table des fichiers ouverts vers la même entrée dans la table des inodes ?

Table des Blocs Présents en Mémoire

La *table des blocs présents en mémoire* (appelée aussi -malheureusement [1]- table des inodes en mémoire) représente l'ensemble des tampons du système pour l'ensemble des fichiers ouverts par tous les processus.

On constate que pour des raisons d'efficacité, le système va ramener en mémoire centrale **quelques** blocs de chaque fichier ouvert, en fonction de **prévisions** qu'il peut faire, de sorte à gagner du temps sur les entrées-sorties.

Il y a donc un **asynchronisme** entre les demandes des utilisateurs et leurs réalisation réelle : les données à écrire seront conservées en mémoire, dans la tables des inodes, jusqu'au moment jugé opportun par le système pour les transporter sur le périphérique ; des prévisions sur les lectures permettront de les devancer.

1. ce n'est pas une copie en mémoire de la tables des inodes !

Ouvertures Multiples de Fichiers

Lorsque deux processus ouvrent le **même fichier** (ils demandent explicitement un *open()*) ils auront chacun son propre pointeur sur la table des fichiers ouverts.

Ceci est vrai que cette demande d'ouverture soit en lecture, écriture ou les deux. Seuls comptent leurs droits de réaliser ces opérations.

Dans ce cas, chaque processus aura dans sa table des descripteurs un pointeur vers une entrée différente dans la table des fichiers ouverts. Ces deux éléments de la table des fichiers ouverts pointeront vers la même entrée de la table des inodes en mémoire.
Noter l'expression *une entrée par ouverture explicite* dans le schéma.

Contenus des Tables

nb. descripteurs nombre d'éléments pointant sur la même entrée de la table des fichiers ouverts ($\neq$ nombre de processus)

mode d'ouverture classique : lecture, écriture, lecture et écriture

position position courante atteinte (voir *lseek()*) ; par exemple, pour un fichier ouvert en lecture, après lecture de 3 caractères, la position courante est 4. Noter que c'est la comparaison entre cette position et la taille du fichier qui permet de savoir si la fin de fichier est atteinte.

ptr. inode un pointeur sur la table des inodes en mémoire.

nb. pointeurs nombre d'entrées de la tables des fichiers pointant sur le même élément ; permet de savoir si on peut fermer effectivement un fichier, c'est-à-dire vider sur le périphérique s'il y a eu écriture et libérer l'espace.

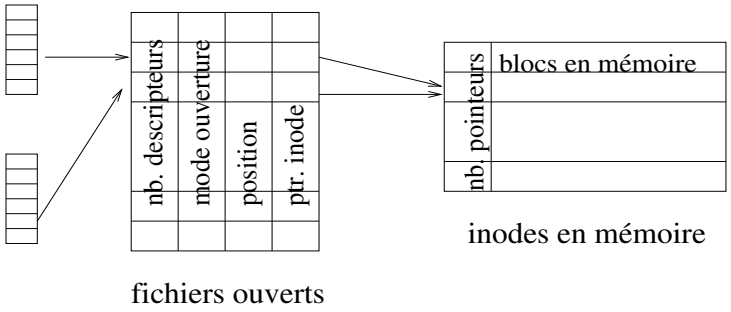
Résumé

Si un processus a fait *open()* il a forcément obtenu une nouvelle entrée dans la table des fichiers ouverts, pointant soit vers un nouvel élément dans la table des inodes en mémoire, soit vers un élément déjà existant dans cette dernière.

On peut partager une même entrée dans la table des fichiers ouverts soit par héritage entre processus, soit en demandant un nouveau descripteur sur un fichier précédemment ouvert dans le même processus.

Ouvertures Multiples de Fichiers - Exemple

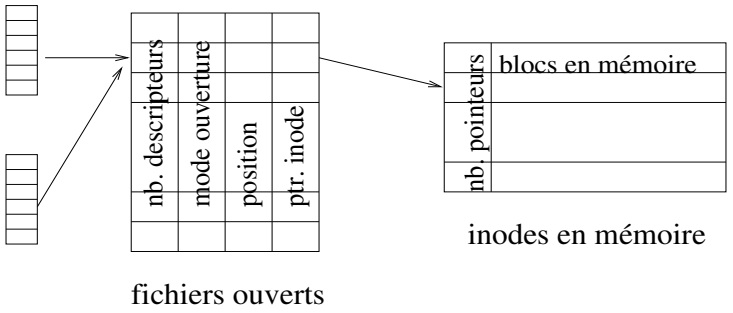
Deux processus ouvrant le même fichier, chacun avec ses propres paramètres, se représentent ainsi :



- Noter que les ouvertures peuvent être conflictuelles : les deux processus en écriture sur la même donnée du fichier, l'un en lecture et l'autre en écriture sur la même donnée, . . .

Accès Multiples à des Fichiers - Exemple

Un processus hérite de l'ouverture faite par un autre :



- Noter que les opérations peuvent être conflictuelles, avec les mêmes remarques que précédemment.
- **Attention** : la même entrée dans la tables des fichiers ouverts est accessible aux deux processus.

Remarques, Questions

- Un même processus peut obtenir plusieurs descripteurs sur une même entrée de la table des fichiers ouverts, **pas** en faisant deux ouvertures, mais en utilisant *dup(), dup2()*
- Que se passe-t-il lorsqu'un processus ouvre le même fichier en faisant deux demandes *open()* ?
- Dans le cas d'ouverture multiple en écriture, par deux processus, quelle sera la version enregistrée sur disque ?
- Un fichier contient la chaîne de caractères *abcdefghij*. Il est ouvert par un processus $P1$ qui crée un clône $P2$. $P1$ veut lire 4 caractères puis 2 autres dans une lecture suivante. $P2$ veut lire 1 caractère puis 2. Qu'obtiennent-ils comme résultats des lectures ?

Plan

Besoin de Contrôler la Cohérence

Un dysfonctionnement grave peut avoir lieu si le système de gestion des fichiers est corrompu. Exemples de défauts :

- un bloc se trouve à la fois libre (dans la liste des blocs libres) et occupé, (pointé par la table des inodes),
- la taille des fichiers n'est pas cohérente avec le nombre de blocs occupés,
- le nombre de liens durs dans la table des inodes n'est pas cohérent avec le nombre de pointeurs,
- deux fichiers occupent le même bloc de données. . .

Certains défauts sont faciles à rectifier. D'autres sont difficiles ou impossibles à corriger. Enfin, certaines corrections faciles peuvent entraîner la perte d'un ou plusieurs fichiers.

Une Perte de Temps à s'Offrir

Constat : Certaines vérifications ne peuvent se faire qu'en parcourant toute l'arborescence. Donc c'est forcément long. Et **indispensable** à faire.

Quand ? Le moins souvent *possible*... À chaque démarrage du système ? Lorsque le système a été arrêté improprement ? Plus le volume des disques augmente, plus le volume des partitions grandit, moins on en a envie. Il n'y a pas une bonne solution.

Comment ? Il faut balayer toutes les partitions (tous les sous-arbres) au moins une fois. Donc

- il faut avoir des algorithmes efficaces, qui évitent le plus possible de refaire des parcours,
- il faut faire des exécutions parallèles permettant de vérifier plusieurs partitions à la fois.

Attention : Un parcours séquentiel de la table des inodes est utile, mais ce type d'accès n'est pas offert en tant qu'appel système.

Exercices

1. Proposer une correction lorsque le nombre de liens pour un inode i dans la table des inodes est supérieur (resp. inférieur) au nombre f de fichiers trouvés référençant i.
2. Montrer que la situation où deux fichiers occupent un même bloc de données est forcément incohérente ; Étudier les solutions possibles et proposer la correction qui vous semble la mieux adaptée.
3. Donner un exemple où la correction que vous venez de suggérer à la question 2 est mauvaise ou inadaptée.

Plan

Droits supplémentaires

Question : Peut-on améliorer les droits de base, soit pour interdire l'effacement de fichiers non propriétaires, soit pour accéder à des fichiers ou répertoires dans des conditions restreintes.

Idées générales : Ajouter quelques informations et associer des droits différents aux processus selon que l'on regarde les droits d'accès aux fichiers ou l'aspect propriété du processus.

4 bits	1 bit	1 bit	1 bit	9 bits
type f.	set_uid	set_gid	*sticky*	droits classiques

Noter que pour le type on connait maintenant plus de deux types...

Développement

L'élément noté *sticky* sert a empêcher la destruction de fichiers dont on n'est pas propriétaire. Voir typiquement */tmp*. Ce droit est représenté par la lettre `t` et on le positionne par `chmod +t [nomRépertoire]` :

```
drwxrwxrwt 9 root root 8192 ............. /tmp
```

Hors cadre de ce cours

Le bit `set_uid` permet de prendre temporairement l'identité du propriétaire du fichier exécutable. Temporairement se réduit uniquement à la durée d'exécution.
`set_gid` agit de façon identique, mais sur le groupe du propriétaire de l'exécutable.

Plan

Communication Entre Processus

Jusque là chaque processus vivait de façon autonome, confiné et disposant seul de son espace mémoire.
Mais cette solitude est trop restrictive ; un besoin de communication avec les autres processus devient pressant.
Les besoins de communications entre processus sont de deux ordres :

- des données échangées entre processus, chacun les traitant à sa façon, un seul processus disposant de la donnée à un instant donné,
- des données communes qu'ils pourraient partager, accéder et modifier en commun.

Ces échanges et partages sont utiles tant pour les processus système qu'utilisateurs.

Exemples de Partages et Communications

- Une base de données gérant l'allocation de places à un spectacle est une donnée commune partagée entre plusieurs processus de réservation.
- Lorsqu'un processus parent prend connaissance de la fin d'un enfant, il y a expédition d'une donnée par l'enfant à destination du parent.
- `unePatate| chaude | pourToi`
  est un échange de données entre trois processus ; `unePatate` envoie ses résultats à `chaude`, qui les utilise pour fournir des résultats à `pourToi`, qui les utilise, on se demande pourquoi, pusiqu'on ne le dit pas ici.

Types de Communications

La **forme** des échanges ou partages peut être :

simple : l'échange est réduit à une occurence ou un seul élément, expédié par un processus à destination d'un autre ; par exemple :
- l'attente parent ↔ enfant ;
- un événement : une interruption ou un signal qui déclenchent une action (un gestionnaire).

complexe : échanges ou partages continus de données avec gestion de la protection et synchronisation.

Exemples

Cas appelés ***simples*** :

- `wait()` et `waitpid()` permettent à un parent d'attendre la fin d'un ou plusieurs descendants ;
- `kill` est une commande envoyant une information (appelée *signal*) à un processus.

Cas plus ***complexes*** :

- `unePatate|chaude| pourToi` revient à lancer trois processus en reliant la sortie standard de `unePatate` avec l'entrée standard de `chaude` ainsi que la sortie standard de `chaude` avec l'entrée standard de `pourToi`.

On verra que le système prend en charge la synchronisation, par exemple, ne pas annoncer à `chaude` ou à `pourToi` qu'il n'y a plus de données si `unePatate` n'a pas fini son exécution.

Éléments de Solution

Pour résoudre ces problèmes, il faut :

- des moyens de communication, données ou structure de données que les processus peuvent échanger ou auxquelles ils peuvent accéder ;
- des primitives d'accès et de protection ;
- des mécanismes de synchronisation.

Le système peut assurer dans certains cas la **prise en charge** de la protection ou de la synchronisation, soulageant d'autant les processus utilisateurs (et les programmeurs). Mais lorsqu'il ne peut le faire, les processus utilisateurs en auront la charge.

Ce Chapitre

Dans ce chapitre on traite :

- de quelques échanges simples entre parents et descendants ainsi que des signaux ;
- des échanges plus complexes avec des tubes simples ou nommés.

D'autres formes de communications existent et certaines sont traitées dans un chapitre séparé.

En particulier, on peut aussi partager l'espace mémoire d'un même processus par plusieurs *activités* parallèles appelées *processus légers* ou encore *threads*. Ce type de partage sera vu plus loin.

Plan

Échanges Parent - Enfant

Principe (sous Unix) : Tout processus qui se termine annonce à son générateur sa fin.

Le générateur peut :

- prendre connaissance de cette fin avec les primitives `wait()` ou `waitpid()` et analyser des informations relatives à cette fin (s'est-il terminé normalement ? ...) ;
- ignorer toute fin de descendant.

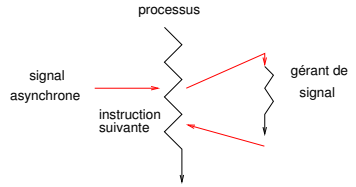
L'enfant est dans un état `zombi` (appelé aussi `defunct` dans certains systèmes) à partir de sa terminaison jusqu'à la prise en considération par le parent. Ce dernier peut être terminé... Voir la bibliographie pour une étude détaillée.

Noter que l'enfant ne sait pas que son parent se fiche de sa fin, c'est pourquoi il sera dans un état `zombi` tant que le parent n'est pas terminé.

Signaux

Caractéristiques des signaux :

- Une forme d'*interruption* logicielle ; appelée ainsi à cause du comportement asynchrone : le processus ne sait ni s'il recevra ni quand il recevra un signal, d'où la similitude de fonctionnement avec une interruption matérielle.



- C'est un moyen pour expédier à un processus une information urgente.
- Toutes les informations urgentes sont connues et cataloguées, dans une liste complète et exhaustive du système.

Exemples

- Une erreur de *segmentation* provoque l'expédition d'un signal appelé `SIGSEGV` au processus fautif (actif), l'expéditeur étant une fonction système ;
- `Ctrl C` (contrôle c) génère l'expédition d'un signal, dont le traitement par défaut est l'arrêt du processus.
- `kill -9 32600` expédie le signal numéro `9` au processus numéro `32600`. Le gérant du signal `9` consiste à détruire (terminer) le processus en question.

Il est **préférable** d'appeler les signaux par leurs noms plutôt que par leurs numéros. En effet, certains signaux portent des numéros différents selon le système d'exploitation.

Exemple : `kill -HUP 32600` ou `kill -SIGHUP 32600` sont préférables à `kill -1 32600`, car portables.

Actions Possibles sur un Signal

Un processus peut :

- accepter l'action par défaut prévue (souvent arrêt), ou
- gérer le signal (pas tous les signaux), ou
- l'ignorer (pas tous).

Dans tous les cas, une action par défaut est prévue dans le système. Cette action peut être d'ignorer le signal.

Seuls les processus issus du même propriétaire peuvent échanger des signaux.

Gestion des Signaux

Pour Programmer la gestion d'un signal il faut :

- écrire la gérante(i.e. la fonction gérante),
- faire l'association entre l'identité du signal et la fonction. Cette association permet d'indiquer au système la fonction à exécuter si le signal se produit.

Forme générale des programmes :

```
void gerante (int numeroSignal){
        //contenu de la gérane
}
int main(){  ...
associer (nomSignal, gerante);
//à partir de là, gerante() sera appelée si le
//signal identifié par nomSignal se produit
}
```


Association Signal et Fonction

`associer` se dit `sigaction()` en `C`. Signature :

```
#include <signal.h>
int sigaction(int signum, //identité du signal
     const struct sigaction *act, //cf. ci-dessous
     struct sigaction *oldact);//NULL souvent
```

Avec la définition :

```
struct sigaction {
      void (*sa_handler)(int); //gérante
      sigset_t sa_mask; //masque à appliquer
      int sa_flags; //options
}
```

`gérante` ci-dessus est un *pointeur sur fonction*. Noter qu'en C, le *nom* d'une fonction est un pointeur sur son code.

Quelques Signaux

Signal	Valeur	Action	Signal	Valeur	Action
SIGHUP	1	T	SIGPIPE	13	T
SIGINT	2	T	SIGALRM	14	T
SIGQUIT	3	T	SIGTERM	15	T
SIGILL	4	T	SIGUSR1	30,10,16	T
SIGFPE	8	M	SIGUSR2	31,12,17	T
SIGKILL	9	TEF	SIGCHLD	20,17,18	I
SIGSEGV	11	M	SIGCONT	19,18,25	
			SIGSTOP	17,19,23	DEF

Action désigne l'action par défaut. `SIGUSR` sont des signaux sans signification particulière, laissés à la disposition de l'utilisateur.

T : terminer processus
I : ignorer signal
M : image mémoire
D : interrompre processus
E : ne peut être géré
F : ne peut être ignoré

Exemple

Gestion du signal `SIGSEGV` de sortie de l'espace mémoire.

```
//déclaration de la gerante
void geger (int elSig){
   cout <<"recu le signal "<<elSig<<"; ";
   cout <<"que faire que faire?" <<endl;
   exit(1);
}
int main(){
   struct sigaction actshoum;
   actshoum.sa_handler = geger;
   int ru=sigaction(SIGSEGV, &actshoum, NULL);
   int *demer; demer=NULL;
   cout <<"oouups\n" <<*demer<<"trop tard"<<endl;
}
```


Remarques

- Un seul bit par signal est utilisé dans la table des processus du système ⇒ des signaux peuvent être perdus, par exemple dans le cas d'expéditions en rafale.
- Pour ignorer un signal le gérant s'appelle `SIG_IGN` : dans l'exemple, `actshoum.sa_handler=SIG_IGN`.
- De même, `SIG_DFL` désigne le gérant par défaut.

Il faut refaire l'association (signal ↔ gérant) à chaque changement de gérant.

Compléments, Questions et Réponses

- Faut-il relancer une entrée-sortie interrompue par un signal ?

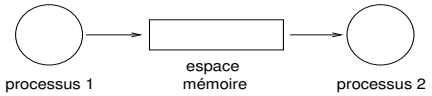
  Certains systèmes le font, d'autres (dont linux) non. Dans ce dernier cas, il faut relancer l'entrée-sortie, en attribuant à `sa_flags` (voir la structure `sigaction`) la valeur `SA_RESTART` ; dans l'exemple, `actshoum.sa_flags=SA_RESTART`.
- Quand est-ce que le système consulte et avertit les processus de l'arrivée de signaux ?

  Lors du passage du mode noyau au mode utilisateur. En particulier, lors du retour d'une interruption qui a généré un signal, le signal sera traité à la fin du traitement de l'interruption.

Plan

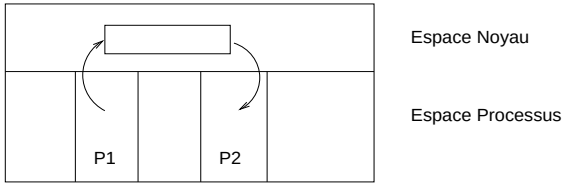
Notion de Tube

Un *tube* est un espace de mémoire accessible par plusieurs processus, tel qu'un processus (ou plusieurs) peu(ven)t y déposer des données (écrire) et/ou extraire des données (lire).



Question : par rapport aux segments mémoire des processus, où est cet espace ? **Réponse** partielle : certainement pas dans l'espace d'un des processus... Alors vraisemblablement dans l'espace des données du système d'exploitation.

Principe de Fonctionnement



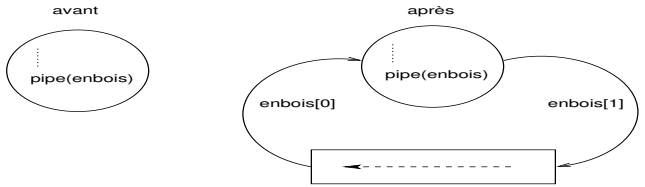
Vu des processus, tout se passe comme s'ils accédaient un *pseudo*-périphérique ; les mouvements se feront entre leurs espaces de données et l'espace du système qu'ils accèdent en commun.

Dans la suite, on voit deux types de tubes, qui diffèrent par les méthodes de création de l'espace ainsi que par la parenté des processus accédant à l'espace : les tubes *simples* et les tubes *nommés*.

Tubes Simples

Un *tube simple* est une structure de donnée en mémoire créée par un processus.

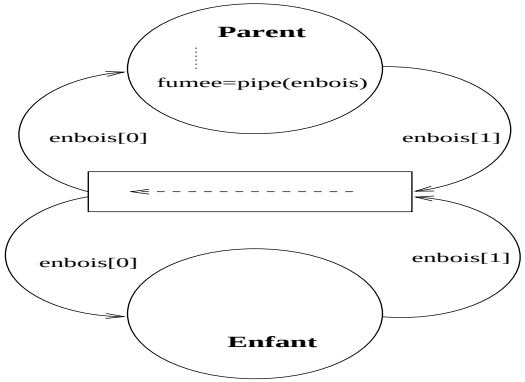
- Un appel système spécifique de création : `pipe()` ;
- Résultat : deux descripteurs, permettant des accès par : `read()` et `write()`. Fermeture : `close()`.



Exemple :
```
int enbois[2]; //déclaration
int ranet=pipe(enbois);//création
int erstice=write(enbois[1], ...,...);
int erne=read(enbois[0], ...,...);//utilisation
```

Modèle Lecteur-Écrivain et Tubes

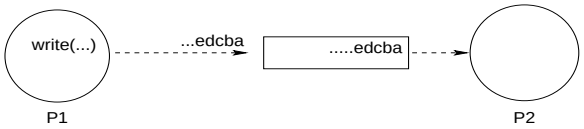
Reste à résoudre le problème du partage du tube par plus d'un processus. L'héritage des descripteurs lors d'un clônage (`fork()`) permet de répondre.



On dispose ainsi d'un espace mémoire dans lequel parent et enfant peuvent lire et écrire.

Tubes - Caractéristiques

- flots de données (on parlera de caractères pour simplifier), avec une gestion *premier entré, premier sorti* du flot.



- Tout élément consommé (lu) est enlevé du tube : c'est une file... ;
- il y a consommation du caractère en position courante ⇒ pas de déplacement avant ou arrière (pas de `lseek()`) ;
- un tube simple est limité à une hiérarchie issue du processus créateur (donc appartenant à un seul utilisateur) ;
- la taille du tube est bornée ;
- les entrées-sorties sont *atomiques*[2]. Approximation : toute opération commencée est seule à modifier le tube ; elle est entièrement terminée avant le passage à la suivante.

2. notion approfondie dans un prochain chapitre

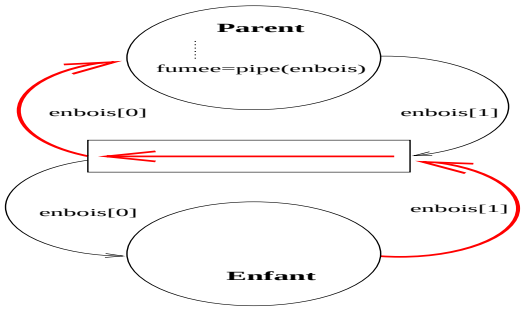
Tubes - Synchronisation

Le système prend en charge le fonctionnement suivant :

- Lecteur bloqué sur une demande de lecture lorsque le tube est vide.
- Écrivain bloqué sur une demande d'écriture lorsque le tube est plein.
- Lecteur averti s'il veut lire alors que le tube est vide et qu'il n'y a plus d'écrivains : `read()` retourne 0 (zéro) et c'est le seul cas où zéro est retourné dans ce mode de fonctionnement appelé bloquant.
- Écrivain averti s'il veut écrire et qu'il n'y a plus de lecteurs, quel que soit l'état du tube : ce n'est pas un résultat de `write()` ! mais la réception du signal SIGPIPE que matérialise l'avertissement.
- Une exclusion mutuelle pour l'accès au tube est assurée (toute entrée-sortie est terminée avant de réaliser une autre).

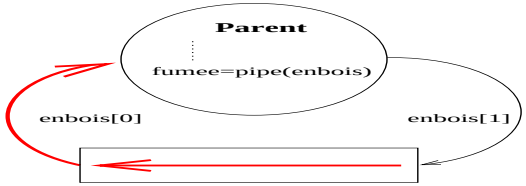
Tubes - Interblocage

Attention : jusque là, on a admis que le système fermait les fichiers non fermés à la terminaison d'un processus. Ce fonctionnement est acceptable tant que l'on n'a pas besoin de fermer les fichiers **avant** la fin du processus.
Reprenons le cas suivant : l'enfant écrit, le parent lit, puis l'enfant se termine.



Blocage

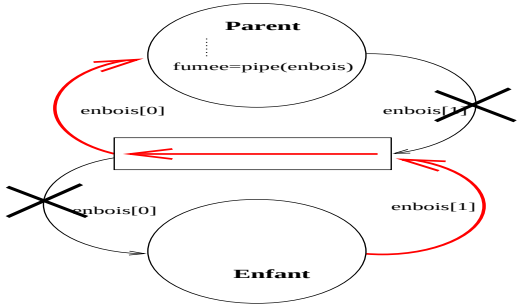
Il y a blocage lorsque le parent va chercher à lire, piégé par lui-même : un descripteur d'écriture est encore ouvert, donc il y a un écrivain présent !



Conclusion : avec les tubes, il faut faire comme dans les trains, attention à la fermeture, sinon blocages et interblocages sont possibles.

Une Pratique Utile

Chaque processus ferme les descripteurs inutiles, avant toute lecture ou écriture.



On peut aussi faire des entrées-sorties non bloquantes, mais il faudra alors prendre en charge la synchronisation.

Interprète de Langage de Commande et Tubes

Le schéma algorithmique de l'interprète de langage de commande lorsqu'une ligne de commande contient un tube, par exemple `chercheLe | deColle` est :

---

se clôner en clône1 ;
créer dans clône1 un tube *denfer* ;
clôner clône1 en clône2 ;
si *(processus clône1)* **alors**
  fermer *denfer[1]* ; se recouvrir par deColle ;
si *(processus clône2)* **alors**
  fermer *denfer[0]* ; se recouvrir par chercheLe ;
si *(dernier caractère est &)* **alors** reprendre lecture clavier ;
sinon attendre la fin de clône1 ;

Interprète de Langage de Commande - Complément

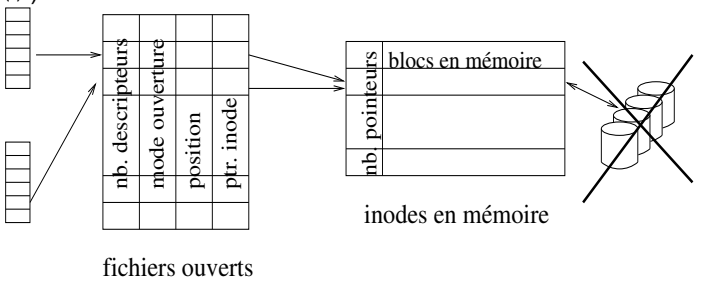
Remarque : Il manque dans le schéma algorithmique précédent les correspondances suivantes :

- dans `clône1`, l'entrée standard et `denfer[0]` devraient coïncider ;
- dans `clône2`, la sortie standard et `denfer[1]` devraient coïncider.

Pour réaliser une telle coïncidence, voir les appels système `dup()` et `dup2()` qui permettent d'obtenir plusieurs descripteurs sur le même objet.

Gestion des Tubes par le Système

Les tubes sont gérés comme un fichier, dans la table des fichiers ouverts du système. Mais contrairement aux fichiers, aucun transfert (disque ↔ mémoire centrale) n'est effectué et aucun déplacement (`lseek()`) n'est autorisé. Pour mémoire :



Limites des Tubes Simples

Un rappel des limites des tubes simples vues précédemment :

- héritage obligatoire de descripteurs si on veut partager le tube entre plusieurs processus ;
- les processus appartiennent au même utilisateur ;
- le partage entre plus de deux processus peut devenir délicat, sauf si on ne cherche pas à savoir qui parmi les écrivains a écrit ou qui parmi les lecteurs a lu.

Les deux premières limites peuvent être dépassées avec les tubes nommés décrits dans le paragraphe suivant.
La dernière est inhérente au fonctionnement des tubes en général.

Plan

Présentation, Caractéristiques

Les tubes nommés offrent les possibilités générales de communication décrites pour les tubes, en étendant l'accès à des processus appartenant à des utilisateurs différents. Caractéristiques :

- Une structure localisée en mémoire centrale, toujours dans l'espace du système, identifiée de façon à permettre l'accès à des processus issus de propriétaires différents.
- Possibilité de Rendez-Vous entre processus.
- Des droits d'accès permettent d'autoriser ou interdire l'accès en fonction des propriétaires des processus.

Elles entraînent plusieurs questions :

- Qui (quel processus) va créer la structure ? Comment ?
- Comment identifier la structure ?
- Comment savoir qu'un processus est présent au Rendez-Vous ?

Identification

Besoin : donner un identifiant à la structure pour que tout processus puisse demander l'accès.

Deux appels système vont être utilisés :

1. `mkfifo()` qui permet de réserver un nom, **sans** créer la structure en mémoire,
2. `open()` tout simplement, qui va créer la structure si elle n'existe pas et demander l'accès.

Principes :

- `mkfifo()` crée un fichier de type `p`, donc un fichier spécial, qui sert **uniquement** à mémoriser un nom, un propriétaire et des droits d'accès.
- une demande `open()` d'un fichier de type `p` provoque la **création** de la structure en mémoire si elle n'existe pas, réalise le **rendez-vous** (détails plus loin) et permet de faire des entrées-sorties conformément aux droits du fichier spécial.

Exemple

```
int olérable=mkfifo("fipeps",
                    S_IRWXU|S_IRGRP|S_IWGRP|I_IROTH) ;
```

va créer un **inode**, de type spécial `p`, avec les droits interprétés comme dans toute ouverture de fichier (ici, en octal, 764) et une entrée dans le répertoire de création.

Dans la table des inodes on aura :

num.	type	droits	proprio.	...	pointeurs
48761	p	comme tout inode			vide

Dans le répertoire on aura :

num	nom
48761	fipeps

Exemple - suite

Si un processus fait

```
int rouvable=open("fipeps",O\_RDONLY);
```

le système vérifie qu'il a les droits correspondants sur le fichier spécial, puis

- crée la structure en mémoire si elle n'existe pas,
- avertit les processus ayant ouvert en écriture qu'il y a un lecteur (c'est le rendez-vous).

La structure sera gérée en mémoire comme un tube simple.

Toutes les entrées-sorties sur `rouvable` se feront dans cette structure.

Exemple - fin

Le processus qui a fait l'ouverture en lecture demandera

```
int rocetro=read(rouvable,...,...);
```

et toutes les lectures se feront dans le tube, avec un blocage si **aucun** caractère n'est présent ; la lecture sera satisfaite (retour positif) dès que le tube ne sera pas vide $\Rightarrow$ le résultat de `read()` contiendra la longueur réellement lue.

De même, si un autre processus fait

```
int ernet = open ("fipeps", O_WRONLY) ;
```

alors ses écritures

```
int raveineuse=write(ernet,...,...);
```

se feront dans le tube.

Rendez-Vous

Question : Comment est réalisé le rendez-vous (qui attend qui) ?

le Rendez-Vous est mis en œuvre à l'ouverture, et garantit l'existence d'au moins un lecteur et d'au moins un écrivain.

`open()` est bloquant : lors d'une demande d'ouverture en lecture, le système vérifie qu'il n'y ait pas au moins un écrivain. Si non, le processus demandeur est endormi. Si oui, c'est que des écrivains attendent ; ils ont fait une demande `open()` en écriture et ont été bloqués ; le système les réveille.

Attention : Un interblocage est possible suite à des défauts de programmation (voir ci-après).

Fermeture

Question : Que se passe-t-il à la fermeture et éventuellement à la réouverture ?

Le fonctionnement est identique à celui des tubes simples concernant la synchronisation et l'avertissement des processus.
Donc lecteurs et écrivains seront avertis de l'absence d'acolytes (seulement lorsque le tube aura été vidé pour les lecteurs).

Lorsque tous les processus ont fermé leurs descripteurs, le fichier spécial n'est **pas détruit**.

Lorsqu'un processus a été débloqué à la demande d'ouverture, il n'est plus possible de se remettre en attente. En d'autres termes, si un lecteur (resp. écrivain) reste seul, il ne sera pas endormi en attendant un autre processus, sauf si étant seul, il ferme le tube et le réouvre.

Droits

Question : Quelles vérifications sont faites sur les droits d'accès ?

Comme pour les fichiers en fonction du propriétaire du processus demandeur. Il n'est pas nécessaire d'être propriétaire du fichier spécial pour créer la structure en mémoire (penser au rendez-vous toujours)

Noter que le fichier spécial reste **vide** tout au long de l'utilisation !

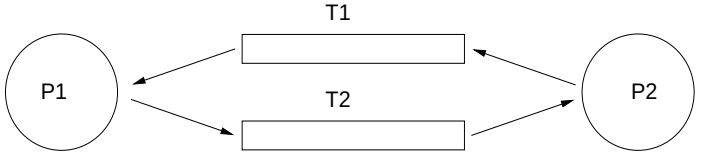
Seule la structure en mémoire se remplit et se vide. Le contenu d'un tube n'est donc **pas** conservé si tous les lecteurs sont partis en laissant des données orphelines dans le tube.

La destruction du fichier spécial se fait comme tout fichier (`rm` ou `unlink()`) et obéit aux mêmes vérifications de droits.

Une commande `mkfifo` existe aussi.

Interblocage - un Exemple

Deux processus veulent échanger des données en utilisant un tube nommé dans chaque direction.



La situation suivante :

P1	P2
`ouvrir(T1,lecture)`	`ouvrir(T2,lecture)`
`ouvrir(T2,écriture)`	`ouvrir(T1,écriture)`

provoque un interblocage, chaque processus attendant un déblocage.
Petit exercice : Un troisième processus peut les sauver.

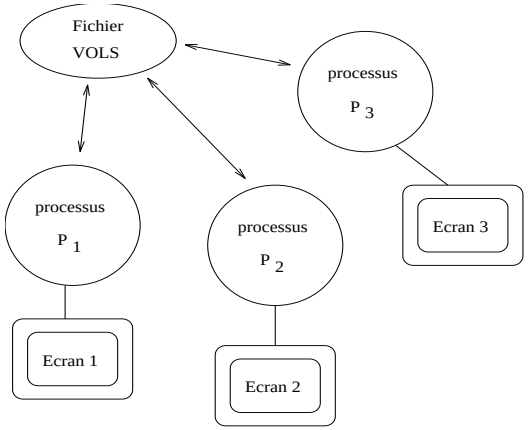
Plan

Objectifs du Chapitre

- Montrer que toutes les applications qui accèdent à des données, plus généralement à des ressources communes, ont besoin de réglementer ces accès ; réglementer veut dire :
  - ne pas laisser plusieurs processus accéder et modifier la même donnée *simultanément*, puisque des incohérences pourraient en résulter ;
  - assurer une certaine équité entre les accédants.
- Ces applications relèvent tant système d'exploitation que des applications des utilisateurs .
- Connaître les primitives que le système d'exploitation fournit afin de réglementer ces accès.

Exemple d'Application Utilisateurs

Réservation de places d'avion (ou train, spectacle etc.).



Variable commune accédée et modifiable par tous : *nombre de places réservées* appelée `nbPlacesRes` pour chaque vol.

Variable commune accédée en lecture seule, *nombre de places total*, appelée `nbPlacesMax` pour chaque vol.

Algorithme Classique et Problème

On suppose que trois processus `P1, P2, P3` exécutent le même code pour réserver des places. On se restreint à un vol donné dans cet exemple. Chaque processus dispose d'une variable locale `nbPlacesDem` représentant le nombre de places qu'il veut réserver suite à une demande locale. Supposons l'algorithme suivant :

---

si *(`nbPlacesDem` $\leq$ `nbPlacesMax` $-$ `nbPlacesRes`)* **alors**
  `nbPlacesRes` $+=$ `nbPlacesDem` ;
  afficher ("réservation effectuée pour" `nbPlacesDem` "place(s)") ;
sinon
  afficher ("il reste seulement", `nbPlacesMax` $-$ `nbPlacesRes`, "places") ;

Un Déroulement Possible

Supposons la situation globale suivante :

$\texttt{nbPlacesMax} = 100$; $\texttt{nbPlacesRes} = 98$;

P1 veut **deux** places et P2 en veut **une**.

P1 est actif ; dans sa sa variable locale il a $\texttt{nbPlacesDem} = 2$.
P1 effectue le test, obtient un résultat positif, puis est interrompu (horloge) de suite après le test :

---

si *(nbPlacesDem* $\leq$ *nbPlacesMax* $-$ *nbPlacesRes)* **alors**
  ⊗nbPlacesRes $+=$ nbPlacesDem ;
  afficher ("réservation effectuée pour" nbPlacesDem "place(s)") ;
sinon
  afficher ("il reste seulement", nbPlacesMax $-$ nbPlacesRes, "places") ;

Déroulement - Suite

On suppose que P2 devient actif.
Dans sa variable locale il a $\texttt{nbPlacesDem} = 1$.
Il effectue le test, et obtient **aussi** un résultat positif.

Ainsi, P2 effectue la modification $\texttt{nbPlacesRes} = 99$, puis affichera *réservation effectuée pour 1 place(s)*.

Lorsque P1 redeviendra actif, il effectuera la modification $\texttt{nbPlacesRes} = 101$ puis affichera *réservation effectuée pour 2 places(s)*.

De façon certaine, on peut affirmer que l'algorithme ne fonctionne pas correctement,

- à cause de l'accès avec modification aux données communes,
- ou à cause de l'interruption qui s'est produite à un *mauvais moment*...

Exemples Système

Dans la gestion des processus, on sait que le nombre de processus maximal est fixé. Il faut vérifier lors de chaque création de processus qu'on ne dépasse pas ce nombre maximal.
Exemple d'algorithme avec deux variables communes accédées par le système : nbProcEnCours et nbMaxProc, autosuggestives.
Un exemple très simple d'algorithme :

---

si *nbProcEnCours* $<$ *nbMaxProc* **alors**
  nbProcEnCours++ ;
  dernierAttrib = dernierAttrib + 1 ;
  retourner (numProc $=$ dernierAttrib) ;
sinon
  retourner(erreur) ;

Déroulement Possible

Supposons que deux processus, P1 et P2, soient en cours d'exécution de *fork()*. Rappelons que les appels système sont interruptibles comme toute autre fonction (voir les quelques exceptions dans la gestion des interruptions).

Comme précédemment, il suffit d'envisager l'interruption suivante lors de l'exécution :

---

si *nbProcEnCours* $<$ *nbMaxProc* **alors**
  ⊗nbProcEnCours++ ;
  dernierAttrib = dernierAttrib + 1 ;
  retourner (numProc $=$ dernierAttrib) ;
sinon
  retourner(erreur) ;

Types de Problèmes

En général, ces problèmes se retrouvent partout où une ressource commune est accédée puis, modifiée en fonction de son contenu courant.
Dans les structures vues jusque là, voici quelques exemples :

- avec les tubes, représentant un problème classique de *producteur consommateur* : gérer le nombre d'éléments disponibles dans tubes lors des entrées-sorties. **Exercice** : pourquoi ? dans quelles circonstances exactement ?
- gestion des impressions dans un système : insertion dans des files d'attente, allocation des périphériques, . . .
- allocation de la mémoire, etc, etc.

Question : Comment résoudre ce problème ?
Réponse partielle : Soit le système offre des primitives qui permettent de vérouiller temporairement une partie du code, soit il faut construire ces primitives . . .

Plan

Section Critique

Environnement : plusieurs processus exécutant un même code et partageant des données communes.

Une **section critique** est une partie du code qui doit être exécutée par un seul processus à la fois.
On dit qu'il y a **exclusion mutuelle** entre ces processus.

Attention : ceci ne veut pas dire que le processus en section critique est ininterruptible, mais seulement que, si un processus P1 est entré dans cette section, aucun autre ne doit pouvoir la commencer avant que P1 n'ait terminé.

Il faut un protocole d'accès (ensemble de règles) que **tous** les processus **doivent** respecter (doivent utiliser) pour exécuter cette section critique.

Section Critique - Propriétés

Remarque préalable : Si un processus décide de passer outre le protocole, alors l'ensemble de l'application *peut dysfonctionner*. Le débogage sera d'autant plus difficile que l'on ne peut pas reproduire la même situation ayant provoqué le plantage.

Une solution doit avoir les **propriétés** suivantes :

- **exclusion** : un seul processus en SC ;
- **évolution** : un processus en dehors de sa SC ne peut bloquer d'autres processus ; autrement dit, seuls les processus demandant à entrer en SC participent à la décision ;
- **attente bornée** : pas de famine. Noter qu'ainsi, la demande d'entrer en section critique ne peut être reportée indéfiniment.

supplément : pas de présomption sur le nombre de processeurs.

Tentatives de Solution

On peut vérifier que les solutions simples ne fonctionnent pas.
Exemple : une variable commune *verrou* initialisée à *faux* et utilisée ainsi dans chaque processus :

---

si *(non verrou)* **alors**
  verrou = vrai ;
  SectionCritique ;
verrou = faux ;

...toujours pour la même raison que les exemples déjà cités.

Voir la bibliographie pour divers développements parfois longs à ce sujet.

Besoin d'Opérations Atomiques

Si l'on dispose d'un outil permettant de façon atomique (d'atome au sens unité - non interruptible, pas au sens explosif) de tester et verrouiller une ressource (donnée, fichier, ...), alors on peut trouver des solutions.

En d'autres termes, on veut tester **et** modifier **simultanément** un état.

Exemple : Supposons qu'une opération *verrouFichier(nomFichier)* soit disponible et que cette opération soit **atomique**. Son fonctionnement consisterait par exemple, à réaliser l'algorithme :

- si un lien symbolique appelé *lock* sur *nomFichier* existe, alors bloquer le processus demandeur ;
- sinon, créer ce lien symbolique. Dans ce cas, tout futur demandeur sera bloqué.

Alors on pourrait résoudre le problème, sachant qu'une opération atomique *leverVerrou(nomFichier)* devrait de même permettre de supprimer le lien.

Situation Actuelle (et pour longtemps ?)

Il existe des solutions purement logicielles (voir la bibliographie) plutôt compliquées (dites aussi élégantes...).
La plupart des solutions s'appuient sur une combinaison *matériel et logiciel*.
Exemples :

- inhiber les interruptions (oui, mais) à l'intérieur d'appels système, donc en mode privilégié ;
- réquistionner le bus des adresses ou données (cas des machines multi-processeurs).

Une des solutions les plus connues consiste à fournir des primitives gérant des *sémaphores*. Dijkstra (fin des années 1960) en est à l'origine.

Plan

Sémaphore

Les sémaphores sont des objets **communs partagés par plusieurs processus**, dotés d'opérations permettant de résoudre la synchronisation de processus.

On commence par une expression simple : Un sémaphore est une variable entière *s*, à valeurs non-négatives, à laquelle sont associées deux opérations **atomiques**, *Demander*(*s*) et *Libérer*(*s*).

Demander (s) {	**Libérer** (s) {
si $s = 0$ **alors** attendre ;	$s = s + 1$;
sinon $s = s - 1$; }	Réveiller un processus en attente ;}

Attention : implicitement, on suppose
- qu'il existe un moyen permettant de mémoriser les demandes non satisfaites ;
- que la variable *s* est initialisée avant toute demande.

Autre forme

Un sémaphore est une classe :
`Sémaphore S { entier val; listeProcessus L; }`
avec deux opérations **atomiques** :

Demander (S) {	**Libérer** (S) {
$S.val = S.val - 1$;	$S.val = S.val + 1$;
si $S.val < 0$ **alors**	**si** $S.val \leq 0$ **alors**
ajouter demandeur dans S.L ;	enlever un processus de S.L ;
le passer à l'état bloqué ;	le passer à l'état prêt ;
}	}

Remarque : demander ≡ P ≡ Wait ≡ Up
libérer ≡ V ≡ Signal ≡ Down

Sémaphores - Utilisation

Dans un problème d'exclusion mutuelle, le sémaphore est une donnée commune, initialisée à 1.
Chaque processus voulant exécuter une section critique fait :

Demander(sémaphore) ;
    SectionCritique ;
Libérer(sémaphore) ;

Fonctionnement :

- Demander(sémaphore) va être passante si le sémaphore est strictement positif et bloquante pour toute autre valeur.
- Libérer(sémaphore) va permettre au prochain demandeur (soit dans la file d'attente ou qui viendra plus tard) de réaliser la section critique.
- Noter que lorsque *sémaphore* ≤ 0 dans la deuxième forme, sa valeur absolue représente la longueur de la file d'attente.

Exemple de Fonctionnement

On reprend l'exemple de réservation de places. Soit `verrouReserv` un sémaphore commun initialisé à 1. Chaque processus va faire :

Demander(verrouReserv) ;
si *(nbPlacesDem* $\leq$ *nbPlacesMax* $-$ *nbPlacesRes)* **alors**
    nbPlacesRes $+=$ nbPlacesDem ;
    afficher ("réservation effectuée pour" nbPlacesDem "place(s)") ;
sinon
    afficher ("il reste seulement", nbPlacesMax $-$ nbPlacesRes, "places") ;
Libérer(verrouReserv) ;

On peut reprendre maintenant l'exemple qui a mené au dysfonctionnement et montrer que cette fois-ci il n'y a plus d'incohérences.

Exemple d'Exécution - 1

On suppose que verrouReserv est créée et initialisée à 1 par un processus d'initialisation de l'application.
Rappel de la situation globale :

`nbPlacesMax` = 100 ; `nbPlacesRes` = 98 ;
P1 veut **deux** places et P2 en veut **une**.
P1 est actif ; dans sa sa variable locale il a `nbPlacesDem` = 2.

- P1 fait `Demander(verrouReserv)` ; alors *verrouReserv* = 0 et P1 passe en section critique, effectue le test, obtient un résultat positif, puis est interrompu (horloge) de suite après le test :
- On suppose que P2 devient actif. Dans sa variable locale il a `nbPlacesDem` = 1. Il fait `Demander(verrouReserv)` ; alors *verrouReserv* = −1 et P2 va dans la file d'attente de `verrouReserv`.
- Si P3 fait aussi une demande de réservation, il passera comme P2 en file d'attente et on aura `verrouReserv=-2`.

Exemple d'Exécution - 2

- Lorsque P1 sera réveillé, il finira la réservation, fera `nbPlacesRes=100` et affichera réservation effectuée pour 2 place(s).
- Tant que P1 n'a pas fait `Libérer(verrouReserv)`, aucun processus ne peut effectuer une réservation.
- P1 fera `Libérer(verrouReserv)` ; alors *verrouReserv* = −1 et un processus de la file d'attente sera réveillé.
- P2 sera réveillé si la file d'attente est gérée selon le principe *premier entré premier sorti*.
- P2 affichera il reste seulement 0 places.
- P2 fera `Libérer(verrouReserv)` ; alors *verrouReserv* = 0 et un processus de la file d'attente sera réveillé, en l'occurence P3, qui donnera un résultat similaire à P2 pour la réservation et finira par remettre *verrouReserv* = 1.

Sémaphores - Question Fondamentale

Les sémaphores et les opérations associées permettent de résoudre l'**exclusion mutuelle** et aussi la **synchronisation** de processus (voir bibliographie pour la synchronisation).

Un système d'exploitation se doit d'offrir aux programmeurs de tels objets et opérations.

Question : Comment fait-il pour assurer l'atomicité des opérations demander() et libérer() ?

Réponse : par l'une des techniques signalées : masquage des interruptions, réquisition de bus, . . . **mais** le système d'exploitation est auto-confiant... Noter qu'il suffit de masquer les interruptions à l'entrée de chaque opération, puis de les démasquer à la fin.

Sous **Unix**, ces primitives sont offertes par des appels système, *semget()* pour définir le(s) sémaphore(s) et *semop()* pour réaliser des opérations comme demander() ou libérer().

Plan

Notion de Verrou Fatal

Cette notion donne lieu à des expressions très poétiques, comme *étreinte fatale* ou *embrasse mortelle*. Autant la connaître pour l'éviter. On parle de **verrou fatal** lorsqu'un processus P1 mobilise au moins une ressource R1 et attend une autre ressource R2 détenue par un processus P2 qui ne la libérera que lorsque P1 aura libéré R1.

Ceci peut être **direct**, comme dans le rendez-vous avec les tubes nommés :

- P1 tient R1 et attend R2 ;
- P2 tient R2 et attend R1.

ou **indirect** :

- P1 tient R1 et attend R2 ;
- P2 tient R2 et attend R3 ;
- P3 tient R3 ....
- . . .
- Pi tient Ri et attend Ri+1 ;
- . . .
- Pn tient Rn et attend R1 .

Caractérisation

Une situation de verrou fatal existe entre processus si les conditions suivantes sont constatées simultanément :

- il y a **exclusion mutuelle** : un processus est en section critique, les autres attendant sa sortie ;
- il n'y a pas de **préemption** : les ressources ne sont pas préemptibles, autrement dit, une ressource ne peut être relachée que volontairement par le processus qui la détient ;
- Il exsite une **attente circulaire** : un ensemble de processus $\{P_0, P_1, \ldots P_i, \ldots P_n\}$ est tel que P_0 attend une ressource tenue par P_1, ..., P_i attend une ressource tenue par P_{i+1}, ..., P_n attend une ressource tenue par P_0.

Cette situation peut se décrire par un graphe orienté, appelé graphe d'allocation de ressources.

Résolution des Verrous Fatals

Comment peut-on traiter et résoudre le problème de ces verrous ?

- Si on veut **éviter** la situation, il faut qu'au moins une condition soit évitée. Pratiquement, ceci revient à éviter la dernière, donc à chercher un circuit dans un graphe orienté... Pas d'algorithme efficace disponible...
- Si on veut **détecter** la situation, on retombe sur le même problème. Et alors, même avec un algorithme efficace, il faudrait décider de tuer arbitrairement un processus ou préempter une ressource.
- Ne rien faire est donc une solution de repos.

Plan

Rôles de la Gestion Mémoire

Cette composante du système d'exploitation doit :

- allouer de l'espace mémoire à chaque processus ;
- protéger l'espace de chaque processus : ne pas laisser un processus déborder hors de son propre espace ;
- répondre aux demandes d'allocation dynamique des processus, relativement aux segments de pile et tas ;
- rendre indépendants la mémoire requise par les processus et l'espace physique disponible ; en particuler un processus peut demander un espace supérieur à celui disponible ;

. . . tout en évitant de bloquer un processus sous prétexte qu'il n'y a pas d'espace disponible et bien sûr avec la plus grande efficacité (on verra que ce n'est pas une phrase creuse).

Allocation de l'Espace

Allouer l'espace signifie :

- à l'arrivée d'un nouveau processus décider où le **loger** et passer les coordonnées de l'espace occupé de la structure de données représentant l'espace libre vers celle représentant l'espace occupé,
- réciproquement, à la terminaison d'un processus passer l'espace précédemment occupé dans la structure de données représentant l'espace libre,
- sans créer des trous (difficiles à gérer), ou plutôt, se donner les moyens d'éviter les trous.

Mais Que Font le Compilateur, l'Éditeur de Liens ?

Un programme exécutable contient les **adresses** des données, des instructions.
Exemple : On considère les instructions suivantes en langage source : `int act, erne, ox;` puis `act=erne+ox;`

En langage machine, les trois données vont être référencées chacune par une adresse, par exemple 3000 pour *act*, 3008 pour *erne* et 3016 pour *ox*.

L'instruction d'addition va consister selon les architectures en un nombre variable d'instructions ; dans tous les cas, il faudra :

1. récupérer le contenu de l'adresse 3000, la stocker dans un registre,
2. de même pour le contenu de l'adresse 3008,
3. faire l'addition, puis stocker le réultat à l'adresse 3016

Choix Difficile

Question : Qui a fait ce choix d'adresses ?
On dénonce : Le complilateur, l'éditeur de liens.
Remarque : Si ce choix est immuable, alors il faudra systématiquement installer l'exécutable à une adresse fixe en mémoire, de sorte que toutes les instructions et données soient **exactement** là où le compilateur et l'éditeur de liens l'ont prévu.

Ceci est **inacceptable** dans un système multi-tâche.

Solution : attribuer des adresses relatives lors de la compilation et édition de liens ; on aura besoin d'un mécanisme permettant de translater les adresses lors de l'exécution.

Plan

Relogement ou Relocation

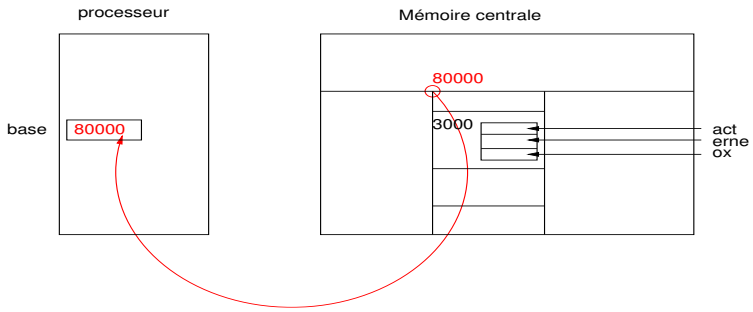
Le **relogement** consiste à pouvoir loger *quelque part* en mémoire (partout, donc là où il y a de l'espace) un processus. Si ce principe est réalisé correctement, on devrait pouvoir même **déplacer** un processus entre deux séquences où il est actif.

L'idée de base consiste à laisser le compilateur et l'éditeur de liens faire le calcul des adresses en partant de 0 (zéro) pour le début du processus. On dira alors que les adresses de chaque exécutable sont des **adresses relatives**.

Pour passer de l'adresse relative à l'**adresse absolue** un mécanisme de translation matériel, inclus dans le processeur sera utilisé.

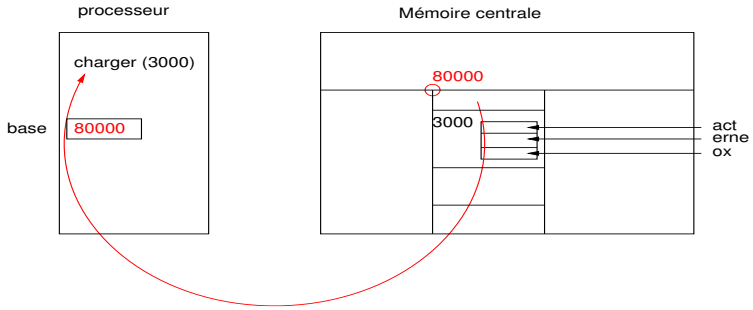
Schéma de Relogement - Exemple

Supposons qu'un processus ait été logé à partir de l'adresse absolue 80000 et admettons qu'il contienne les trois données `act, erne, ox` vues précédemment.



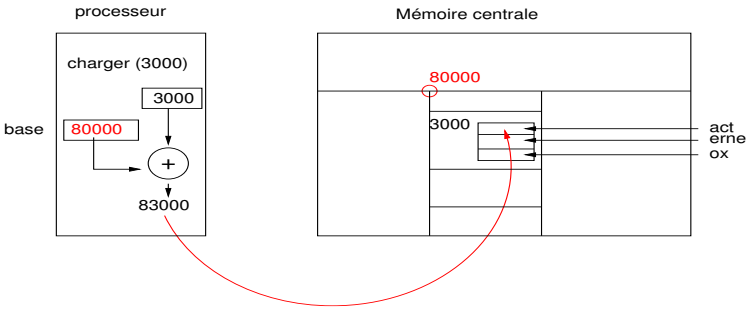
Au chargement de ce processus, l'adresse de base 80000 est allouée par le gestionnaire de mémoire ; lorsque le processus devient actif, l'adresse de base est chargée dans le registre *base*.

Schéma de Relogement - Exemple



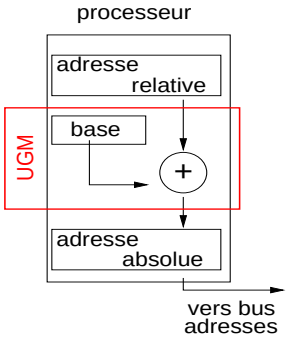
L'instruction `charger le contenu de l'adresse` 3000 est décodée et exécutée. **question embêtante** : comment est-elle arrivée dans le processeur ?

Schéma de Relogement - Exemple



L'adresse relative 3000 est translatée dans le processeur en l'adresse absolue 83000. Cette adresse sera lancée sur le bus des adresses et on récupère la valeur de `act`. Et ainsi de suite.

Unité de Gestion Mémoire



Le travail de translation est fait par l'Unité de Gestion Mémoire, UGM en français, MMU (Memory Management Unit) en anglais.
Son travail va croître au fur et à mesure de ce chapitre.
Résumé du chargement d'un processus :

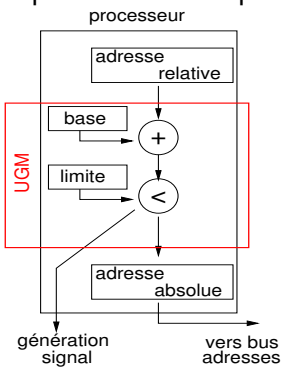
- avant le chargement du processus en MC, allocation de l'espace mémoire et décision de l'adresse début a_0,
- a_0 est stockée dans la table des processus,
- lorsque le processus obtient l'UC, à la fin d'exécution de l'ordonnanceur, a_0 est chargée dans le registre *base*,
- l'exécution peut s'enchaîner.

Remarques

- Dire que l'exécution peut s'enchaîner veut dire qu'on peut amener instructions et données dans le processeur. Noter que toutes les adresses, des instructions, comme des données sont évidemment translatées, ce qui répond à la question posée dans la vue numérotée 280.
- Si par la suite on doit loger ce processus ailleurs (i.e. le déplacer) il suffit de modifier a_0 dans la table des processus.

Protection

Question : Comment peut-on empêcher un processus d'accéder à l'espace d'un autre processus ?



Solution simple : ajouter un nouveau registre dans l'UGM qui contiendra l'adresse maximale et une comparaison qui peut empêcher la suite d'exécution.
Si l'adresse générée ne dépasse pas la limite, alors elle est dirigée normalement vers le bus des adresses, sinon, un signal de débordement (segmentation) est expédié au processus.

On peut maintenant admettre le **principe général** suivant : les processeurs contiennent une UGM composée de registres et capable de faire des opérations.

Plan

Le Problème

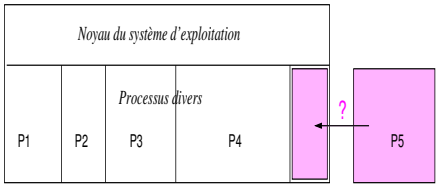
Le système d'exploitation doit gérer un ensemble de processus dont la taille totale peut dépasser la mémoire centrale réelle disponible. Mieux, il doit même gérer cet espace lorsque la demande d'espace d'un seul processus est supérieure à l'espace disponible.

Comment faire ? utiliser l'espace disque temporairement, en complément de l'espace de mémoire centrale. Cet espace disque porte le nom de *mémoire d'échange* (en français) ou *swap* (en anglais).

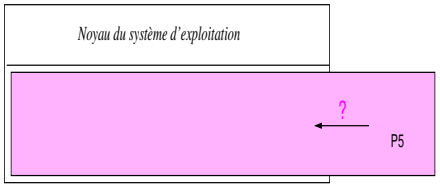
On parle de **mémoire virtuelle** lorsqu'on considère l'espace global demandé par l'ensemble des processus, puisqu'on réagit comme si on disposait physiquement de tout cet espace.

Attention : les allées et venues entre mémoire centrale et disque peuvent provoquer des pertes importantes dans les temps d'exécution comme le montrent les exemples suivants.

Exemples



Dans cet exemple, la composante de gestion de la mémoire du système d'exploitation doit décider quel(s) processus seront délogés temporairement sur disque avant de prendre en compte le nouvel arrivant.



Dans cet exemple, il faudra découper le processus demandeur en morceaux, qui seront chargés/déchargés de la MC. **Comment découper** ? Un exemple connu : le découpage d'une matrice.

Principes Fondamentaux

Constat important : Lors de l'exécution d'un processus, on n'a jamais besoin de l'ensemble du processus en mémoire. En effet, pour exécuter une instruction, il est nécessaire de disposer uniquement du code de l'instruction et des données impliquées dans l'instruction. Le principe de découpage des processus part de ce constat.

Attention : Ne pas confondre *Unité de Gestion Mémoire* et *gestionnaire mémoire*.

- La première est une composante matérielle, faisant partie du processeur ; son rôle est la transformation *adresse relative* → *adresse absolue*, cette transformation pouvant devenir complexe (cf. plus loin).
- La deuxième est une composante logicielle du système d'exploitation ; elle prend en charge l'installation des processus en mémoire, la libération de l'espace, etc.

Plan

Pagination

Plusieurs techniques de découpage des processus existent, dont la *segmentation* et la *pagination* (bibliographie). On étudie cette dernière.

La **pagination** consiste à découper un processus en morceaux réguliers de taille fixe, appellés *pages*. Les pages sont de taille relativement petite : quelques *Koctets*.

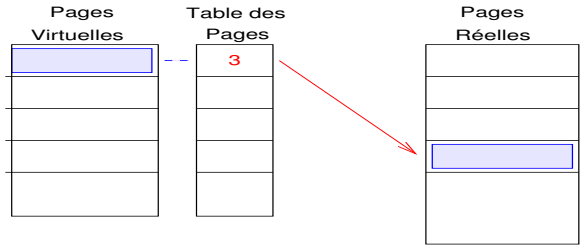
Ainsi, bien que les quatre segments vus pour un processus gardent leur signification et rôle, le découpage en mémoire est différent.

Pour établir une correspondance entre les pages du processus et la mémoire centrale, on va découper en pages de taille identique cette dernière.

Un processus n'a pas besoin d'être entièrement présent en mémoire pour s'exécuter.

Correspondance Page Virtuelle et Réelle

On appellera *page virtuelle* une page du processus et *page réelle* celle de la mémoire centrale.
Une *table des pages* permet d'établir la correspondance entre pages virtuelles et réelles.



La table des pages donne pour chaque page virtuelle, son adresse réelle. Dans cet exemple, le 3 dans le premier élément de la tables des pages signifie que la page virtuelle numéro 0 est dans la page réelle numéro 3.

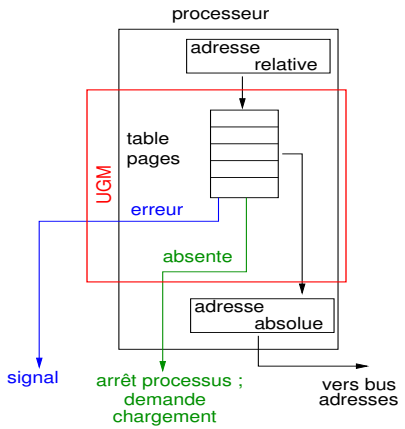
Un Point de la Situation

- Le système d'exploitation devra maintenir une table des pages par processus.
- Une page peut être absente de la MC. Dans ce cas, il faudra la ramener en mémoire lorsque nécessaire.

Question : Où doit être localisée la table des pages pour un processus en cours d'exécution ? Dans l'UGM, bien entendu, mais attention...

Ce qu'implique la pagination :
- Une table dynamique des pages est gérée par processus.
- Elle doit être chargée dans l'UGM avant l'exécution du processus.
- Pour chaque adresse relative, l'UGM va tester si la page est présente ; si oui, elle va effectuer la translation de l'adresse en page virtuelle vers l'adresse en page réelle ; sinon, il faudra ramener cette page en mémoire avant de continuer l'exécution du processus concerné.

Schéma de Fonctionnement



Questions :

- Comment signaler qu'une page est présente ou absente ?
- Comment est effectuée la translation d'une adresse relative virtuelle en adresse absolue réelle ?
- L'espace du processus étant découpé, est-il possible qu'une instruction ou une donnée soit à cheval sur deux pages ? Oui...
- Dans ce cas, que faire si la première page est présente et la deuxième absente ?

Absence/Présence

- Un booléen dans la table des pages permet de savoir si une page est présente ou absente.
- Cette table des pages est donc mise à jour par le gestionnaire de mémoire (système).
- Elle est chargée dans l'UGM lorsque le processus devient actif, avant le lancement de l'exéction, déchargée lorsqu'il perd l'UC.
- Elle est modifiée alors au fur et à mesure des transferts des pages MC↔espace d'echange.
- Elle est aussi modifiée lors des demandes d'allocation d'espace par le processus.

Codage des Adresses

Chaque adresse est divisée en deux parties :

numéro de page virtuelle	adresse dans cette page

On dit aussi *déplacement dans la page* (*offset* en anglais) pour la partie droite.

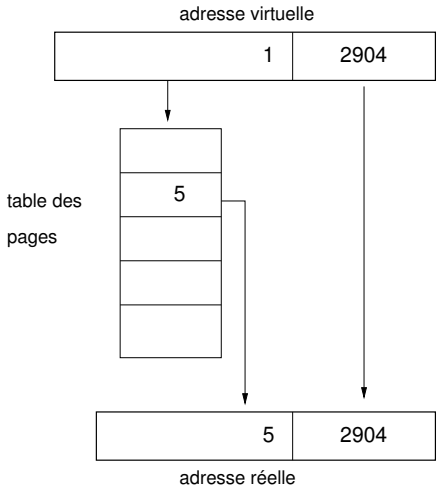
Exemple :

- Avec des pages de taille 4*Koctets*, une donnée à l'adresse virtuelle 7000 est dans la page virtuelle 1, avec un déplacement de 2904 ($4096 + 2904 = 7000$).
- Si les pages sont de 4*koctets*, on peut coder la valeur du déplacement sur 12 bits.

On aura par exemple avec des adresses virtuelles sur 40 bits (donc une mémoire virtuelle maximale de 1*TOctets*), 28 bits pour coder le numéro de page et 12 bits pour le déplacement dans la page.

Soit une capacité maximale de 256*Mpages* virtuelles.

Translation des Adresses

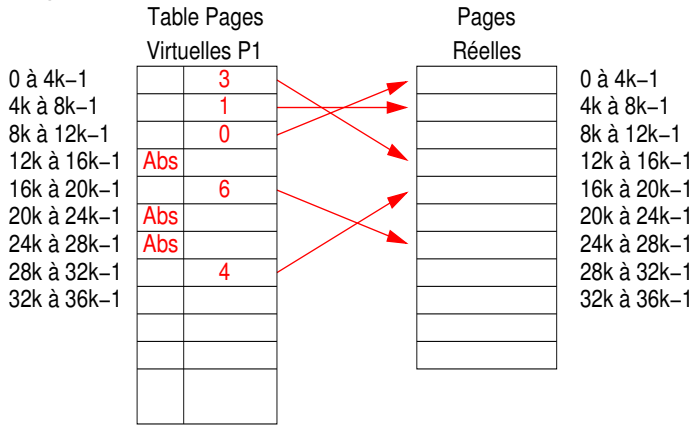


Seul le numéro de page virtuelle est translaté ; la translation est faite à partir de la table des pages.
En effet, les pages virtuelles et réelles sont de même taille, donc le déplacement est identique dans les deux.

Le codage des adresses virtuelles peut être fait sur une longueur sans liaison ni avec la taille du bus des adresses, ni avec la mémoire réelle disponible.

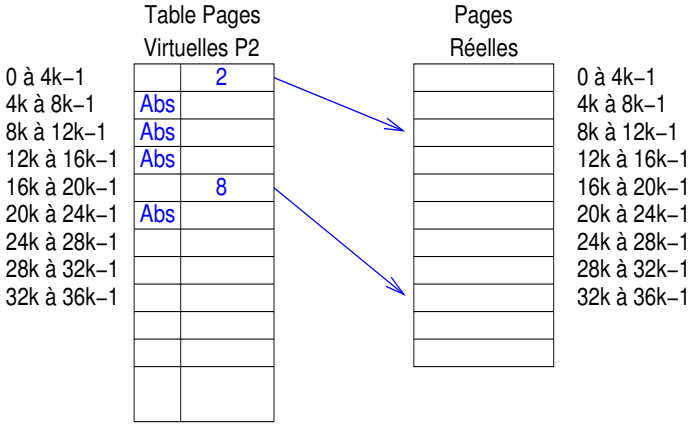
Représentation Globale

Une représentation globale d'un processus, des pages mémoire et de l'indicateur *présent/absent* :



Représentation Globale - Autre Processus

Pour un autre processus, P2, c'est une autre table de pages virtuelles, pointant forcément vers d'autres pages réelles, sauf si des pages de P1 ont été remplacées par des pages de P2.



Plan

Taille de la Table des Pages

Exemple : Un processus demandant 4*Goctets* de mémoire, va demander une table de 2^{20} éléments !

Et il faudra charger cette table dans l'UGM avant toute exécution ? !

Exercice1 : Proposer une solution (voir TD).

Exercice2 : On veut connaître la taille des pages allouées par le système. Proposer les principes d'une solution. Le programme correspondant n'est pas difficile à élaborer.

Défaut de Page

Des situations délicates peuvent se produire lors de l'exécution d'un processus :

- Après exécution d'une instruction, l'instruction suivante est dans une page absente ;
- pire, une instruction peut chevaucher deux pages, la première présente, l'autre absente ; elle peut aussi être lancée et nécessiter des données d'une page absente.

Le processus actif ne peut plus continuer l'exécution pour une de ces raisons, et un mécanisme de *défaut de page* est lancé.
Hic : dans les situations comme une interruption, l'instruction en cours est terminée. Ou alors, il s'agit d'une instruction qui n'est jamais terminée : une exception (divison par zéro, . . .), une sortie de l'espace mémoire. . .
Là, on se retrouve donc dans une situation pouvant être plus embêtante : un démarrage sans pouvoir finir.

Déroulement sur Défaut de Page

Lorsqu'un défaut de page est constaté, si l'instruction en cours n'est pas terminée, l'état courant (du processeur) est sauvegardé, pour relancer l'instruction. L'UGM interromp le processeur et l'exécution du gérant provoque le déroulement suivant :

1. Le processus demandant la page manquante est *bloqué*.
2. Le gestionnaire de mémoire (système) doit localiser la page demandée en mémoire secondaire (disque).
3. La page manquante est chargée en mémoire, provoquant probablement le déchargement d'une autre page.
4. **Les** tables des pages pour **les** processus concernés sont mises à jour.
5. Le processus demandeur est passé à l'état *prêt*.

Plan

Politiques de Gestion des Pages

Plusieurs politiques de gestion des pages sont étudiées. En effet, lorsqu'une page est absente, il faut la charger donc probablement décharger une page présente. Le problème fondamental est :

Quelles pages actuellement présentes doivent être déchargées, afin de rendre ce mécanisme le moins pénalisant possible ?

Chaque politique a ses avantages et inconvénients. La bibliographie est bien fournie sur ce domaine.

Plan

Appels de Bas et Haut Niveaux

Les appels système de bas niveau consistent à demander le déplacement du **point de rupture**, définissable comme étant la première adresse non utilisée dans l'espace d'adressage du processus.

Lorsque le point de rupture est à la fin de la page courante, le déplacement provoque une demande d'allocation d'une nouvelle page.

Deux appels système de bas niveau sont disponibles : `brk()` et `sbrk()`.

Les appels de plus haut niveau, `malloc()`, `calloc()`, `free()`, `realloc()` ou les opérateurs d'encore plus haut niveau `new`, `delete`, utilisent ces appels de base, plutôt inconfortables.