SYSTEMES INFORMATIQUES NFA003

Michel MEYNARD

PLAN

1. Introduction 1
2. Représentation de l'information 8
3. Structure des ordinateurs 15
4. La Couche Machine 42
5. Les Systèmes de Gestion de Fichiers 59
6. Les processus Unix 91
7. Conclusion 106

Bibliographie

[Tanenbaum 1988] Architecture de l'ordinateur, InterEditions

[De Blasi 1990] Computer Architecture, Addison-Wesley

[Peterson, Silberschatz 1985]Operating System Concepts, Addison-Wesley

1. Introduction

Définition d'un SE
Couche logicielle offrant une interface entre la machine matérielle et les utilisateurs.

Objectifs
- convivialité de l'interface (GUI/CUI);
- clarté et généricité des concepts (arborescence de répertoires et fichiers, droits des utilisateurs, ...);
- efficacité de l'allocation des ressources en temps et en espace;

Services
- multiprogrammé (ou multi-tâches) préemptif ;
- multi-utilisateurs ;
- fonctionnalités réseaux (partage de ressources distantes) ;
- communications réseaux (protocoles Internet) ;
- personnalisable selon l'utilisation (développeur, multimédia, SGBD, applications de bureau, …)

<table>
<tr><td>Utilisateur 1</td><td>u2</td><td>prog1</td></tr>
<tr><td>Shell</td><td>éditeur (appli)</td><td>compilateur</td></tr>
<tr><td colspan="3">noyau du SE</td></tr>
<tr><td colspan="3">matériel (hardware)</td></tr>
</table>

1.1 Historique

– A l'origine : machine énorme programmée depuis une console, beaucoup d'opérations manuelles ;
– développement de périphériques d'E/S (cartes 80 col., imprimantes, bandes magnétiques), développement logiciel : assembleur, chargeur, éditeur de liens, librairies, pilotes de périphérique.
– langages évolués (compilés); exemple de job : exécution d'un prg Fortran;
  – montage manuel de la bande magnétique contenant le compilateur F;
  – lecture du prg depuis le lecteur de cartes 80 col.
  – production du code assembleur sur une bande ;
  – montage de la bande contenant l'assembleur;
  – assemblage puis édition de lien produisant le binaire sur une bande;
  – chargement et exécution du prog.

Remarques :
– beaucoup d'interventions manuelles !
– sous-utilisation de l'UC ;
– machine à 2 millions de dollars réservée par créneaux d'1h !

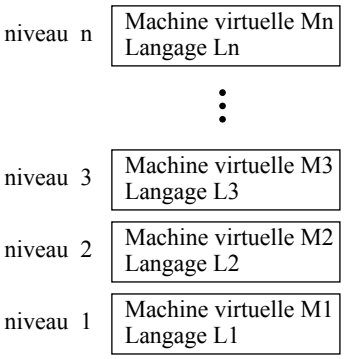
solution 1
– regroupement (batch) des opérations de même type;
– seuls les opérateurs manipulent la console et les périph. ;
– en cas d'erreur, dump mémoire et registres fourni au programmeur;

solution 2
– moniteur résidant en mémoire séquençant les jobs;
– cartes de ctrl ajoutées spécifiant la tâche à accomplir;
– définition d'un langage de commande des jobs (JCL) ancêtre des shell.

solution 3
– améliorer le moniteur pour en faire un SE multiprogrammé !
– stocker le SE sur disque dur et l'amorcer (bootstrap) depuis un moniteur résidant en ROM (le BIOS) ;

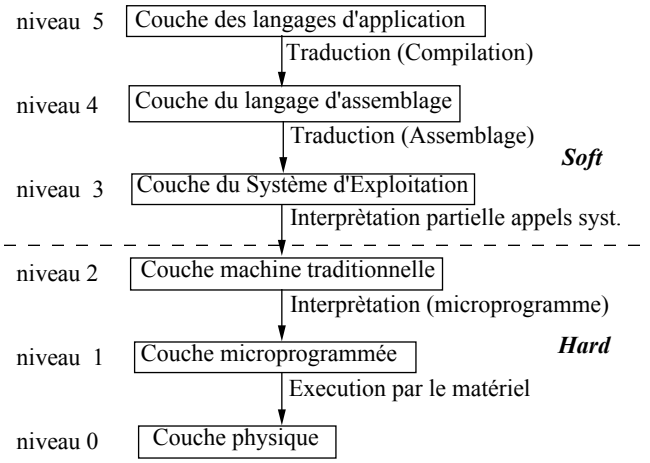
1.2 Principe de décomposition

Architecture multi-niveaux (ou couches)
Afin d'améliorer les performances, et en raison des impératifs technologiques, le jeu d'instructions des machines réelles est limité et primitif. On construit donc au-dessus, une série de couches logicielles permettant à l'homme un dialogue plus aisé.



Les programmes en L_i sont :
- soit **traduits** (compilés) en L_{i-1} ou L_{i-2} ou … L_1,
- soit **interprétés** par un interprèteur tournant sur L_{i-1} ou L_{i-2} ou … L_1

1.2.1 Décomposition des SI



niveaux composant (électronique) puis atomique (physique solide)

Vocabulaire : niveau ou langage ou machine

- 0 : portes logiques, circuits combinatoires, à mémoire ;
- 1 : une instruction machine (code binaire) interprétée par son microprogramme ;
- 2 : suite d'instructions machines du **jeu d'instructions**
- 3 : niveau 2 + ensemble des services offerts par le S.E. **(appels systèmes)** ;
- 4 : langage d'assemblage symbolique traduit en 3 par le programme assembleur ;
- 5 : langages évolués (de haut niveau) traduits en 3 par compilateurs ou alors interprétés par des programmes de niveau 3.

1.3 Matériel et Logiciel

Matériel (Hardware)
Ensemble des composants mécaniques et électroniques de la machine : processeur(s), mémoires, périphériques, bus de liaison, alimentation…

Logiciel (Software)
Ensemble des programmes, de quelque niveau que ce soit, exécutables par un ou plusieurs niveaux de l'ordinateur. Un programme = mot d'un langage.
Le logiciel est immatériel même s'il peut être stocké physiquement sur des supports mémoires.

Matériel et Logiciel sont conceptuellement équivalents

Toute opération effectuée par logiciel peut l'être directement par matériel et toute instruction exécutée par matériel peut être simulée par logiciel.

Le choix est facteur du coût de réalisation, de la vitesse d'exécution, de la fiabilité requise, de l'évolution prévue (maintenance), du délai de réalisation du matériel…

Exemples de répartition matériel/logiciel

- premiers ordinateurs : multiplication, division, manip. de chaînes, commutation de processus … par logiciel : actuellement descendus au niveau matériel ;
- à l'inverse, l'apparition des processeurs micro-programmés à fait remonter d'un niveau les instructions machines ;
- les processeurs RISC à jeu d'instructions réduit ont également favorisé la migration vers le haut ;
- machines spécialisées (Lisp, bases de données) ;
- Conception Assistée par Ordinateur : prototypage de circuits électroniques par logiciel ;
- développement de logiciels destinés à une machine matérielle inexistante par simulation (contrainte économique fondamentale).

La frontière entre logiciel et matériel est très mouvante et dépend fortement de l'évolution technologique (de même pour les frontières entre niveaux).

A chaque niveau, le programmeur communique avec une **machine virtuelle** sans se soucier des niveaux inférieurs.

2. Représentation de l'information

2.1 Introduction

La technologie passée et actuelle a consacrée les circuits mémoires (électroniques et magnétiques) permettant de stocker des données sous forme **binaire**.

Remarque :
des chercheurs ont étudié et continuent d'étudier des circuits ternaires et même décimaux…

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

les bits sont généralement stockés séquentiellement et sont conventionnellement numérotés de la façon suivante :
b_{n-1} b_{n-2} … b_2 b_1 b_0
1 0 … 0 1 1

On regroupe ces bits par paquets de n qu'on appelle des **quartets** (n=4), des **octets** (n=8) « *byte* », ou plus généralement des **mots** de n bits « *word* ».

Poids fort et faible
La longueur des mots étant la plupart du temps 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.

Exemple : mot de 16 bits
b_{15} b_{14} … b_8 b_7 b_6 … b_0
octet le plus significatif octet le moins significatif
Most Signifant Byte *Least Significant Byte*

Unités multiples
Ces mots sont eux-mêmes groupés et on utilise fréquemment les unités multiples de l'octet suivantes :

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 ≅ 10^9 octets noté 1 **Go**

1 Téra-octet = 2^{40} octets ≅ 10^{12} octets noté 1 **To**

2.2 Représentation des entiers positifs

2.2.1 Représentation en base 2

Un mot de n bits permet de représenter 2^n configurations différentes. En base 2, ces 2^n configurations sont associées aux entiers positifs x compris dans l'intervalle [0, 2^n-1] de la façon suivante :

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

Ainsi, un quartet permet de représenter l'intervalle [0, 15], un octet [0, 255], un mot de 16 bits [0, 65535].

Exemples :
00…0 représente 0
000…001 représente 1
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).

2.2.2 Représentation en base 2^p

La représentation d'un entier x en base 2,
$b_{n-1}b_{n-2}\ldots b_0$, est lourde en écriture. Aussi lui préfère-t-on une représentation plus compacte en base 2^p. On obtient cette représentation en découpant le mot $b_{n-1}b_{n-2}\ldots b_0$ 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.

Usuellement, 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. Généralement, on suffixe le nombre hexa par un H. De plus, on le préfixe par un 0 lorsque le symbole le plus à gauche est une lettre.

Exemples :
x=200 et n=8
en binaire : 11 001 000 (128+64+8)
en octal : 3 1 0 (3*64+8)
en hexadécimal : C 8 (12*16+8) : 0C8H

2.3 Représentation des caractères

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

Utilisation
– entrées/sorties
– représentation externe (humaine) des programmes (sources) et données y compris les données numériques

Code ou jeu de car.
Ensemble de caractères associés **aux mots binaires** les représentant. La quasi-totalité des codes ont une taille fixe (7 ou 8 ou 16 bits) afin de faciliter les recherches dans les mémoires machines.

– ASCII (7) ;
– EBCDIC (8) ;
– ISO 8859-1 ou ISO Latin-1 (8)
– UniCode (16).

2.3.1 Jeux de caractères

Code 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) et 96 caractères imprimables (32..127).

Codes de contrôle importants :
0 Null <CTRL> <@>; 3 End Of Text <CTRL> <c>;
4 End Of Transmission <CTRL> <d>; 7 Bell
8 Backspace 10 Line Feed 12 Form Feed
13 Carriage Return 27 Escape <CTRL> <z>

Codes imprimables importants :
20H Espace; 30H..39H "0".."9"; 41H..5AH "A".."Z"
61H..7AH "a".."z"

Le code ASCII est normalisé par le CCITT (Comité Consultatif International Télégraphique et Téléphonique) qui a prévu certaines variantes nationales : {} aux USA donnent éè en France …

La plupart du temps, l'unité mémoire étant l'octet, le 8° bit est utilisé :
- soit pour détecter les erreurs de transmission (bit de parité)
- soit pour définir des caractères spéciaux (semi-graphiques, accents) dans un code ASCII "étendu".

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

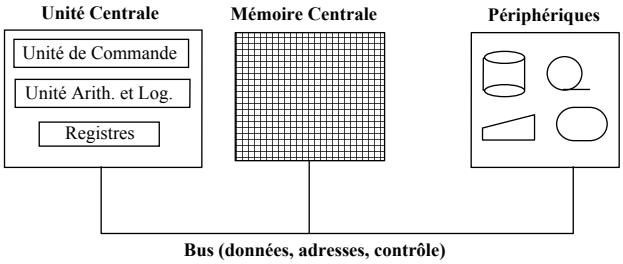
ISO-8859-1

Hexa	LSD															
MSD	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	×	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	÷	ø	ù	ú	û	ü	ý	þ	ÿ

3. Structure des ordinateurs

Le modèle d'architecture de la plupart des ordinateurs actuels provient d'un travail effectué par John **Von Neumann** en 1946.

Le modèle de Von Neumann



Principes du modèle de programmation :
code = **séquence** d'instructions en MC ;
données stockées en MC ;

Actuellement, d'autres types d'architecture (5° génération, machines systoliques, …) utilisant massivement le **parallélisme** permettent d'améliorer notablement la vitesse des calculs.

On peut conjecturer que dans l'avenir, d'autres paradigmes de programmation **spécifiques** à certaines applications induiront de nouvelles architectures.

3.1 L' Unité Centrale (UC)

ou processeur (*Central Processing Unit CPU)*

Cerveau de l'ordinateur, l'UC exécute séquentiellement les instructions stockées en Mémoire Centrale. Le traitement d'une instruction se décompose en 3 temps : **chargement, décodage, exécution**. C'est l'unité de commande qui ordonnance l'ensemble, tandis que l'UAL exécute des opérations telles que l'addition, la rotation, la conjonction…, dont les paramètres et résultats sont stockés dans les registres (mémoires rapides).

Les registres

Certains registres spécialisés jouent un rôle particulièrement important. Le Compteur Ordinal (CO) *Instruction Pointer IP, Program Counter PC* pointe sur la prochaine instruction à exécuter; le Registre Instruction (RI) contient l'instruction en cours d'exécution; le registre d'état *Status Register, Flags, Program Status Word PSW* contient un certain nombre d'indicateurs (ou drapeaux ou bits) permettant de connaître et de contrôler certains états du processeur; Le pointeur de pile *Stack Pointer* permet de mémoriser l'adresse en MC du sommet de pile (structure de données *Last In First Out LIFO* indispensable pour les appels procéduraux); Des registres d'adresse (index ou bases) permettent de stocker les adresses des données en mémoire centrale tandis que des registres de travail permettent de stocker les paramètres et résultats de calculs.

L'Unité de Commande

Celle-ci exécute l'algorithme suivant :

répéter

1. **charger** dans RI l'instruction stockée en MC à l'adresse pointée par le CO;
2. CO:=CO+taille(instruction en RI);
3. **décoder** (RI) en micro-instructions;
4. (localiser en mémoire les données de l'instruction;)
5. (charger les données;)
6. **exécuter** l'instruction (suite de micro-instructions);
7. (stocker les résultats mémoires;)

jusqu'à l'infini

Lors du démarrage de la machine, CO est initialisé soit à l'adresse mémoire 0 soit à l'adresse correspondant à la fin de la mémoire (2^m-1). A cette adresse, se trouve le moniteur en mémoire morte qui tente de charger l'amorce "*boot-strap*" du système d'exploitation.

Remarquons que cet algorithme peut parfaitement être simulé par un logiciel (interprèteur). Ceci permet de tester des processeurs matériels avant même qu'il en soit sorti un prototype, ou bien de simuler une machine X sur une machine Y (émulation).

L'Unité Arith. et Log.

Celle-ci exécute des opérations :

arithmétiques : addition, soustraction, C_2, incrémentation, décrémentation, multiplication, division, décalages arithmétiques (multiplication ou division par 2^n).

logiques : et, ou, xor, non, rotations et décalages.

Selon le processeur, certaines de ces opérations sont présentes ou non. De plus, les opérations arithmétiques existent parfois pour plusieurs types de nombres (C2, DCB, virgule flottante) ou bien des opérations d'ajustement permettent de les réaliser.

Enfin sur certaines machines (8086) ne possédant pas d'opérations en virgule flottante, des co-processeurs arithmétiques (8087) peuvent être adjoint pour les réaliser.

Les opérations arithmétiques et logiques positionnent certains indicateurs d'état du registre PSW. C'est en testant ces indicateurs que des branchements conditionnels peuvent être exécutés vers certaines parties de programme.

Pour accélérer les calculs, on a intérêt à utiliser les registres de travail comme paramètres, notamment l'accumulateur (AX pour le 8086).

3.2 La Mémoire Centrale (MC)

La mémoire centrale de l'ordinateur est habituellement constituée d'un ensemble ordonné de 2^m cellules (cases), chaque cellule contenant un mot de n bits. Ces mots permettent de conserver programmes et données ainsi que la pile d'exécution.

3.2.1 Accès à la MC

La MC est une mémoire électronique et l'on accède à n'importe laquelle de ses cellules au moyen de son adresse comprise dans l'intervalle [0, 2^m-1]. Les deux types d'accès à la mémoire sont :

la **lecture** qui transfère sur le bus de données, le mot contenu dans la cellule dont l'adresse est située sur le bus d'adresse.

l'**écriture** qui transfère dans la cellule dont l'adresse est sur le bus d'adresse, le mot contenu sur le bus de données.

La taille n des cellules mémoires ainsi que la taille m de l'espace d'adressage sont des caractéristiques fondamentales de la machine. Le mot de n bits est la plus petite unité d'information transférable entre la MC et les autres composants. Généralement, les cellules contiennent des mots de 8, 16 ou 32 bits.

3.2.2 Contenu/adresse (valeur/nom)

Attention à ne jamais confondre le contenu d'une cellule, mot de n bits, et l'adresse de celle-ci, mot de m bits même lorsque n=m.

Parfois, le bus de données a une taille multiple de n ce qui permet la lecture ou l'écriture de plusieurs mots consécutifs en mémoire. Par exemple, le microprocesseur 8086 permet des échanges d'octets ou de mots de 16 bits (appelés "mots").

Exemple (cellules d'un octet)

Adresse	Contenu bin								hexa.
0	0	1	0	1	0	0	1	1	53H
1	1	1	1	1	1	0	1	0	0FAH
2	1	0	0	0	0	0	0	0	80H
…									
32	0	0	1	0	0	0	0	0	20H (32_{10})
…									
2^m-1	0	0	0	1	1	1	1	1	1FH

Parfois, une autre représentation graphique de l'espace mémoire est utilisé, en inversant l'ordre des adresses : adresses de poids faible en bas, adresses fortes en haut. Cependant, pour le 8086, lorsqu'on range un mot (16 bits) à l'adresse mémoire i, l'octet de poids fort se retrouve en i+1. Il est aisé de s'en souvenir mnémotechniquement via la gravité dans les liquides de densités différentes.

3.2.3 RAM et ROM

Random Access Memory/ Read Only Memory

La RAM est un type de mémoire électronique **volatile** et **réinscriptible**. Elle est aussi nommée mémoire vive et plusieurs technologies permettent d'en construire différents sous-types : **statique**, **dynamique** (rafraîchissement). La RAM constitue la majeure partie de l'espace mémoire puisqu'elle est destinée à recevoir programme, données et pile d'exécution.

La ROM est un type de mémoire électronique **non volatile** et **non réinscriptible**. Elle est aussi nommée mémoire morte et plusieurs technologies permettent d'en construire différents sous-types (ROM, PROM, EPROM, EEPROM (Flash), …). La ROM constitue une faible partie de l'espace mémoire puisqu'elle ne contient que le moniteur réalisant le chargement du système d'exploitation et les Entrées/Sorties de plus bas niveau. Sur les PCs ce moniteur s'appelle le *Basic Input Output System*. Sur les Macintosh, le moniteur contient également les routines graphiques de base. C'est toujours sur une adresse ROM que le Compteur Ordinal pointe lors du démarrage machine.

DDR SDRAM : Double Data Rate Synchronous Dynamic RAM est une RAM dynamique (condensateur) qui a un *pipeline* interne permettant de synchroniser les opérations R/W.

3.3 Les périphériques

Les périphériques, ou organes d'Entrée/Sortie (E/S) *Input/Output (I/O)*, permettent à l'ordinateur de **communiquer** avec l'homme ou d'autres machines, et de **mémoriser massivement** les données ou programmes dans des fichiers. La caractéristique essentielle des périphériques est leur **lenteur** :

Processeur cadencé en Giga-Hertz : instructions éxécutées chaque nano-seconde (10^{-9} s) ;

Disque dur de temps d'accès entre 10 et 20 ms (10^{-3} s) : rapport de 10^7 !

Clavier avec frappe à 10 octets par seconde : rapport de 10^8 !

3.3.1 Communication

L'ordinateur échange des informations avec l'**homme** à travers des terminaux de communication homme/machine : clavier ←, écran →, souris ←, imprimante →, synthétiseur (vocal) →, table à digitaliser ←, scanner ←, crayon optique ←, lecteur de codes-barres ←, lecteur de cartes magnétiques ←, terminaux consoles stations ↔ …

Il communique avec d'autres machines par l'intermédiaire de **réseaux** ↔ locaux ou longue distance (via un modem).

3.3.2 Mémorisation de masse

ou mémorisation secondaire

Les mémoires électroniques étant chères et soit volatiles (non fiables) soit non réinscriptibles, le stockage de masse est réalisé sur d'autres supports. Ces autres supports sont caractérisés par :

- non volatilité et réinscriptibilité
- faible prix de l'octet stocké
- lenteur d'accès et modes d'accès (séquentiel, séquentiel indexé, aléatoire, …)
- forte densité
- parfois amovibilité
- *Mean Time Between Failures* plus important car organes mécaniques → **stratégie de sauvegarde**

Supports Optiques

Historiquement, les **cartes 80 colonnes** ont été parmi les premiers supports mais sont complètement abandonnées aujourd'hui. Les **rubans perforés**, utilisés dans les milieux à risque de champ magnétique (Machines Outils à Commande Numérique), sont leurs descendants directs dans le cadre des supports optiques.

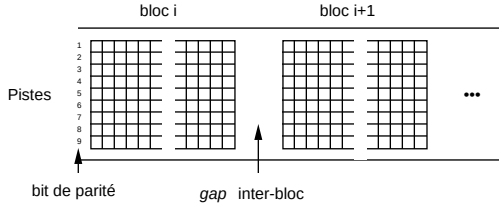
Supports Magnétique

Aujourd'hui (années 90), les supports magnétiques constituent la quasi-totalité des mémoires de masse des ordinateurs à usage général.

Bandes magnétiques

Ce sont des supports à accès **séquentiel** particulièrement utilisés dans la sauvegarde. On les utilise de plus en plus sous forme de **cassettes**. Les dérouleurs de cassettes sont appelés *streamers*. Les densités et vitesses sont variables : quelques milliers de *bytes per inch (bpi)* et autour d'un Mo/s. Le temps d'accès dépendant de la longueur de la bande …

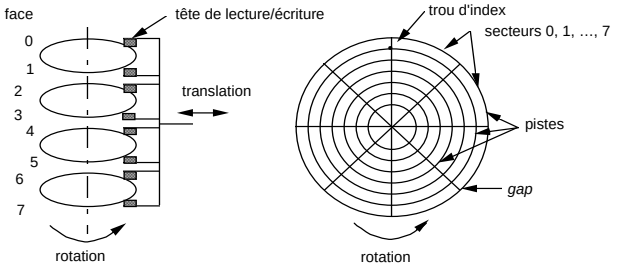
schéma d'une bande magnétique



Disques magnétiques

Ils constituent la majorité des mémoires secondaires. Durs ou souples, fixes ou amovibles, solidaires (Winchester) ou non (dispack) de leurs têtes de lecture/écriture, il en existe une très grande diversité. Les disques durs ont des capacités variant entre quelques dizaines à quelques centaines de Mo, des vitesses de transfert autour du Mo/s et des temps d'accès approchant les 10 ms.

schéma d'un disque magnétique



Composé de faces, de pistes concentriques, de secteurs *"soft sectored"*, la densité des disques est souvent caractérisée par le nombre de *"tracks per inch" (tpi)* . Sur les disques durs, un

cylindre est constitué d'un ensemble de pistes de même diamètre.
Un contrôleur de disque (carte) est chargé de transférer les informations entre un ou plusieurs secteurs et la MC. Pour cela, il faut lui fournir : le sens du transfert *(R/W)*, l'adresse de début en MC (Tampon), la taille du transfert, la liste des adresses secteurs (n° face, n° cylindre, n° secteur). La plus petite **unité de transfert** physique est **1 secteur**.

Pour accéder à un secteur donné, le contrôleur doit commencer par translater les bras mobiles portes-têtes sur le bon cylindre, puis attendre que le bon secteur passe sous la tête sélectionnée pour démarrer le transfert. Le **temps d'accès** moyen caractérise la somme de ces deux délais moyens.

Exemple : TA_{moyen} et transfert d'1 secteur ?
disque dur 8 faces, 50 cylindres, 10 secteurs/piste d'1 Ko, tournant à 3600 tours/mn, ayant une vitesse de translation de 1 m/s et une distance entre la 1° et la dernière piste de 5 cm.

TA_{moyen} = (5 cm/2)/1 m/s + (1/60 t/s)/2= 25ms+8,3 ms
Transfert d'1 secteur = (1/60)/10 = 1,66 ms

Supports optiques

Supports optiques

unités de lecture fonctionnant au moyen d'un faisceau laser ;
densités de stockage supérieures au magnétique : de 10^2 à 10^4 fois plus ;
temps d'accès plus longs : archivage de masse.

CDROM : disques compacts (même format que les CDs audio) pré-enregistrés par pressage en usine et non réinscriptibles : (logiciels, annuaires, encyclopédies, …).

CDR : inscriptibles une seule fois ;

CDRW : réinscriptibles (1000 fois)

DVDR, DVDRW : idem CD mais avec des capacités plus importantes : 4,7 Go contre 700 Mo, double couches ...

Magnéto-optiques : combinant la technologie optique (laser) et magnétique (particules orientées), ils sont réinscriptibles. Amovibles, plus denses que les disques magnétiques mais moins rapides, ils constituent un compromis pour les archivages et les fichiers rarement accédés.

3.4 Contrôleur d'E/S et IT

A l'origine, l'UC gérait les périphériques en leur envoyant une requête puis en attendant leur réponse. Cette **attente active** était supportable en environnement monoprogrammé.

Actuellement, l'UC délègue la gestion des E/S aux processeurs situés sur les cartes contrôleur (disque, graphique, …) :
l'UC transmet la requête d'un processus à la carte contrôleur ;
l'UC « endort » le processus courant et exécute un processus « prêt » ;
le contrôleur exécute l'E/S ;
le contrôleur prévient l'UC de la fin de l'E/S grâce au mécanisme **d'interruption** ;
l'UC désactive le processus en cours d'exécution puis « réveille » le processus endormi qui peut continuer à s'exécuter.

Grâce à ce fonctionnement, l'UC ne perd pas son temps à des tâches subalternes !

Généralement, plusieurs **niveaux d'interruption** plus ou moins prioritaires sont admis par l'UC

3.5 Le(s) Bus

Le **bus de données** est constitué d'un ensemble de lignes bidirectionnelles sur lesquelles transitent les bits des données lues ou écrites par le processeur. (Data 0-31)

Le **bus d'adresses** est constitué d'un ensemble de lignes unidirectionnelles sur lesquelles le processeur inscrit les bits formant l'adresse désirée.

Le **bus de contrôle** est constitué d'un ensemble de lignes permettant au processeur de signaler certains événements et d'en recevoir d'autres. On trouve fréquemment des lignes représentant les **signaux** suivants :

V_{cc} et GROUND : tensions de référence ←
$\overline{RESET}$: réinitialisation de l'UC ←
R/$\overline{W}$: indique le sens du transfert →
MEM/$\overline{IO}$: adresse mémoire ou E/S →

Technologiquement, les bus de PC évoluent rapidement (Vesa Local Bus, ISA, PCI, PCI Express, ATA, SATA, SCSI, …)

3.6 Améliorer les performances

3.6.1 Hiérarchie mémoire et Cache

Classiquement, il existe 3 niveaux de mémoire ordonnés par vitesse d'accès et prix décroissant et par taille croissante :
Registres ;
Mémoire Centrale ;
Mémoire Secondaire.

Afin d'accélérer les échanges, on peut augmenter le nombre des niveaux de mémoire en introduisant des **CACHE** ou antémémoire de Mémoire Centrale et/ou Secondaire : Mémoire plus rapide mais plus petite, contenant une copie de certaines données :

Fonctionnement :
Le demandeur demande à lire ou écrire une information ;
si le cache possède l'information, l'opération est réalisée, sinon, il récupère l'info. depuis le fournisseur puis réalise l'op.

Le principe de **séquentialité** des instructions et des structures de données permet d'optimiser le chargement du cache avec des segments de la mémoire fournisseur. Stratégie de remplacement est généralement LRU (Moins Récemment Utilisée).

3.6.1.1 Niveaux et localisation des caches

Cache Processeur réalisé en SRAM

- Niveau 1 (L1) : séparé en 2 caches (instructions, données), situé dans le processeur, communique avec L2 ;

- Niveau 2 (L2) : unique (instructions et données) situé dans le processeur ;
- Niveau 3 (L3) : existe parfois sur certaines cartes mères.

Par exemple, Pentium 4 ayant un cache L2 de 256 Ko.

Cache Disque

De quelques Méga-octets, ce cache réalisé en DRAM est géré par le processeur du contrôleur disque. Il ne doit pas être confondu avec les tampons systèmes stockés en mémoire centrale (100 Mo). Intérêts de ce cache :

Lecture en avant (arrière) du cylindre ;
Synchronisation avec l'interface E/S (IDE, SATA, …)
Mise en attente des commandes (SCSI, SATA)

3.6.2 Pipeline

Technique de conception de processeur avec plusieurs petites unités de commande placées en série et dédiées à la réalisation d'une tâche spécifique. Plusieurs instructions se chevauchent à l'intérieur même du processeur.

Par exemple, décomposition simple d'une instruction en 4 étapes :

Fetch : chargement de l'instruction depuis la MC ;
Decode : décodage en micro-instructions ;
Exec : exécution de l'instruction (UAL) ;
Write Back : écriture du résultat en MC ou dans un registre

Soit la séquence d'instruction : i1, i2, i3, ...

sans pipeline :

i1F, i1D, i1E, i1W, i2F, i2D, i2E, i2W, i3F, ...

avec pipeline à 4 étages

i1F	i1D	i1E	i1W	
	i2F	i2D	i2E	I2W
		i3F	i3D	I3E
			i4F	i4D

Chaque étage du pipeline travaille « à la chaîne » en répétant la même tâche sur la série d'instructions qui arrive.

Si la séquence est respectée, et s'il n'y a pas de conflit, le débit d'instructions (*throughput*) est multiplié par le nombre d'étages !

Problèmes et solutions :
Rupture de séquence : vidage des étages !
Dépendance d'instructions : mise en attente forcée

Architecture superscalaire :
Plusieurs pipeline (2) dans le même processeur travaillent en parallèle (→instons indépendantes).

Exemple :
Pentium 4 : selon ses versions, de 20 à 31 étages !

3.6.3 SIMD

Single Instruction Multiple Data désigne un ensemble d'instructions vectorielles permettant des opérations scientifiques ou multimédia. Par exemple, l'AMD 64 possède 8 registres 128 bits et des instructions spécifiques utilisables pour le streaming, l'encodage audio ou vidéo, le calcul scientifique.

3.6.4 DMA et BUS Mastering

L'accès direct mémoire ou DMA (Direct Memory Access) est un procédé informatique où des données circulant de ou vers un périphérique (port de communication, disque dur) sont transférées directement par un contrôleur adapté vers la mémoire centrale de la machine, sans intervention du

microprocesseur si ce n'est pour initier et conclure le transfert. La conclusion du transfert ou la disponibilité du périphérique peuvent être signalés par interruption.

La technique de Bus Mastering permet à n'importe quel contrôleur de demander et prendre le contrôle du bus : le maître peut alors communiquer avec n'importe lequel des autres contrôleurs sans passer par l'UC.
Cette technique implémentée dans le bus PCI permet à n'importe quel contrôleur de réaliser un DMA. Si l'UC a besoin d'accéder à la mémoire, elle devra attendre de récupérer la maîtrise du bus. Le contrôleur maître lui vole alors des cycles mémoires.

3.7 Les périphériques

Les périphériques, ou organes d'Entrée/Sortie (E/S) *Input/Output (I/O)* , permettent à l'ordinateur de **communiquer** avec l'homme ou d'autres machines, et de **mémoriser massivement** les données ou programmes dans des fichiers. La caractéristique essentielle des périphériques est leur **lenteur**.

En effet, sur les micro-ordinateurs travaillant à quelques dizaines de Méga-Hertz, le cycle d'un accès mémoire est de quelques dizaines de nano-secondes (10^{-9} s). En revanche, une communication à 10^4 bits/s permet d'échanger 1 octet/milliseconde, et un disque dur a un temps d'accès de quelques dizaines de ms pour un Ko. Le rapport d'échelle se situe donc entre 10^3 et 10^5 (voire beaucoup plus pour les imprimantes) pour l'accès à un octet.

3.7.1 Communication

L'ordinateur échange des informations avec l'**homme** à travers des terminaux de communication homme/machine : clavier ←, écran →, souris ←, imprimante →, synthétiseur (vocal) →, table à digitaliser ←, scanner ←, crayon optique ←, lecteur de codes-barres ←, lecteur de cartes magnétiques ←, terminaux consoles stations ↔ …
Il communique avec d'autres machines par l'intermédiaire de **réseaux** ↔ locaux ou longue distance (via un modem).

3.7.2 Mémorisation de masse
ou mémorisation secondaire

Les mémoires électroniques étant chères et soit volatiles (non fiables) soit non réinscriptibles, le stockage de masse est réalisé sur d'autres supports. Ces autres supports sont caractérisés par :
- non volatilité et réinscriptibilité
- faible prix de l'octet stocké
- lenteur d'accès et modes d'accès (séquentiel, séquentiel indexé, aléatoire, …)
- forte densité
- parfois amovibilité
- *Mean Time Between Failures* plus important car organes mécaniques → **stratégie de sauvegarde**

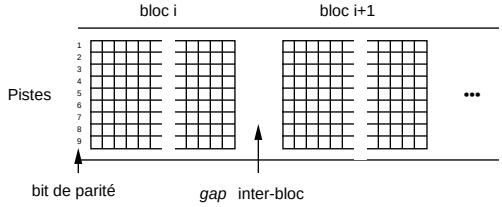
Supports Optiques
Historiquement, les **cartes 80 colonnes** ont été parmi les premiers supports mais sont complètement abandonnées aujourd'hui. Les **rubans perforés**, utilisés dans les milieux à risque de champ magnétique (Machines Outils à Commande Numérique), sont leurs descendants directs dans le cadre des supports optiques.

Supports Magnétique
Aujourd'hui (années 90), les supports magnétiques constituent la quasi-totalité des mémoires de masse des ordinateurs à usage général.

Bandes magnétiques

Ce sont des supports à accès **séquentiel** particulièrement utilisés dans la sauvegarde. On les utilise de plus en plus sous forme de **cassettes**. Les dérouleurs de cassettes sont appelés *streamers*. Les densités et vitesses sont variables : quelques milliers de *bytes per inch (bpi)* et autour d'un Mo/s. Le temps d'accès dépendant de la longueur de la bande …

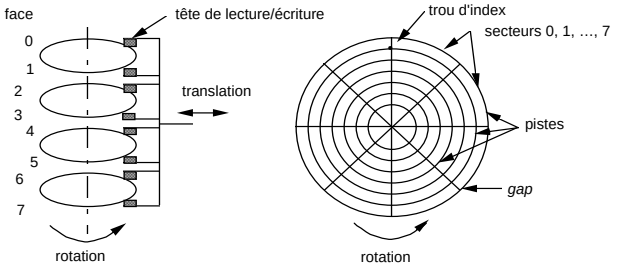
schéma d'une bande magnétique



Disques magnétiques

Ils constituent la majorité des mémoires secondaires. Durs ou souples, fixes ou amovibles, solidaires (Winchester) ou non (dispack) de leurs têtes de lecture/écriture, il en existe une très grande diversité. Les disques durs ont des capacités variant entre quelques dizaines à quelques centaines de Mo, des vitesses de transfert autour du Mo/s et des temps d'accès approchant les 10 ms.

schéma d'un disque magnétique



Composé de faces, de pistes concentriques, de secteurs *"soft sectored"*, la densité des disques est souvent caractérisée par le nombre de *"tracks per inch" (tpi)* . Sur les disques durs, un

cylindre est constitué d'un ensemble de pistes de même diamètre.
Un contrôleur de disque (carte) est chargé de transférer les informations entre un ou plusieurs secteurs et la MC. Pour cela, il faut lui fournir : le sens du transfert *(R/W)*, l'adresse de début en MC (Tampon), la taille du transfert, la liste des adresses secteurs (n° face, n° cylindre, n° secteur). La plus petite **unité de transfert** physique est **1 secteur**.

Pour accéder à un secteur donné, le contrôleur doit commencer par translater les bras mobiles portes-têtes sur le bon cylindre, puis attendre que le bon secteur passe sous la tête sélectionnée pour démarrer le transfert. Le **temps d'accès** moyen caractérise la somme de ces deux délais moyens.

Exemple : TA_{moyen} et transfert d'1 secteur ?
disque dur 8 faces, 50 cylindres, 10 secteurs/piste d'1 Ko, tournant à 3600 tours/mn, ayant une vitesse de translation de 1 m/s et une distance entre la 1° et la dernière piste de 5 cm.

TA_{moyen} = (5 cm/2)/1 m/s + (1/60 t/s)/2= 25ms+8,3 ms
Transfert d'1 secteur = (1/60)/10 = 1,66 ms

Nouveaux supports optiques et magnéto-optiques

De nouveaux produits tels que les vidéodisques (images), les disques optiques numériques et les disques magnéto-optiques sont "récemment" apparus sur le marché. Ils sont caractérisés par des densités de stockage supérieures au magnétique : de 10^2 à 10^4 fois plus; mais aussi par des temps d'accès plus longs ! Par conséquent, ils servent essentiellement à l'archivage de masse.

CDROM : disques compacts (même format que les CDs audio) pré-enregistrés par pressage en usine et non réinscriptibles : (annuaires, encyclopédies, …). Les unités de lecture fonctionnent au moyen d'un faisceau laser.

CDWORM *(Write Once Read Many)* : les CDs peuvent être écrit une seule fois sur le site de l'utilisateur avec une unité de lecture/écriture à faisceau laser.

Magnéto-optiques : combinant la technologie optique (laser) et magnétique (particules orientées), ils sont réinscriptibles. Amovibles, plus denses que les disques magnétiques mais moins rapides, ils constituent un excellent compromis pour les archivages et les fichiers rarement accédés.

3.7.3 Hiérarchie de mémoire

Usuellement, il existe 3 niveaux de mémoire ordonnés par vitesse d'accès et prix décroissant et par taille croissante : registres, mémoire centrale, mémoire secondaire. Les machines et systèmes d'exploitation sont conçus afin de réguler et de minimiser les temps d'exécution de l'UC sur les données. Les données les moins utilisées sont donc stockées sur les mémoires les plus lentes et les plus grosses tandis que les plus utilisées restent le plus près possible de l'UC. La vitesse/prix et la taille étant deux paramètres conflictuels, différents types de solutions sont possibles :

1. Elargissement du bus de données : 8080 (8 bits), 8086 (16 bits), 80386 (32 bits).

2. introduction d'un niveau intermédiaire dans la hiérarchie de mémoire : les mémoires **cache** sont des RAM très rapides interposées entre l'UC et la MC. Utilisant divers algorithmes prédicatifs, ces caches ont comme fonction de contenir les instructions et données dont l'UC aura besoin.

3. *Prefetching* (pré-chargement) : Dans l'algorithme déroulé par l'Unité de Commande de l'UC, on parallèlise le traitement de l'exécution courante avec le chargement de l'instruction suivante.

3.7.4 Canaux ou Processeurs d'E/S et IT

Différents boîtiers électroniques sont conçus afin de piloter certains périphériques selon certaines **normes** (ou interfaces) V24, Centronics, SCSI…. Ces processeurs d'E/S sont commandés par l'UC qui leur confie l'exécution de tâches spécialisées. Les commandes peuvent être fort simple mais aussi donner lieu à un programme canal plus complexe.

Ces tâches peuvent engendrer une **attente active** de la part de l'UC mais on préfère généralement désynchroniser les E/S et l'UC. Pour ce faire, on utilise majoritairement le mécanisme d'**Interruption** (IT) de l'UC qui permet de suspendre le traitement courant pour entreprendre l'exécution d'un programme prioritaire. Par exemple, après avoir lancé une E/S disque via un processeur d'E/S, l'UC peut exécuter un autre programme qui sera interrompu par le processeur d'E/S lorsque le transfert aura été effectué!

Généralement, plusieurs **niveaux d'interruption** plus ou moins prioritaires sont admis par l'UC. Par exemple, sur le 8086, 2 niveaux de priorités existent: les IT normales qui peuvent ne pas être prises en compte, et les IT **non masquables** qui déroutent tout programme en cours.

3.8 Le(s) Bus

Le **bus de données** est constitué d'un ensemble de lignes bidirectionnelles sur lesquelles transitent les bits des données lues ou écrites par l'UC.

Le **bus d'adresses** est constitué d'un ensemble de lignes unidirectionnelles sur lesquelles l'UC inscrit les bits formant l'adresse désirée. Selon les cas, cette adresse peut désigner une cellule mémoire ou bien un dispositif d'E\S.

Le **bus de contrôle** est constitué d'un ensemble de lignes permettant à l'UC de signaler certains événements et d'en recevoir d'autres. On trouve fréquemment des lignes représentant les **signaux** suivants :

V_{cc} et GROUND : tensions de référence ←
$\overline{\text{RESET}}$: réinitialisation de l'UC ←
R/$\overline{\text{W}}$: indique le sens du transfert →
MEM/$\overline{\text{IO}}$: adresse mémoire ou E/S →

4. La Couche Machine

Elle constitue le niveau **le plus bas** auquel l'utilisateur a accès.

Types d'ordinateurs (évolution constante !)

les ordinateurs personnels (PC ou Macintosh) ;
- les ordinateurs de bureau ;
- les ordinateurs portables ;
- les assistants personnels (ou PDA) ;

les moyens systèmes (midrange) (ex IBM AS/400-ISeries, RISC 6000...)

les mainframes (serveurs centraux) (ex. : IBM zSeries 64 bits, Siemens SR2000 et S110 ...) ;

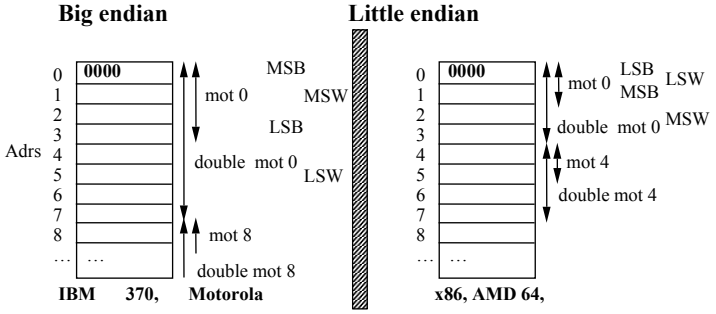
les superordinateurs (Blue Gene machine IBM utilisant 65536 Power PC réalisant 136,8 TFlops);

les stations de travail (PC puissants pour CAO, …) ;

4.1 Exemples

Pentium 4 (x86 CPU)
8 registres 32 bits (AL, AH, AX, EAX);
Espace adrs : 4 Go ;
Mémoire virtuelle segmentée (CS code, DS data, SS stack) avec un sélecteur de segment 16 bits + adrs virtuelle 32 bits ;
(x87) coprocesseur arithmétique flottante (FPU) 32, 64 ou 80 bits

AMD 64
16 registres 64 bits (AL, AH, AX, EAX, RAX)
Espace adrs : 256 To
Mémoire virtuelle non segmentée sur 64 bits
SIMD avec registres 128 bits



En Little endian, mnémotechniquement, la "gravité" est orientée vers le bas de la mémoire (adresses supérieures). Les Power PC d'IBM sont bi-endian.

4.2 Format des instructions

Programme = suite d'**instructions** machines

Une instruction est composée de plusieurs champs :
• 1 champ obligatoire : le **code opération** désigne le type d'instruction. Sa longueur est souvent **variable** afin d'optimiser la place et la vitesse utilisée par les instructions les plus fréquentes (Huffman).
• 0 à n champs optionnels : les **opérandes** désignent des données immédiates ou stockées dans des registres ou en MC. Le type de désignation de ces données est nommé **mode d'adressage**.

Exemple :

Code Opération	$Opérande_1$	$Opérande_2$

De plus, la taille des cases et mots mémoires doivent être des multiples de la taille d'un caractère afin d'éviter le **gaspillage de place** ou des **temps** de recherche prohibitifs. Les codes alphanumériques usuels étant sur 8 bits (EBCDIC) ou 7+1 bits (ASCII), c'est la raison du découpage des MC en octets.

Enfin, des adresses devant également être stockées en MC, c'est la raison pour laquelle la taille de l'espace d'adressage est généralement un multiple de 2^8 octets (64 Ko, 16 Mo, 4 Go). Sauf lorsque la technique d'adressage utilise des segments recouvrants : 8086 : 1Mo = 2^{20} o; 1 adrs = 2 mots.

4.3 Modes d'adressage

Remarque : pour simplifier, nous utiliserons dorénavant la notation mnémonique des instructions machines, notation alphanumérique qui correspond à la couche 4 du langage d'assemblage.

Types de donnée représentés par les opérandes :
- donnée **immédiate** stockée dans l'instruction machine
- donnée dans un **registre** de l'UC
- donnée à une **adresse** en MC

Nombres d'opérandes :
Nous supposerons un nombre maximal de 2 opérandes, ce qui représente le cas général. Souvent, un opérande implicite n'est pas désigné dans l'instruction : c'est l'**accumulateur** qui est un registre de travail privilégié; le Z80 a au maximum un opérande explicite (et A comme opérande implicite).

Source et Destination :
Lorsque 2 opérandes interviennent dans un transfert ou une opération arithmétique, l'un est source et l'autre destination de l'instruction. L'ordre d'apparition varie suivant le type d'UC :
IBM 370, 8086 : ADD DST, SRC *DST := DST+SRC*
PDP-11, 68000 : ADD SRC, DST *DST := DST+SRC*

4.3.1 Adressage immédiat

La valeur de la donnée est stockée dans l'instruction. Cette valeur est donc copiée de la MC vers l'UC lors de la phase de chargement (*fetch*) de l'instruction.

Avantage : pas d'accès supplémentaire à la MC
Inconvénient : taille limitée de l'opérande

Exemples : Z80
(1) ADD A, <n> ; Code Op. 8 bits, n sur 8 bits en C2
(2) LD <Reg>, <n> ; Code Op 5 b., Reg 3 b., n 8 b.

4.3.2 Adressage registre

La valeur de la donnée est stockée dans un registre de l'UC. La désignation du registre peut être explicite (2) ou implicite (1) (A sur Z80).

Avantage : accès rapide % MC
Inconvénient : taille limitée de l'opérande

Selon le nombre de registres de travail de l'UC, la taille du code des instructions varie :
8 registres nécessitent 3 bits (Z80)
16 registres nécessitent 4 bits(68000)

4.3.3 Adressage direct

La valeur de la donnée est stockée à une adresse en MC. C'est cette adresse qui est représentée dans l'instruction.

Avantage : taille quelconque de l'opérande
Inconvénient : accès mémoire supplémentaire, taille importante de l'instruction

Selon le type d'implantation mémoire requis par l'UC, plusieurs adressages directs peuvent coexister.

Exemples : 8086
MOV AL, <dis> ; déplacement intra-segment (court)
transfère dans le registre AL, l'octet situé à l'adresse dis dans le segment de données : dis est codé sur 16 bits, *Data Segment* est implicite.

ADD BX, <aa> ; adresse absolue aa= S,D (long)
ajoute à BX, le mot de 16 bits situé à l'adresse D dans le segment S : D et S sont codés sur 16 bits.

L'IBM 370 n'a pas de mode d'adressage direct, tandis que le 68000 permet l'adressage direct court (16 bits) et long (32 bits).

4.3.4 Adressage indirect

La valeur de la donnée est stockée à une adresse m en MC. Cette adresse m est stockée dans un registre r ou à une adresse m'. C'est r ou m' qui est codé dans l'instruction (m' est appelé un **pointeur**). L'adressage indirect par registre est présent dans la totalité des UC, par contre, l'adressage indirect par mémoire est moins fréquent. Il peut cependant être simulé par un adressage direct dans un registre suivi d'un adressage indirect par registre.
Peu de machines disposent de mode d'adressage indirect à plusieurs **niveaux d'indirection** !

Avantage : taille quelconque de l'opérande
Inconvénient : accès mémoire supplémentaire(s), taille importante de l'instruction (pas par registre)

Exemples : Z80 indirection seulement par HL
ADD A, (HL) ; adressage indirect par registre
codage de l'instruction sur 8 bits (code op.) : A et HL sont désignés implicitement

LD (HL), <reg> ; adressage indirect par registre
code op. sur 5 bits et reg sur 3 bits; HL implicite

4.3.5 Adressage indexé (ou basé)

Il est parfois nécessaire d'accéder à des données situées à des adresses consécutives en MC. En adressage indexé, on charge un **registre d'index** avec l'adresse de début de cette zone de données, puis on spécifie dans l'instruction, le déplacement à réaliser à partir de cet index. L'adresse réelle de la donnée accédée est donc égale à l'adresse de l'index ajoutée au déplacement.
Certaines UC exécutent automatiquement l'incrémentation ou la décrémentation de leurs registres d'index ce qui permet, en bouclant, de réaliser des transferts ou d'autres opérations sur des zones (chaînes de caractères).

Avantage : taille importante de la zone (256 octets si déplacement sur 8 bits)
Inconvénient : taille importante de l'instruction (si codage du déplacement)

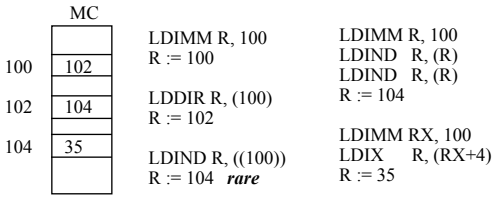
Exemple : Z80 indexation par IX et IY
LD (IX+<dépl>), <reg> ; adressage indexé par IX
codage de l'instruction : code op. sur 12 bits, IX sur 1 bit, reg sur 3 bits, dépl sur 8 bits.

Sur le 8086, MOVSB (MOVe String Byte) permet de transférer l'octet en (SI) vers (DI) puis d'incrémenter ou décrémenter SI et DI. Remarquons que l'indexation sans déplacement équivaut à l'indirection.

4.3.6 Remarques

Dans une instruction, lorsque deux opérandes sont utilisés, deux modes d'adressages interviennent. Souvent certains modes sont incompatibles avec d'autres pour des raisons de taille du code instruction ou de temps d'accès : par exemple en 8086, jamais deux adressages directs. Par contre, un opérande fait parfois appel à la conjonction de deux modes d'adressage : mode indexé et basé + déplacement du 8086 !

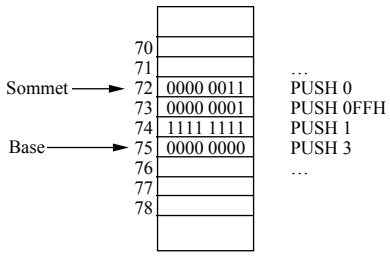
Schéma des différents modes d'adressage



Toute indirection à n niveaux peut être simulée dès lors qu'on possède une instruction à indirection simple.

4.3.7 Adressage par pile

La **pile d'exécution** de l'UC est constituée d'une zone de la MC dans laquelle sont transférés des mots selon une stratégie Dernier Entré Premier Sorti (*Last In First Out* ***LIFO***). Le premier élément entré dans la pile est placé à la **base** de la pile et le dernier élément entré se situe au **sommet** de la pile.



Les instructions d'entrée et de sortie de pile sont **PUSH** <n> et **POP** <reg>. Selon le type d'UC, la pile remonte vers les adresses faibles (voir schéma) ou bien descend vers les adresses fortes.

L'adresse du sommet de pile est toujours conservée dans un registre nommé **pointeur de pile** (*Stack Pointer SP*). Sur certaines machines, un registre général peut servir de SP. Certaines UC utilisent également un **pointeur de base de pile** (*Base Pointer BP*).

4.3.8 Utilisations de la pile

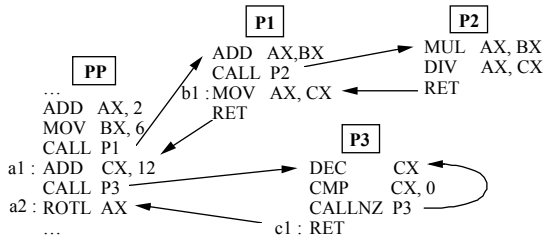
Avantages :
• structure de données LIFO et opérations de manipulation physiquement implantées : vitesse
• instructions courtes car opérandes implicites

Inconvénients :
• pas toujours dans une zone MC protégée
• sa cohérence nécessite égalité du nombre d'empilages et de dépilages (programmeur).
• capacité souvent limitée (par exemple 1 segment)

L'appel procédural

L'intérêt de l'utilisation de sous-programmes nommés **procédures** lors de l'écriture de gros programmes a été démontré : **concision**, **modularité**, **cohérence**, … Le programme principal (PP) fait donc appel (*CALL*) à une procédure P1 qui exécute sa séquence d'instructions puis rend la main, retourne (*RET*) à l'instruction du PP qui suit l'appel. Cette rupture de séquence avec retour doit également pouvoir être réalisée dans le code de la procédure appelée P1, soit récursivement, soit vers une autre procédure P2. Ces appels imbriqués, en nombre quelconque, rendent impossible l'utilisation d'une batterie de registres qui sauveraient les valeurs de retour du CO !

Exemple d'appels procéduraux imbriqués

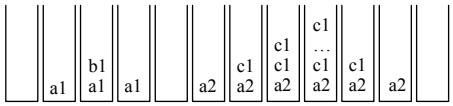


Les appels procéduraux imbriqués nécessitent l'utilisation de la pile de la manière suivante :

CALL <adrs> génère automatiquement (couche 1) :
1. PUSH CO ; le compteur ordinal pointe toujours sur l'instruction suivante
2. JMP <adrs> ; jump = goto

RET génère automatiquement :
1. POP CO ; branchement à l'adresse de retour

Exemple d'évolution de la pile



Attention aux appels récursifs mal programmés qui provoquent des débordements de pile (*Stack Overflow*) ! P3 : 2^n appels maximum (mots de n bits).

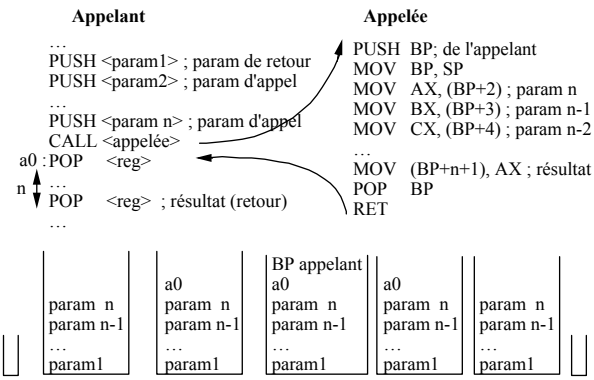
Paramètres et Variables locales

La plupart des langages de programmation évolués (Pascal, c, …) permettent le **passage de paramètres** et l'utilisation de **variables locales** aux procédures. Les paramètres sont passés soit **par valeur**, soit **par adresse**. Les variables locales sont créées à l'activation de la procédure et détruites lors de son retour (**durée**). D'autre part, leur **visibilité** est réduite aux instructions de la procédure.

Passage de paramètre

Dans les programmes écrits en langage machine, la gestion des paramètres d'appel et de retour est à la charge du programmeur (registres, MC, pile). Par contre, un **compilateur** doit fournir une gestion générique des paramètres quel que soit leur nombre et leur mode de passage. La plupart du temps, la pile est utilisée de la façon suivante :
Dans l'appelant, juste avant l'appel (CALL), le compilateur génère des instructions d'empilage (PUSHs) des paramètres d'appel et de retour. Juste après le CALL, il génère le même nombre de dépilage afin de nettoyer la pile. Dans le corps de la procédure appelée, la première opération consiste à affecter à un registre d'index ou de base (BP) la valeur du sommet de pile ± taille adresse de retour. Par la suite, les références aux paramètres sont effectuées via ce registre (BP±0..n) !

Exemple de passage de paramètres :



Variables locales

L'implémentation de l'espace dédié aux variables locales est réalisé dans la pile d'exécution. Les premières instructions machines correspondant à la compilation d'une procédure consistent toujours à positionner les variables locales dans la pile, juste au dessus de l'adresse de retour. Dans l'exemple précédent, il suffit de rajouter autant de PUSH, après le MOV BP,SP, que nécessaires. Ces variables locales seront ensuite accédées via des adressages (BP-i) générés par le compilateur. Ce type d'implémentation permet l'appel procédural ainsi que la récursivité.

Paramètres et variables locales
Finalement, chaque instance d'appel de procédure possède un espace d'adressage local composé :

- d'une part, des paramètres d'appel et de retour localisés dans la pile à (BP+2..n);
- d'autre part, des variables locales localisées dans la pile à (BP+1..m).

Exemple simple
Nous considérerons une procédure récursive simple n'utilisant que des paramètres et variables locales codés sur un mot machine. La fonction **mult** est une procédure à un paramètre de retour réalisant la multiplication de 2 entiers positifs par additions successives. Nous ne traiterons pas des problèmes de dépassement de capacité ni de l'optimisation de l'algorithme (y=0).

```
entierpositif mult(entierpositif x, entierpositif y)
entierpositif i ;      // inutile dans l'algo. !
si x=0 alors retourne 0
sinon début
        i=y
        retourne mult(x-1,i) + y
        fin
```

Etudions le code compilé de cette procédure et d'un appel initial avec des données en entrée : x=2, y=5.

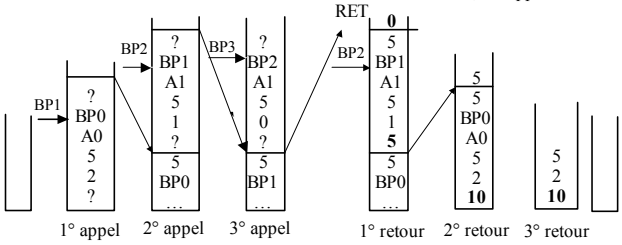
Image du code compilé et de la pile

Appelant

```
…
PUSH <multret> ; param de retour
PUSH  x ; param d'appel 2
PUSH  y ; param d'appel 5
CALL  mult
A0: POP    DX
    POP    DX
    POP    AX  ; résultat (retour)
    CALL  AFFICHEAX
    …
```

Mult

```
PUSH BP            ; de l'appelant
MOV  BP, SP
PUSH DX            ; i qcq en (BP-1)
CMP  (BP+3), 0     ; x=0 ?
JE   ZERO
MOV  (BP-1),(BP+2)   ; i:=y
MOV  AX, (BP+3)      ; x
DEC  AX              ; x-1
PUSH DX          ; retour qcq
PUSH AX          ; appel x-1
PUSH (BP-1)      ; appel i
CALL mult        ; récursif
A1: POP  DX    ; dépile
    POP  DX    ; dépile appel
    POP  AX    ; résultat (retour)
    ADD  AX, (BP+2)   ; résult + y
    MOV  (BP+4), AX   ; range résultat
    JMP  FIN
ZERO : MOV (BP+4),0 ; range rés.=0
FIN :   POP DX ; var locale i
    POP  BP          ; BP appelant
    RET
```



Cet exemple illustre bien le danger de croissance de la pile lors d'appels récursifs mal programmés ! Remarquons que la **dérécursivation** évidente de mult peut être réalisée par le programmeur mais parfois aussi par le compilateur.

Autres utilisations de la pile

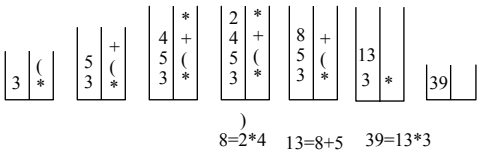
Cette structure de données LIFO est massivement employée dans le cadre d'algorithmes divers et variés.

Exemples

• dérécursivation automatique : dans l'exemple précédent, les CALL récursifs peuvent être évités en empilant successivement les valeurs des paramètres d'appels puis en réalisant des additions successives lors des dépilages.

• évaluation d'expressions arithmétiques infixées et parenthésées : 3*(5+4*2)

3*(5+4*2)



• parcours d'arbre (préfixe, infixe, postfixe)

• etc …

5. Les Systèmes de Gestion de Fichiers

5.1 Les Fichiers

5.1.1 Introduction

Définition conceptuelle :
Un fichier est une **collection** organisée d'informations de même nature regroupées en vue de leur conservation et de leur utilisation dans un Système d'Information.

Remarque :
inclut les SI non automatisés (agenda, catalogue de produits, répertoire téléphonique,…)

Définition logique :
C'est une **collection ordonnée** d'articles (enregistrement logique, item, "record"), chaque article étant composés de champs (attributs, rubriques, zones, "fields"). Chaque champ est défini par un nom unique et un domaine de valeurs.

Remarque :
Selon les SE, la longueur, le nombre, la structure des champs est fixe ou variable. Lorsque l'article est réduit à un octet, le fichier est qualifié de **non structuré**. Au niveau logique, plusieurs modèles de base de données ont été définis : modèle relationnel [Codd 76], réseau, hiérarchique.

Définition physique :
Un fichier est constitué d'un ensemble de **blocs** (enregistrement physique, granule, unité d'allocation, "block", "cluster") situés en **mémoire secondaire**. Les articles d'un même fichier peuvent être groupés sur un même bloc (Facteur de groupage ou de Blocage (FB) = nb d'articles/bloc) mais on peut aussi avoir la situation inverse : une taille d'article nécessitant plusieurs blocs. En aucun cas, un article de taille ≤ taille d'un bloc n'est partitionné sur plusieurs blocs → lecture 1 article = 1 E/S utile.

Remarque :
les blocs de MS sont alloués à un fichier selon différentes méthodes liées au type de support qu'il soit adressable (disques, …) ou séquentiel (bandes, cassettes, "streamers"). Ces méthodes d'**allocation** sont couplées à des méthodes de **chaînage** des différents blocs d'un même fichier et seront étudiées dans le chapitre SGF.

5.1.2 Opérations et modes d'accès

Un certain nombre d'opérations génériques doivent pouvoir être réalisées sur tout fichier :

- création	création et initialisation du **noeud descripteur** (i-node, File Control Block, Data Control Block) contenant taille, date modif., créateur, adrs bloc(s), …
- destruction	désallocation des blocs occupés et suppression du noeud descripteur
- ouverture	réservation de tampons d'E/S en MC pour le transfert des blocs
- fermeture	**recopie des tampons MC vers MS** (sauvegarde)
- lecture	consultation d'un article
- écriture	insertion ou suppression d'un article

La lecture et l'écriture constituent les **modes d'accès** et peuvent être combinés (mise à jour) lors de l'ouverture d'un fichier (existant). A la création, un fichier est toujours ouvert en écriture. Un SE multi-utilisateurs doit toujours vérifier les droits de l'utilisateur lors de l'ouverture d'un fichier.

5.1.3 Caractéristiques fonctionnelles

Volume : taille d'un fichier en octets ou multiples. Si articles de longueur fixe alors taille article*nb articles.

Taux de consultation/mise à jour pour un traitement donné : rapport entre le nombre d'articles intervenant dans le traitement et nb total d'articles.

Exemple : fichier "personnel", traitement "paye"
==> taux de consultation = 100%

Remarque : un taux de consultation important implique souvent un traitement par lot avec une méthode d'accès séquentielle.

Fréquence d'utilisation : nb de fois où le fichier est utilisé pendant une période donnée.

Taux d'accroissement : pourcentage d'articles ajoutés pendant une période donnée.

Taux de renouvellement/suppression : pourcentage d'articles nouveaux/supprimés pendant une période donnée. (TR=TS ==> volume stable)

Ces différentes caractéristiques ainsi que le type prépondérant d'utilisation (traitement par lot ou interactif) d'un fichier doivent permettre de décider de la structuration des articles, du type de support de stockage et des méthodes d'accès.

On classe souvent les fichiers en différentes catégories :

fichiers **permanents** : informations vitales de l'entreprise : Client, Stock, Fournisseurs...
durée de vie illimitée, Fréquence d'Utilisation élevée, màj périodique par mouvements ou interactive.

fichiers **historiques** : archives du SI : Tarifs, Prêts-Bibliothèque, journal des opérations…
pas de màj, taux d'accroissement élevé, taux de suppression nul.

fichiers **mouvements** : permettent la màj en batch des permanents afin d'éviter incohérences ponctuelles : Entrée/Sortie hebdomadaire stock, heures supplémentaires Janvier, …
durée de vie limité, fréquence d'utilisation très faible.

fichiers de **manoeuvre** : durée de vie très courte (exécution d'un programme) : spool, fichier intermédiaire.

5.1.4 Structure et Longueur des articles

Articles de longueur fixe et de structure unique
Dans la plupart des cas, chaque article d'un fichier contient le même nombre de champs chaque champ étant de taille fixe. Il existe un type unique d'articles.

Exemple : fichier STOCK

nom champ	*type*	*taille*	*exemple*
REF	N	4	0018
DESIGN	A	15	VIS 8*20
QTE	N	7	550 000
DATE	D	6	23/09/92

Articles de longueur fixe et de structure multiple
Les articles peuvent varier de structure parmi plusieurs sous-types d'articles (record variant de Pascal, union de C, C++). La longueur fixe des articles d'un tel fichier correspond à la longueur maximale des différents sous-types.

Exemple : fichier PERSONNE, articles de 42 octets

NOM(20), PRENOM(15), SITUATION(1),
  cas SITUATION=M alors DATE_MAR(6)
  cas SITUATION=D alors DATE_DIV(6)
  cas SITUATION=C alors rien

Remarque : les deux types d'articles précédents constituent la quasi-totalité des fichiers. La longueur fixe des articles permet de réaliser efficacement les calculs d'adresses de ceux-ci.

Articles de longueur var. et de structure multiple
L'intérêt principal de ce type d'articles consiste dans le gain de place (suppression des blancs) constitué par le compactage des fichiers concernés. L'inconvénient majeur réside dans la difficulté à calculer l'adresse d'un article. Ce type d'articles est particulièrement utilisé à des fins d'archivage sur support séquentiel et pour la téléinformatique.

La séparation des articles et des champs est souvent effectuée par insertion de préfixes indiquant la longueur de l'article ou du champ.

Exemple : fichier PERSONNE

TailleArt(1), NOM(1+20), PRENOM(1+15), SIT(1),
  cas SIT=M alors DATE_MAR(6)
  cas SIT=D alors DATE_DIV(6)
  cas SIT=C alors rien

18,5,UHAND,4,PAUL,M,23/08/91,
12,5,PETIT,4,JEAN,C,
19,6,HOCHON,4,PAUL,D,10/10/89,
…

5.1.5 Méthodes d'accès

Accès Séquentiel
Les articles sont totalement et strictement ordonnés. L'accès (lecture/écriture) à un article ne peut être réalisé qu'après l'accès à l'article précédent. Un **pointeur** d'article courant permet de repérer la position dans le fichier à un instant donné. L'**ouverture** en lecture positionne le pointeur sur le **1er article** puis les lectures successives font progresser le pointeur jusqu'à la position End Of File (**EOF**). Selon les cas, l'ouverture en écriture positionne le pointeur en début de fichier (rewrite) ou en fin de fichier (append). Il existe une opération spécifique de **remise à zéro** du pointeur (retour en début de fichier, rembobinage, "reset", "rewind").

Historiquement lié au support **bande** magnétique et cartes perforées, cette méthode d'accès est la plus simple et de surcroît est **universelle**. Elle reste très utilisée notamment pour les fichiers non structurés (textes, exécutables,…) ou les fichiers historiques. Pour les fichiers permanents, si le taux de consultation ou de màj est important, la solution séquentielle doit être envisagée. Cependant, l'accès séquentiel est impensable pour un bon nombre d'op. interactives (réservations, op. bancaires, …)

Accès direct
(ou sélectif, aléatoire, "random")

Ce type d'accès nécessite un support **adressable**! Un ou plusieurs champs des articles servent **d'expression d'accès (clé)** pour "**identifier**" et accéder à un ou plusieurs articles. On peut directement lire ou écrire l'article grâce à une opération du type suivant :

lire/écrire(fichier, valclé, adrsMCtransfert)

On peut également vouloir accéder aux articles par l'intermédiaire de plusieurs clés :

Exemple : fichier Personne; clé1 = N°SS; clé2 = Nom

Par la suite, nous traiterons de la manière de réaliser l'accès direct sur un fichier à **clé unique**. Selon les cas, la transposition aux clés multiples est plus ou moins simple !

Adressage direct

Ce type d'adressage est un cas d'école puisqu'il associe à chaque **valeur de clé** l'**adresse physique** du bloc contenant l'article. Il y a ainsi identité des valeurs de clé d'article et des adresses physiques de bloc. Un inconvénient évident est que le domaine des valeurs de clé doit correspondre exactement aux domaine des adresses physiques. De plus, l'espace adressable est très fortement **sous-occupé** (FB=1) et ne peut être partagé par plusieurs fichiers ayant des valeurs de clés conflictuelles ! Par contre, le temps d'accès à un article est **minimal**. La clé doit être identifiante !

Adressage relatif

La clé est un nombre entier correspondant au **n**uméro **l**ogique (0..n-1) d'article dans le fichier. On obtient aisément l'adresse physique (bloc ; dépl.) de celui-ci :
taille fixe : AdPhy:=(Orga(nl div FactBloq); taille*(nl mod FactBloq))
taille var. : il existe une table de correspondance [nl --> AdPhy]

Si la numérotation logique n'est plus continue (suppressions), de l'espace inutile continue à être occupé par le fichier ! En effet, la compression après chaque suppression serait trop coûteuse. L'insertion suivante sera donc réalisée dans un trou. L'ordre des nl ne correspond donc pas forcément à l'ordre d'insertion !

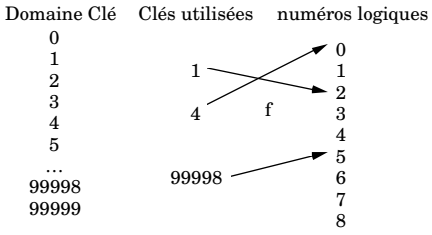
Adressage dispersé, calculé ("hash-coding")

L'adressage dispersé consiste à calculer un **n**uméro **l**ogique d'article à partir d'une valeur de clé et d'une fonction de hachage : nl:=f(c).

f permet de réduire le domaine des nl par rapport au domaine des clés afin de donner un volume de fichier supérieur au volume utile mais inférieur à card(domaineClé)*tailleArt.

Avantages :
calcul en MC ==> accès très rapide
clés quelconques : non identifiantes ==> collisions

exemple :



Dans cet exemple, un domaine de 10^6 clés potentielles est réduit à un espace logique de 9 numéros logiques permettant de stocker les 3 articles réels.

hachage parfait
Une fonction de dispersion idéale est celle qui réalise une **bijection** de l'ensemble des clés existantes vers l'ensemble des numéros logiques. Il n'y a ainsi aucun espace perdu. La recherche d'une fonction idéale est calculable sur un ensemble statique de clés identifiantes mais impossible sur un ensemble dynamique

Collisions ou conflits
Il y a collision lorsque la fonction de hachage associe un numéro logique déjà utilisé à un nouvel article. Il faut alors traiter la collision pour insérer le nouvel article dans le fichier :
- soit dans une zone de collision générale accédée soit séquentiellement, soit par une autre fonction f2
- soit dans une zone de collision spécifique à chaque numéro logique.

La détermination d'une bonne fonction de hachage donnant un **faible taux de collision** est un gage de rapidité d'accès. **Exemple** : somme des entiers (16 bits) composant la clef modulo taille fichier. Avec une clé non identif., le taux de collision augmente.

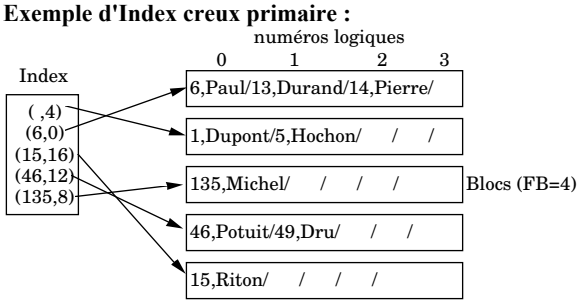
Lorsque le volume utile du fichier croît, il est nécessaire de réorganiser celui-ci, par exemple, en allouant un espace logique double du précédent, en modifiant f et en réorganisant les articles existants.

inconvénients :
collisions coûteuses
pas d'accès séquentiel (accès calculés répétés)
taux d'occupation mémoire <= 1
taille du fichier connue à priori
réorganisations coûteuses

Adressage indexé

Un **index** est une table de couples (valeur clé, nl) **triée** sur les valeurs de la clé. L'index est **dense** si toutes les clés du fichier y sont recensées. Sinon l'index est dit **creux** et une clé c présente dans le fichier et absente de l'index est dite couverte par la clé tout juste inférieure présente dans l'index. Un index creux implique une **clé primaire**, c'est à dire que le fichier soit **trié** sur cette clé. D'autres index secondaires doivent alors être dense !

Exemple d'Index creux primaire :

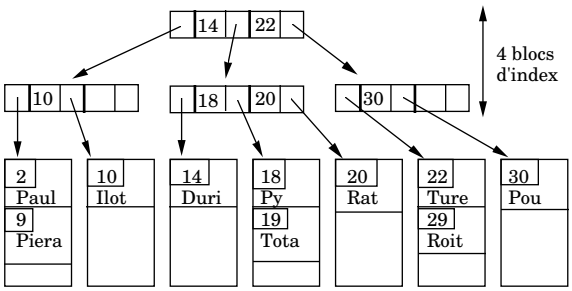


La recherche d'une clé d'un article est réalisée par dichotomie sur le fichier d'index. Lorsque le fichier atteint un volume important, son fichier d'index ne tient plus sur un seul bloc et on est alors forcé de parcourir séquentiellement les blocs d'index.

Aussi, on préfère une organisation arborescente (**b-arbre**) de l'index dans laquelle existe une hiérarchie de sous-tables d'index creux permettant une bonne rapidité d'accès. En fait, on indexe chaque bloc d'index !

Contraintes sur un b-arbre d'ordre p (#ptrs/bloc index)
- tout chemin (racine --> feuille) est de longueur identique = hauteur (exemple h=2)
- chaque bloc d'index est toujours au moins à moitié plein (sauf la racine) : chaque bloc d'index contient k clés avec EntInf(p/2)≤k≤p-1 (ex. p=3).

Exemple d'index par b-arbre :



1 bloc de fichier = 1 à 2 articles

Afin de maintenir les contraintes lors des suppressions ou des insertions d'articles, on est parfois obligé de **réorganiser** l'arborescence !
Moins rapide que le hachage pour l'accès direct, l'adressage **indexé** permet cependant de traiter également l'**accès séquentiel** à moindre coût (index creux ou b-arbre).

Remarques

- Il existe toujours un compromis (place utilisée/temps d'accès).
- Les insertions et suppressions d'articles provoquent des trous dans les blocs de données. L'index b-arbre gère ceux-ci mais les index linéaires (creux ou denses) ne le font pas… (voir SGF)
- Pratiquement, les b-arbres et les index creux (ISAM) prédominent dans les SGBD (vitesse + accès séquentiel).
- index multiples : on peut ajouter des index secondaires denses peu efficaces pour l'accès séquentiel.

5.2 Système de Gestion des Fichiers

5.2.1 Organisation arborecente

La quasi-totalité des SGF sont organisés en arborescence de **répertoires** (catalogue, "directory") contenant des sous-répertoires et fichiers. Suivant le cas, les répertoires sont considérés comme des fichiers (unix) ou bien demeurent dans des zones de MS spécifiques (MS-DOS zone DIR). Le répertoire permet d'accéder au **noeud descripteur** d'un fichier qu'il contient. Ce noeud permettra de connaître la localisation et l'ordonnancement des blocs contenant les données de ce fichier. Par la suite, on supposera des répertoires de même nature que les fichiers.

5.2.2 Allocation de mémoire secondaire

5.2.2.1 Support séquentiel (bande)

- pistes longitudinales (1 car/ 8|16 pistes //)
- densité d'enregistrement en bpi (1600..6000)
- blocs séparés par des gap inter-bloc (2 cm)
- fichiers séparés par des gaps inter-fichier
- entêtes de blocs et de fichiers permettant le positionnement.
- on ne modifie jamais directement une bande.
- les maj sont effectuées lors de la recopie sur une autre bande.
- faible coût, accès lent ==> archives, copies, sauvegardes

5.2.2.2 Support adressable (disque)

Espace adressable numéroté (0..b-1) de b blocs. L'accès à deux **blocs consécutifs** est peu coûteux (translation et rotation minimale de la tête de lecture/écriture). Remarquons que la numérotation logique des blocs ne correspond pas à la topologie (facteur d'entrelacement).

2 problèmes :

- disponibilité ou occupation d'un bloc
- localisation et ordonnancement des blocs d'un fichier

Gestion des blocs disponibles

Une liste des blocs libres doit être maintenue afin de gérer les créations et destructions de fichiers.

implémentations :

• tableau de bits : 0100011111101…110 (mot de b bits)
demande bloc : chercher le 1° 0 dans la liste puis 0 -> 1
rejet bloc : 1 -> 0

• considérer cette liste des blocs libres comme un fichier, donc utiliser une des méthodes d'allocation suivantes.

Allocation contigüe

Ex : fic1 (b0..b3); fic2(b9..b12); …; fick(b5..b6)

• temps d'accès séquentiel et direct optimal
• noeud descripteur contenant blocDebut, nbBlocs
• peu utilisé car les fichiers ont des tailles dynamiques ==> surévaluation a priori des volumes fichiers, réorganisations fréquentes et coûteuses lors des augmentations de volume.
• utilisation pour les tables systèmes de taille fixe :
MS-DOS : boot, FAT1, FAT2, DIR
Unix : boot, SF type, i-node table

Problème de l'allocation dynamique (=new, malloc)
lors d'une demande de n blocs libres, le système doit chercher un **trou** (suite de blocs libres) suffisament grand parmi les trous de l'espace disque.

Stratégies

• Premier trouvé ("**First-Fit**") : on parcours la liste des trous jusqu'à en avoir trouvé un assez grand.
• Plus Petit ("**Best-Fit**") : on cherche dans **toute** la liste la borne supérieure des trous => parcours intégral ou liste des trous **triée** et recherche dicho.
• Plus Grand ("Worst-Fit") : inverse de Best-Fit.

Des simulations ont prouvées la meilleure performance de BF et FF en temps et en espace utilisé. Lorsque l'espace libre est suffisant mais qu'il n'existe pas de trou assez grand, cette **fragmentation externe** implique un compactage des fichiers.

Allocation chaînée

Les blocs de données d'un fichier contiennent un pointeur sur le bloc suivant. Le noeud descripteur contient, lui, la **tête** de liste. Lors de la création, la tête est mise à **nil**, puis, au fur et à mesure des demandes, une allocation contigüe (optimale en temps d'accès) est tentée. Si celle-ci ne peut avoir lieu, on choisit le(s) premier bloc disponible.

Avantage
• plus de fragmentation externe donc seule limitation = taille espace disque libre.

Inconvénients
• Accès séquentiel seulement !

• **fragmentation interne** des fichiers nécessitant de nombreux mouvements de tête. Des utilitaires de compactage permettent la réorganisation périodique des fichiers en allocation contigüe.

• fiabilité : si un pointeur est détruit logiciellement ou matériellement (bloc illisible) on perd tout le reste du fichier.

• encombrement disque dû aux pointeurs si blocs de petite taille

Allocation chaînée déportée

On déporte le chaînage dans une table système (FAT pour MS-DOS) dont chaque entrée pointe sur le bloc de données suivant du fichier. La tête de liste est positionnée dans le noeud descripteur. Cette table de chaînage est chargée en MC (volume courant) afin de permettre des accès directs rapides.

Avantage
• Accès direct et séquentiel peu coûteux

Inconvénients
• fragmentation interne (compactages périodiques)
• accès direct après parcours de la liste en MC

Allocation indirecte (indexée)

Un **bloc d'index** contient la liste des pointeurs sur les blocs de données du fichier. Ce bloc d'index est lui-même pointé par un champ du noeud descripteur de fichier. A la création le bloc d'index est initialisé à nil, nil, …, nil et lors des écritures successives, les premiers pointeurs sont affectés des adresses des blocs alloués par le gestionnaire de mémoire disponible. Ici encore la contiguïté est tentée ! A l'ouverture, le bloc index est chargé en MC afin de permettre les accès.

Avantage
• accès direct et séquentiel rapide

Inconvénients
• fragmentation interne => compactages périodiques
• taille gaspillée dans les blocs d'index >> taille des pointeurs en alloc chaînée.
• petits fichiers : pour 2 ou 3 adresses de blocs, on monopolise un bloc index !
• très grands fichiers : un bloc index étant insuffisant, plusieurs blocs index peuvent être chaînés et un mécanisme de multiple indirection peut être utilisé.

Allocation directe

Le **noeud descripteur** contient tous les pointeurs sur les blocs de données. De taille fixe et petite, cette structure de données ne pourra être utilisée que pour des fichiers de faible volume.

Avantage
• accès direct et séquentiel très rapide
• peut être mixtée avec indirection ou chaînage

Inconvénients
• taille limitée des fichiers
• fragmentation interne

5.3 Exemples

5.3.1 Le SGF MS-DOS

Nous décrivons ci-après l'**organisation physique** d'un volume MS-DOS sous la forme d'une disquette 5,25 pouces. Les principes d'allocation de MS-DOS restent les-mêmes quel que soit le support adressable, seuls le nombre et la taille des champs varient.

Une disquette, ou disque souple ou "floppy disk", est constituée d'un support plastique mince de forme discale d'un rayon de 5,25 pouces (1 pouce = 2,54 cm) sur lequel a été déposé un substrat magnétique. Dans le cas de disquettes double face double densité, cette surface homogène de particules magnétisables est structurée en 2 faces de **40 pistes** possédant chacune **9 secteurs**. Les pistes concentriques sont numérotées de 0 à 39 depuis l'extérieur vers l'intérieur. Chaque secteur contient **512** octets utiles. La capacité d'une disquette est donc de 2 * 40 * 9 * 512 = 360 Koctets.

L'unité d'allocation ("**cluster**"), ou bloc, est la plus petite partie d'espace mémoire allouable sur une disquette. Pour une disquette 5,25 pouces à 360 Ko, un **bloc** est constitué de **deux secteurs consécutifs** et a donc une capacité d'**1 Ko**. La numérotation des secteurs est effectuée de la façon suivante :

Face 0 Piste 0 Secteurs 0 à 8
Face 1 Piste 0 Secteurs 9 à 17
Face 0 Piste 1 Secteurs 18 à 26
Face 1 Piste 1 Secteurs 27 à 35 etc…

Fichier et catalogue

Un **fichier** MS-DOS est une **suite d'octets** désigné par un identifiant composé d'un **nom de 8 caractères** et d'une **extension de 3 caractères**. Par exemple, COMMAND.COM, PRG1.PAS, SALAIRES.DBF, TEXTE1.DOC…

Un **catalogue** ("directory") MS-DOS est un type de fichier particulier permettant de regrouper différents fichiers de données et/ou d'autres catalogues dans une même entité. L'architecture du Système de Fichiers est donc une arborescence dont toutes les feuilles sont des fichiers et tous les nœuds intermédiaires des catalogues. Le catalogue **racine** est désigné par un "backslash" \ et est stocké en début de disquette.

Allocation

Les fichiers sont fragmentés en **allocation chaînée déportée** et MS-DOS conserve dans une table l'adresse des différents fragments. Cette table s'appelle Table d'Allocation des Fichiers ("File Allocation Table") et sera désignée par la suite par **FAT**. **Deux copies** de la FAT sont stockées sur la disquette (secteurs 1,2 et 3,4) afin de garantir la sécurité des données en cas de destruction accidentelle d'une des deux copies. Le **catalogue général (racine)** de la disquette est lui situé sur les secteurs 5 à 11. Le secteur 0, quant à lui, contient le programme d'amorçage sur les disquettes systèmes ("**boot-strap**"). Enfin les secteurs 12 à 719 contiennent les données des différents fichiers.

catalogue racine

C'est une table dont les entrées ("**File Control Block**") ont une longueur de 32 octets qui décrivent les fichiers et les sous-catalogues. Une entrée de répertoire peut être schématisée de la façon suivante :

FCB

<table>
<tr><td>0..7</td><td colspan="4">nom du fichier sur 8 octets</td></tr>
<tr><td>8..15</td><td colspan="2">extension sur 3 octets</td><td>attribut</td><td>réservé par MS-DOS</td></tr>
<tr><td>16..23</td><td colspan="3">réservé par MS-DOS</td><td>heure modificat</td></tr>
<tr><td>24..31</td><td>date modificat.</td><td>1° entrée FAT</td><td colspan="2">Taille fichier (LSB, MSB)</td></tr>
</table>

L'octet d'**attribut** permet de spécifier certaines **protections** : fichier caché, lecture seulement, archive, système, normal ou encore de préciser que le fichier est un catalogue. L'heure et la date de dernière modification permettent de retrouver la dernière version d'un fichier de travail. La **taille** du fichier est codé sur **32 bits** ce qui permet une taille maximale de **4 Giga-octets** ! Ce qui est bien entendu **impossible** sur une disquette de 360 Ko. Enfin, la 1ère entrée dans la FAT permet d'indiquer le **premier maillon du chaînage** dans la FAT qui pointera lui-même sur le second qui pointera sur le troisième etc…

la FAT

C'est une table d'entrées d'une longueur de **12 bits** (1,5 octets) qui spécifie le chaînage permettant de reconstituer un fichier fragmenté sur plusieurs blocs. Les deux premières entrées (0 et 1) de cette table sont réservées par le système pour préciser le **type de la disquette** sur laquelle elle se trouve (simple/double face, simple/double densité). L'entrée numéro 2 correspond au premier **bloc** de la disquette **non réservé** au système, c'est-à-dire le bloc n° 6 (secteurs 12 et 13). Ainsi le fichier dont le "1° entrée FAT" de son entrée de répertoire est 2 verra son premier bloc de données situé sur les secteurs 12 et 13. Dans cette première entrée FAT on trouve la valeur de l'entrée suivante, par exemple 005, qui correspond au bloc de données suivant.

exemple de FAT

<table>
<tr><td>type disquette</td><td>005</td><td>000</td><td></td><td>i</td><td></td><td>…</td><td>FFF</td><td>…</td></tr>
<tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>…</td><td>i</td></tr>
</table>

1° bloc du fichier = bloc n° 6 — 2° bloc du fichier = bloc n° 9 — indique la fin du chaînage

Dans cet exemple, le fichier est contenu sur trois blocs (6, 9 et i+4). La **fin du chaînage** est indiqué par la valeur 0FFFH (nil) tandis qu'une valeur 000 indique un bloc libre. La **gestion des blocs libres** (table de "12 bits") est en effet couplé au mécanisme de FAT.

Deux remarques

- Les entrées du répertoire racine ne sont **pas compactées** et la valeur 0E5H située sur le **premier octet** d'un FCB signifie que cette entrée est libre. Soit cette entrée n'a jamais été utilisée pour décrire un fichier ou un catalogue, soit ce fichier ou ce catalogue a été détruit (**delete** ou rmdir) et le système a simplement **surchargé** le premier octet du nom du fichier par un code 0E5H. Par conséquent, la recherche d'un nom dans le catalogue général est effectuée séquentiellement sur toutes les entrées. On aurait pu tout aussi bien choisir de compacter les entrées du catalogue à chaque destruction de fichier, ce qui aurait diminué le temps de recherche d'un fichier n'existant pas ! Mais cela aurait interdit les utilitaires de récupération de fichiers détruits (undelete).
- La seconde remarque concerne la **fragmentation** des fichiers sur la disquette. Au bout d'un certain nombre de créations et de destructions de fichiers et de répertoires, les différents blocs supportant les données d'un fichier se trouvent être disséminés sur la disquette. Or le transfert en mémoire centrale de tous les blocs de ce fichier va nécessiter un grand nombre de mouvements de translations du bras de lecture. C'est pourquoi il existe des utilitaires de **compactage** des blocs des fichiers d'une disquette permettant de minimiser les temps d'accès à un fichier.

La VFAT 32

Nouveau standard de FAT permettant de gérer des disques durs de grande capacité (>2Go) :

- Entrées de FAT sur 12, 16 ou 32 bits permettant des petits clusters ;
- FCB sur 32, 37 ou 44 octets (FCB étendu).

Exemple : disquette 1.4 Mo

Volume :

- 80 pistes/face : 160 pistes
- 18 secteurs/piste : 2880 secteurs
- 512 octets/secteurs : 1440 Ko soit 1.40 Mo
- 1 secteur/cluster

FAT

- 1 FAT : 9 secteurs * 2 copies ;
- Entrée de FAT : 12 bits ;
- 3072 entrées (2880 clusters)

Répertoire racine :

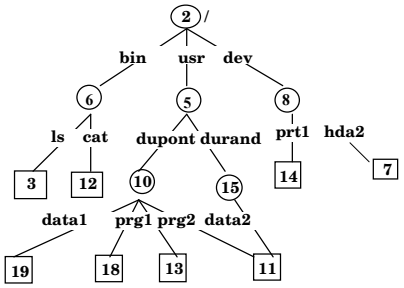
- Répertoire racine sur 14 secteurs
- FCB sur 32 octets
- soit 224 entrées

Taux de transfert : 500 Kbits/s

Les noms longs de fichiers de WinX sont stockés dans le FCB suivant immédiat codé en Unicode.

5.3.2 Le SGF d'Unix

Exemple d'une arborescence de fichiers Unix



• catalogues : bin, usr, dev, dupont, durant;
• fichiers ordinaires : ls, cat , data1, prg1, prg2, data2;
• fichiers périphériques : sous le catalogue dev ("device") : prt1, dsk2.

Caractéristiques

• Entrées/Sorties généralisées ou transparentes
• désignation : chemin d'accès absolu /…, ou relatif
• fichiers **non structurés** = suite d'octets numérotés logiquement de 0 à n-1 (n = longueur du fichier) : structuration à la charge des programmeurs
• accès **direct** à une suite d'octet à partir d'une position i dans le fichier ($0 \leq i \leq n-1$) **et/ou** accès **séquentiel**
• catalogues Unix = liste (nom de fichier, # i-node)
• identification = # i-node
• désignations multiples : compteurs de liens ou de références
• système de protection : **rwx rwx rwx** propriétaire groupe autres

5.4 Programmation des fichiers Unix en C/C++

5.4.1 Généralités

• différence entre **appel système** (manuel section 2) et **fonction de bibliothèque** ("library") (section 3).
- appel système : primitive du noyau interrompant le pus (passe en mode noyau), pas d'édition de liens, vu comme une fon c externe retournant un résultat==-1 si erreur : lire alors la variable 'extern int errno' pour connaitre le type d'erreur (1 : permission, 2 : fichier non existant, …)
- fon bibli : édition de liens avec librairies (E/S formatées, maths, etc …).

• passage des paramètres de la ligne de cde :
main(int nbpar, char *argv[], char *env[]) {…}
nbpar : nombre de paramètres ($#argv)
argv : tableau de chaines == liste des paramètres (0 .. nbpar-1)
env : tableau de chaines == liste des var d'env. (nomvar=valeur)

• sortie de pus : void exit(int status);

• lancement d'une cde externe : void system(char *commande);

exemple :

```
/* essai source c */
#include <stdio.h>
main(int n, char*argv[], char*env[]) {
   printf("essai affichage de .cshrc\n");
   system("cat .cshrc");
   exit(9);
}
toto> cc essai.c; a.out
```


5.4.2 Appels systèmes E/S et Fichiers

- appels systèmes : on travaille directement sur les tampons systèmes des fichiers ;
- fon bibli : on travaille sur une image du fichier en zone utilisateur et cette image est transférée de temps à autre en zone système (E/S tamponnées) : nb d'it mini mais pbs de concurrence !

2 types : - manip. des liens de répertoires
- manip. fichier et i-node par descripteur

descripteur : entier positif désignant le fichier ou le périph : 0,1,2 réservés pour Entrée, Sortie et Err standard

5.4.2.1 Création

int **creat**(char *path, int droits); A EVITER
ou int **open**(char *path, int flags, int droits)
avec flags==O_WRONLY |O_CREAT|O_TRUNC
fournit un descripteur, à partir d'un chemin d'accès et des droits d'accès qui seront masqués par umask.
exemple : fic=creat("/home/lic/toto/ficessai",0666)
ouvre un fichier vide en écriture, pointeur à 0. Attention, si le fichier existait déjà, il conserve ses anciens droits d'accès !!! Il est donc utile de faire précéder la création par : **chmod**(path,droits).

5.4.2.2 Ouverture

int **open**(char *path, int flags[, int mode])
avec flags : O_RDONLY (0) xor O_WRONLY (1) xor O_RDWR (2) , O_APPEND, O_TRUNC…
ouvre un fichier selon un mode (R|W|RW) et positionne le pointeur en début (fin si O_APPEND) de fichier; retourne un descripteur

5.4.2.3 Lecture

int **read**(int desc, char *buf, int nboctets)
retourne le nb d'octets (≤ nboctets) effectivement transférés dans buf et avance le pointeur; 0 si EOF, -1 si problème (desc inexistant…).

5.4.2.4 Ecriture

int **write**(int desc, char *buf, int nboctets)
retourne le nb de car écrits dans le fichier et avance le pointeur, -1 si pb. Si O_APPEND positionné, toute écriture est précédée de l'avancée du pointeur en EOF sinon écriture (surcharge) à partir de la position courante et avance le pointeur.

5.4.2.5 Fermeture

int close(int desc)
le nb de fichiers ouverts d'un processus étant limité (16), il peut être utile de fermer ses fichiers même si le exit les ferme tous automatiquement.

5.4.2.6 Déplacement de pointeur (Accès Direct)

long **lseek**(int desc, long depl, int flags)
déplacement absolu ou relatif du pointeur en fonction de la valeur de flags : SEEK_SET ou 0 : début ; SEEK_CUR ou 1 : courant ; SEEK_END ou 2 : fin. Retourne la position relative du pointeur par rapport au début de fichier (0..n-1) ou -1 si erreur.

5.4.2.7 Redirection d'E/S par duplication de desc.

int **dup**(int desc)
retourne un nouveau descripteur de **valeur minimale** désignant le même fichier ou E/S que desc.
ex : … d=creat("ficredir", 0666); close(1); dup(d); …
redirige la sortie standard vers ficredir

5.4.2.8 Autres appels systèmes (voir man)

access : teste les droits d'accès link, unlink : multi-références
stat : retourne le contenu du i-noeud d'un fichier (lstat, fstat)
chdir : change répertoire courant
chown, chmod : change propriétaire, droits d'accès
mkdir, mkfifo : crèe un répertoire, un tube …

5.4.3 Bibliothèque d'E/S standard du C

Le type FILE permettant de manipuler les fichiers par des FILE * souvent nommés flots « stream » :
- 3 macros : stdin, stdout, stderr pour descripteurs 0, 1, 2
- constante EOF (-1) == caractère lu lorsque fin de fichier

FILE ***fopen**(char *path, char *mode)
path:nom du fichier; mode:{"r","w","a"append,"r+"rw,"w+"raz et rw}

int **fclose**(FILE* f) vidage du tampon sur le fichier.

int [f]**getc**(FILE *g); int [f]**putc**(char c,FILE *g); **getchar**(); **putchar**(c)
lit/écrit un caractère du fichier g ; sans f : stdin, stdout

int **fgets**(char *ch, int n, FILE *f) : lit une chaîne ; **gets** pour stdin
min(nb car jusqu'à '\n', n-1) char +'\0' sont copiés dans ch
Attention, gets remplace le'\n' par '\0' mais pas fgets !

int **fputs**(char *ch, FILE *f) : écrit une chaîne ; **puts** pour stdout
copie ch dans f, sauf le '\0' final

int **fseek**(FILE *f, long depl, int base) : déplace le pointeur de fichier ; base==0 (1,2) : déplacement % au début (courant, fin) de f

int **feof**(FILE* f) : retourne vrai (!=0) si le pointeur est en EOF

int **fflush**(FILE *f) ; vidage du tampon en écriture vers le fichier

int **fpurge**(FILE *f) ; RAZ des tampons en écriture et en lecture sans utilisation du contenu !

5.4.3.1 E/S Formattées

int **fprintf**(FILE *f, format, liste_valeurs) ;
printf pour stdout : écriture formatée : %s chaîne, %c car, %d entier décimal, %x hexa
%10d pour convertir un entier en chaîne de 10 car

int **fscanf**(FILE *f, format, liste_pointeurs) ;
scanf pour stdin : lecture selon un certain format dans un fichier

5.4.3.2 Formatages sur chaines de car

int **sprintf**(char *ch, format, liste_valeurs)
écriture formatée dans une chaîne.

int **sscanf**(char *ch, format, liste_pointeurs)
lecture selon un certain format dans une chaîne

5.4.3.3 Manipulation de chaînes

char ***strcat**(char *s1, *s2); char ***strncat**(char *s1, *s2, int n)
concaténation (bornée par n)

strcmp(unsigned char *s1,*s2); **strncmp**(unsigned char *s1, *s2, int n) : comparaison (bornée par n)

char ***strcpy**(char *s1, *s2); char ***strncpy**(s1, s2, n)
copie (bornée par n)

int **strlen**(char *s) : longueur de s

char ***strchr**(char *s, char c) : recherche du 1er c dans s

char ***strpbrk**(char *s1, char *s2) :
recherche d'un char de s2 dans s1

5.4.4 Bibliothèque d'E/S standard du C++

Manipulateur : objet fonction redéfinissant le format des flots

format numérique :
```
cin>>hex>>i ;   // lecture de i en hexa : 0xA2D
cout<<oct<<i ;   // écriture en octal de i
```

cadrage et largeur du champs de sortie :
```
cout<<left<<setw(5)<<setfill('#')<<123 ;     // 123##
cout<<right<<setw(6)<<setfill('0')<<123 ;  // 000123
```

précision et arrondi :
```
cout<<setprecision(2)<<fixed<<2.0/3 ;
      // 0.67 (fixed|scientific (10e2))
```

divers :
```
cout<<endl  // fin de ligne '\n'
cout<<ends  // fin de chaine C '\0'
cout<<flush  // vidage du tampon sur le flot
...
```

Méthodes :
```
bool eof() ;
istream& read(char* s,int n) ; int gcount() ;
istream& get(char& c) ;
istream& get(char*s,int max,char delim='\n') ;
ostream& write(char* s,int n) ;
ostream& seekp(long depl,seekdir org) ;  // beg, cur, end
...
```

6. Les processus Unix

6.1 Généralités

Processus : suite temporelle d'exécutions d'instructions d'un programme par un processeur.
(programme : données + suite d'instructions)

La gestion des processus (pus) étant très dépendante du SE étudié, nous nous bornerons à définir quelques notions générales avant d'aborder plus particulièrement les pus Unix.

nom d'un pus : où numéro d'identification du pus qui permet sa manipulation par le système et par l'utilisateur l'ayant créé.

ressources : emplacements de mémoire centrale, périodes d'utilisation de l'UC, périphériques … nécessaires à un processus pour son évolution.

état : un processus disposant de toutes les ressources (UC, MC, …) nécessaires à l'exécution de sa prochaine instruction est dans l'état actif. Dans tous les autres cas, il est bloqué sur la ou les ressources manquantes.

Remarque : l'observation des ressources courantes d'un processus actif ne peut être fait qu'entre deux instructions (atomicité).

ressource **locale** à un pus i : si elle ne peut être utilisée que par le pus i. ex : variables du prog.

ressource **commune** : si elle n'est locale à aucun pus. ex : tube

partageabilité : une ressource commune est **critique** (resp. partageable à n=2 points d'accès) si sur un point observable elle ne peut être détenue que par un (resp. n) pus au plus. ex : l'UC est critique; une zone mémoire tube est partageable à 2 points d'accès.

Plusieurs pus sont dits en **exclusion mutuelle** sur r lorsque ils utilisent cette ressource critique r.

mode : niveau de pouvoir (droits) dans lequel s'exécute le pus lui permettant ou non d'accéder à certaines ressources et/ou d'exécuter certaines instructions privilégiées de l'UC. ex : dans de nombreux systèmes deux modes (seulement) existent : mode maître (ou système ou **noyau**)/mode esclave (ou **utilisateur**). Le mode d'un pus peut être statiquement donné à la création ou évoluer dynamiquement (Unix).

durée de vie : un pus naît, après chargement en MC, lors du lancement du programme par le SE et meurt à la fin de l'exécution de ce programme lors du retour au SE.

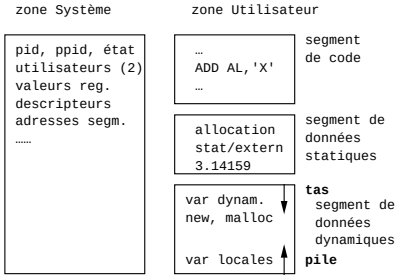
Les mécanismes de **synchronisation** permettent notamment l'activation d'un pus bloqué sur une ressource ou au contraire le blocage d'un pus actif. ex : l'accès de 2 pus à une ressource critique nécessite leur synchronisation afin qu'un seul d'entre eux n'obtienne la ressource.

Les mécanismes de **communication** permettent à plusieurs pus de se transmettre des données. Afin d'éviter la communication par fichier (E/S lentes) on utilise des structures de données en mémoire centrale : variables partagées, tubes, signaux, files de messages, sockets,…

Le **recouvrement** (overlay) est une technique qui permet de remplacer une partie de la mémoire centrale par une autre. ex : un programme très long peut être décomposé en parties se recouvrant pour diminuer l'espace utilisé (quasiment plus utilisé car grande mémoires et pagination).

6.2 Description des processus Unix

image mémoire d'un pus : ensemble des zones mémoires utilisées par un pus :



Un processus Unix réalise ses instructions normales en **mode utilisateur** puis commute en **mode système** lors d'un appel au noyau, d'une interruption, ou d'un déroutement.

La commutation de pus est toujours effectuée en mode système par le pus "partant" (pas de préemption).

6.3 Programmation en C/C++ des processus Unix

6.3.1 Identifications liées à un pus

appels systèmes et fonctions de bibli.

int getpid() **int getppid()**
renvoie le pid (resp. ppid) du pus exécutant l'appel
(résultats affichés par ps(1) -l (champs 3 et 4 : PID PPID))

int getuid() **int getgid()**
renvoie le uid (resp. gid) de l'utilisateur réel :
Utilisateur (réel) : identifie l'U. ayant créé le pus
Utilisateur **effectif** : propriétaire du fichier exécutable lorsque ce dernier a le droit d'accès s "set uid" (chmod u+s monexec)

int geteuid() **int getegid()**
renvoie le euid (resp. egid) de l'utilisateur effectif.

La notion d'utilisateur effectif permet, sous Unix, à un utilisateur X n'ayant pas de droit sur des fichiers de Y d'exécuter un programme conçu par Y qui accède à ces fichiers. Cela permet à Y d'élaborer par programme un système de protection plus fin que les droits rwx.

exemple :
```
93 huchard@sequoia % cat effectif.c
/* illustre le fonctionnement du droit setuid */
#include <stdio.h>
main()
{printf("Ce pus de pid : %d tente d'afficher le fichier \"cache\"\n",getpid());
printf("Utilisateur reel : %d ; \tUtilisateur effectif : %d \n",getuid(),geteuid());
system("cat cache"); }
```

```
94 huchard@sequoia % effectif
Ce pus de pid : 16980 tente d'afficher le fichier "cache"
Utilisateur reel : 3120 ;      Utilisateur effectif : 3120
cache: Permission denied
95 huchard@sequoia % cat cache
cache: Permission denied
96 huchard@sequoia % ls -l
total 47
-rw-------  1 meynard      169 Oct 19 13:33 cache
-rwxr-xr-x  1 meynard    43140 Oct 19 13:35 effectif
-rw-r--r--  1 meynard      157 Oct 19 13:34 effectif.c

sequoia.lirmm.fr:meynard 87 > chmod u+s effectif

98 huchard@sequoia % ls -l
total 47
-rw-------  1 meynard      169 Oct 19 13:33 cache
-rwsr-xr-x  1 meynard    43140 Oct 19 13:35 effectif
-rw-r--r--  1 meynard      157 Oct 19 13:34 effectif.c
99 huchard@sequoia % effectif
Ce pus de pid : 16988 tente d'afficher le fichier "cache"
Utilisateur reel : 3120 ;      Utilisateur effectif : 3155
Ce fichier a l'unique droit en lecture/ecriture pour le
proprietaire. Il est donc invisible sauf si on l'utilise
avec le programme "effectif" qui a le droit set_uid.
100 huchard@sequoia.lirmm.fr:systeme1 % cat cache
cache: Permission denied
```

6.3.2 Création/Destruction

int fork()
crée un pus fils héritant de son père :
- le même segment de code (réentrance)
- le même environnement
- les mêmes descripteurs de fichiers y compris le terminal d'attachement (même ptr !)
- une copie des segments de données (stat., dyn., syst.)

Le résultat de la fonction est 0 pour le fils, pid du fils pour le père (-1 si échec). Les 2 pus sont asynchrones. La mort du père entraîne le retour au grand-père (cshell) et l'adoption du fils par le pus init (pid 1).

void exit(int status)
termine le pus et donne status comme valeur de retour (par convention 0 si pas de problème). Ce status est transmis au père (réel ou adoptif).

unsigned sleep(unsigned nsec)
suspend le pus pendant nsec secondes.

Exemple:

```
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 25 > cat exemplefork.c
/* exemple de fork */
#include <stdio.h>
main()
{int n;
if ((n=fork())==-1)
      {fprintf(stderr,"echec du fork...");exit(1);}
else if (n==0)
      {printf("debut du pus fils de pid: %d; de ppid: %d; \
de uid: %d\n",getpid(),getppid(),getuid());
      sleep(3); printf("fin du fils de nouveau ppid : %d\n",getppid());
      exit(0);
      }
else   {printf("debut du pus pere de pid: %d; de fils: %d; \
de uid: %d\n",getpid(),n,getuid());
      sleep(1); printf("fin du pere\n");exit(0);
      }
}
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 26 > exemplefork
debut du pus pere de pid: 23664; de fils: 23665; de uid: 3155
debut du pus fils de pid: 23665; de ppid: 23664; de uid: 3155
fin du pere
sequoia:/h-dif/meynard/systeme1 27 > fin du fils de nouveau ppid : 1
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 27 >
```

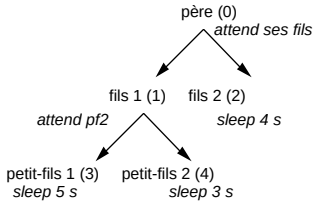
Attention, la manipulation des fichiers partagés par des pus apparentés doit être réalisée par les appels systèmes, sinon il risque fort d'y avoir des problèmes lors de la recopie des tampons (buffers) associés aux fonctions fputs, fprintf…

6.3.3 Synchronisation

int wait(int *ptstatus) *(en réalité types + complexes)*
Permet au père d'attendre (de se bloquer) la terminaison (exit) d'**un** de ses fils. S'il n'existe pas de fils ou que l'un d'eux a déjà terminé, wait retourne aussitôt -1. Sinon il retourne le pid du fils terminé et positionne *pstatus au code d'exit (octet de poids fort du mot de poids faible de *pstatus=octet de poids faible d'exit !).

int waitpid(int pid, int *pstatus, int options)
Permet d'attendre un fils particulier (pid) de façon bloquante (options==0) ou non bloquante (options==WNOHANG *include <sys/wait.h>*).

Exemple



Programmation :

```
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 27 > cat waitex.c
/* waitex.c */
#include <stdio.h>
#include <sys/wait.h>
```



```
main()
{int n,i,j;
printf("debut pere\n");
for (i=0; i<2; i++)
  if ((n=fork())==-1)
    {fprintf(stderr,"echec du fork num : %d",i);exit(1);}
  else if (n==0)
       switch (i) {
       case 0:{printf("debut fils 1\n");
              for (i=0; i<2; i++)
               if ((n=fork())==-1)
    {fprintf(stderr,"echec du fork num : %d",i);exit(1);}
               else if (n==0)
                  switch (i) {
                    case 0:{printf("debut petit fils 1\n");
                    sleep(5);printf("fin petit fils 1\n");
                    exit(3);}
                    case 1:{printf("debut petit fils 2\n");
                    sleep(3);printf("fin petit fils 2\n");
                    exit(4);}
                  }
              while (waitpid(n,&i,WNOHANG)!=n) {printf("fils 1 attend fin de petit-fils
2\n"); sleep(1);}
              printf("fin fils 1\n");
              exit(1);
              }
       case 1:{printf("debut fils 2\n");
                    sleep(4);printf("fin fils 2\n");
                    exit(2);}
        }
for (i=0; i<2; i++) n=wait(&j);
j=j>>8;
printf("dernier pus fils de pid : %d, de code retour : %d\n",n,j);
printf("fin du pere !\n");
exit(0);
}
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 28 > waitex
```

```
debut pere
debut fils 1
debut fils 2
fils 1 attend fin de petit-fils 2
debut petit fils 1
debut petit fils 2
fils 1 attend fin de petit-fils 2
fils 1 attend fin de petit-fils 2
fils 1 attend fin de petit-fils 2
fin petit fils 2
fin fils 1
fin fils 2
dernier pus fils de pid : 1327, de code retour : 2
fin du pere !
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 29 > fin petit fils 1
```

Attention, ces exemples d'exécution ne sont pas les seuls possibles : différents entrelacements possibles.

6.3.4 Recouvrement

Les pus fils ne peuvent pas toujours être codés dans leur père : tout fichier exécutable est lancé depuis le csh. On utilise la technique de recouvrement ("overlay") du code du fils (qui est alors une copie de celui du père) pour **exec**uter un fichier exécutable. Après un appel à l'une des fonctions de bibliothèque exec, l'exécution du pus se poursuit à la 1° instruction du fichier recouvrant.

Les exec(3) sont divisés en 2 catégories de 3 types :

- catégorie l si le nb de paramètres est connu, v sinon ;
- type e si on passe un environnement, p si on recherche dans la variable PATH, rien sinon.

On décrira ici l'appel système execve(2)

void execve(char *fic_exec, *argv[], *env[])
• fic_exec référence (relative|absolue) un fichier exécutable binaire ou script. Pour un script, sa 1° ligne doit être #!/bin/csh.
• argv est un tableau de pointeurs sur les paramètres à passer. Attention le 1° paramètre (d'indice 0) doit être le suffixe de la référence.
• env est un tableau de chaines (NOM=valeur) définissant les variables d'environement.
Le fichier démarre son exécution par main(nbpar,argv,env).

Le pus ainsi modifié conserve la plupart de ses attributs : descripteurs, terminal, ids, …

exemple :
prog listant les paramètres passés puis supprimant le dernier pour se recouvrir lui-même! (overlay récursif)

```
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 23 > cat execex.c
/* execex.c */
#include <stdio.h>
main(int n, char *argv[],char *env[])
{char **p=argv;int i=0;
while (i!=n)
        {printf("%s ",p[i]);
        i++;
        }
printf("\n");
p[--i]=NULL;
if (i!=0) execve(argv[0],argv,env);
   else exit(0);
}
```



```
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 24 > execex
execex
sequoia:/h-dif/meynard/systeme1 25 > execex titi 2 3 toto 5
execex titi 2 3 toto 5
execex titi 2 3 toto
execex titi 2 3
execex titi 2
execex titi
execex
```

6.3.5 Communication par tube

Structure de données FIFO de taille bornée permettant la communication "synchronisée" entre processus. La FIFO est manipulée par 2 descripteurs de fichiers (lecture, écriture).

int pipe(int tube[2])

Au retour, si le résultat==0, on a
• tube[0] correspond au descripteur de lecture
• tube[1] correspond au descripteur d'écriture
sinon (-1) l'appel a échoué.

L'accès au tube se fera ensuite par appels au primitives read et write. En cas de tube vide (res. plein), read (resp. write) sera bloqué. Lorsque tous les pus ont fermé tube[1], et que le tube est vide, un read retournera 0 (EOF). Les tubes sont principalement utilisés dans les filtrages de fichier.

Exemple : filtrage des chiffres dans une chaîne

```
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 24 > cat tubeex.c
/* tubeex.c */
#include <stdio.h>
main(int n, char *argv[], char *env[])
{int tube[2]; char c, *ch;
if (pipe(tube)!=0) {fprintf(stderr,"echec du pipe !\n");exit(1);}
if ((n=fork())==-1)
   {fprintf(stderr,"echec du fork ");exit(2);}
  else if (n==0) /* pus fils */
  {
  close(tube[1]); /* jamais d'ecriture */
  while (read(tube[0],&c,1)!=0)
       if (c<'0' || c>'9') printf("%c ",c);
  exit(0);
  }
/* pus pere */
close(tube[0]);
while ((c=getchar())!=EOF) write(tube[1],&c,1);
close(tube[1]);
wait(NULL);
exit(0);
}sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 25 > tubeex
abc123DEF456!()789.
a b c D E F ! ( ) .
ab12cd34
a b c d
```

Remarquons que si le fils se recouvre grâce à un exec, il perd la **connaissance** de ses descripteurs.
Il faudra donc lui transmettre en paramètres …
Cette méthode est valable pour les descripteurs de fichiers mais aussi pour toutes les valeurs de variables que le fils veut transmettre à son propre overlay.

Exemple: même tube avec un fils réalisant un exec

```
h-dif/meynard/systeme1 26 > cat tubexec.c
/* tubexec.c */
#include <stdio.h>
main(int n, char *argv[], char *env[])
{int tube[2]; char c;
if (pipe(tube)!=0) {fprintf(stderr,"echec du pipe !\n");exit(1);}
if ((n=fork())==-1)
    {fprintf(stderr,"echec du fork ");exit(2);}
  else if (n==0) /* pus fils */
  {char *newargv[3],tube0[10];
   sprintf(tube0,"%d",tube[0]);
   newargv[0]="afftube";
   newargv[1]=tube0;
   newargv[2]=NULL;
   close(tube[1]); /* jamais d'ecriture */
   execve("afftube",newargv,env);
  }
/* pus pere */
close(tube[0]);
while ((c=getchar())!=EOF) write(tube[1],&c,1);
close(tube[1]);
wait(NULL);
exit(0);
}sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 27 > cat afftube.c
/* afftube.c */
#include <stdio.h>
main(int n, char *argv[], char *env[])
{char c; int tube;
if (n!=2) {fprintf(stderr,"1 parametre SVP !\n");exit(1);}
tube=atoi(argv[1]);
while (read(tube,&c,1)!=0)
       if (c<'0' || c>'9') printf("%c ",c);
exit(0);
}sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 28 > tubexec
abcde1235ACDR9856!)(
a b c d e A C D R ! ) (
sequoia.lirmm.fr:meynard:/h-dif/meynard/systeme1 30 >afftube 0
abc123def456('$
a b c d e f ( ' $
```

7. Conclusion

Il est absolument indispensable de comprendre les **concepts de base** des machines et des systèmes afin :

- de manipuler différents systèmes d'exploitation en comprenant les différences et les ressemblances ;
- d'**évaluer** correctement les résultats numériques fournis par les machines (précision) ;
- de **programmer** intelligemment les algorithmes dont on a minimisé la complexité (des E/S fréquentes peuvent ruiner un algorithme d'une complexité inférieure à un autre) ;
- de pouvoir **optimiser** les parties de programme les plus utilisées en les réécrivant en langage de bas niveau ;
- d'écrire des compilateurs ou des interprèteurs performants même si ceux-ci sont écrits en langage de haut niveau ;
- de se préparer à l'arrivée de nouveaux paradigmes de programmation (**programmation parallèle**) ;
- d'oser utiliser des systèmes d'exploitation libres !

6.3.6 Conclusion

Nous n'avons vu qu'une infime partie de la gestion des pus Unix. Il manque, entre autres, la synchronisation par signaux (signal, wait), les priorités (nice), d'autres outils de communication (files de msg, sémaphores, mémoire partagée, sockets), …

- • Objectif de ce chapitre : sensibilisation à la programmation des pus sous Unix
- • peu de concepts théoriques bien qu'**il existe** de nombreux concepts formalisés ;
- • Autres SE : autres outils de programmation ;
- • Plus loin sous Unix : consulter xman et développer …