

Prénom :

Nom :

Groupe TD :

Date :



PROGRAMMATION SYSTÈME

F. Merciol, M. Le Lain, D. Lesage

UBS - IUT

BUT Informatique – 2^e année

Vannes – 2023/2024

Support de cours **étudiant**

<http://r305.merciol.fr/>

(page gauche vide)

Table des matières

Table des figures	4	9.3 Arrière-plan	20
Liste des tableaux	5	9.4 Variables	20
Listings	5	9.5 Syntaxe	20
I Introduction	8		
1 Structuration du cours	8		
2 Évaluation du module	9		
2.1 Calcul	9	10 Annexe	21
2.2 Pollution	10	10.1 mkNamedSem	21
3 Système d'exploitation	10	10.2 rmNamedSem	21
		10.3 namedSemP	21
		10.4 namedSemV	21
II Processus	13		
4 Ressource	13		
4.1 Caractéristiques	13	III Introduction à la programmation réseau	23
4.2 Cohérence	13	11 Brève histoire de l'Internet	23
4.3 Inter-blocage	14	12 Ethernet	24
4.4 Droits	14	13 IP	25
4.5 Limites	15	14 TCP/IP UDP/IP	26
5 Ordonnanceur	15		
6 Processus	16	15 Les Sockets	26
7 IPC	18	15.1 TCP/IP	26
7.1 Sémaphore	18	15.2 UDP/IP	27
7.2 Mémoire	19		
7.3 Message	19	16 Protocoles IP	28
8 L'attente passive	19	17 Annexe	28
8.1 Select	19		
8.2 inotifywait	19		
9 shell	20		
9.1 ps	20	IV Les tâches	30
9.2 Signaux	20	18 tâches et processus	30
		18.1 Histoire	30
		19 Crédit	31
		19.1 Thread	31
		19.2 Runnable	32
		19.3 Tâche anonyme	32
		19.4 Propriétés	32
		20 Synchronisation	33
		20.1 Join	34
		20.2 Moniteur	34
		20.3 synchronized	34

20.4 wait/notify	35	28.7 Expression régulière	51
20.5 Sémaphore	36	28.8 Internationalisation	52
V La mémoire	37	28.9 XML	53
21 Caractéristiques	37	29 Fichiers binaires	54
21.1 Vive	37	30 Annexe	54
21.2 Des processus	38		
21.3 Partagée	38		
21.4 D'échange	39		
21.5 Morcelée	39		
22 Segmentation	40	VII Système de gestion de fichiers	56
23 Pagination	41	31 Besoin de fichiers	56
24 Optimisations	42	31.1 Désignation	56
24.1 Taille mémoire des processus	42	31.2 Typage	56
24.2 Calcul d'adresse	42	31.3 Hiérarchie	57
24.3 Partage de codes	42	31.4 Intégrité	57
24.4 Chargement à la demande	42	31.5 Confidentialité	58
24.5 Partage de données	43	31.6 Fiabilité	58
24.6 Mise en mémoire secondaire	43	31.7 Disponibilité	58
24.7 Anomalie de Bélády	44	32 Support physique	59
VI Les entrées-sorties	45	33 Partition	59
25 Les périphériques	45	34 Table de partition	60
26 Lecture/Ecriture sous Unix	46	35 Structure	60
26.1 Bufferisation	46	35.1 Unité d'allocation	60
26.2 Les fichiers	47	35.2 Descripteur de fichiers	61
26.3 Les répertoires	48	35.3 Table d'implantation	61
27 Les formats de fichier	48	35.4 Catalogue	61
28 Fichiers textes	49	35.5 Table d'allocation	61
28.1 Codage des caractères	49	36 Le SGF Unix	61
28.2 1 champ par ligne	49	36.1 Unité d'allocation	62
28.3 2 champs par ligne	50	36.2 Inode	62
28.4 n champs par ligne	50	36.3 Table d'implantation	62
28.5 Découpage en jetons	50	36.4 Répertoire	63
28.6 Modèle de texte	51	36.5 Volume Unix	63
		37 Montage	63
		38 Type	64
		39 shell	64

Table des figures

1	Architecture mémoire	10
2	Virtualisation	11
3	Logo docker	11
4	Dîner de philosophes	14
5	Exemple multi-cœurs	16
6	Chiffre d'affaires des GAFAM	23
7	Rénater	24
8	Exemple de congestion de trames	24
9	Classes IPV4	25
10	Trames IP	26
11	TCP/IP	26
12	UDP/IP	27
13	Synchronisation de vie d'une tâche	34
14	Synchronisation ré-entrant	34
15	RAM	37
16	Tore magnétique	37
17	Zone mémoire	38
18	Mémoire partagée	39
19	Exemple de répartition de mémoire	40
20	Mémoire segmentée	40
21	Mémoire paginée	41
22	Connecteurs	45
23	Fork avec tampon (sans <i>flush</i>)	47
24	Fork sans tampon (avec <i>flush</i>)	47
25	Empilement de fonctionnalité	50
26	Architecture d'un disque	59
27	Géométrie de disque	59
28	Table d'implantation	62
29	Exemple d'inodes	63

Liste des tableaux

1	Découpage en chapitres	9
2	Grille d'évaluation de QCM	9
3	Valeurs caractéristiques de QCM	9
4	Correspondance ISO / l'Internet	24
5	Comparaison notify/notifyAll	36
6	Table de segmentation	40
7	Calcul de segment	41
8	Table de pagination	41

9	Calcul de page	42
10	Séquence de pages de référence pour comparaison	43
11	Algorithme "Optimal"	43
12	Algorithme "FIFO"	43
13	Algorithme "LRU"	44
14	Algorithme "Aléatoire"	44
15	Algorithme "LFU"	44
16	Anomalie de Bélády	44
17	Exemple de n° majeur	46
18	Exemple de n° majeur/mineur	46
19	Mode d'ouverture de fichier	47
20	Marqueur de fin de ligne	48
21	Codage de caractères	49
22	Les types des fichiers sous Unix	64

Listings

1	Installation de Docker	11
2	Utilisation de Docker	12
3	Le shell de la mort	15
4	Prototype de la fonction select	19
5	Attente passive en shell sur un répertoire	19
6	Compilation des outils sémaphores	21
7	Crée un sémaphore nommé	21
8	Supprime un sémaphore nommé	21
9	Puis-je ? sur un sémaphore nommé	21
10	Vas-y ! sur un sémaphore nommé	21
11	Code Java d'un serveur de sockets	26
12	Code Java d'un client de socket	27
13	Code Java de réception d'un datagramme	27
14	Code Java de l'envoi d'un datagramme	27
15	Réception de datagramme en Java	28
16	Envoyer de datagramme en Java	28
17	Réception de datagramme en Java	29
18	Envoyer de datagramme en Java	29
19	Héritage de Thread	31
20	Implantation de Runnable	32
21	Tâche en classe anonyme	32
22	Limitation à une tâche	33
23	Synchronisation sur la vie d'une tâche	34
24	Synchronisation d'objets	35
25	Synchronisation de méthodes	35

26	Sémaphore avec un moniteur	36
27	Réglage <i>wait</i> de la classe <i>Semaphore</i>	36
28	Réglage <i>notify</i> de la class <i>Semaphore</i>	36
29	Effet de bord des tampons avec <i>fork</i>	47
30	Code Java utilisant <i>readLine</i>	48
31	Code C utilisant <i>fscanf</i>	49
32	Création d'un fichier propriété Java	50
33	Création d'un fichier propriété Java	50
34	Création d'un fichier propriété Java	50
35	Exemple de message formaté en Java	51
36	Extrait documentation expression régulière	52
37	Navigation XML en PHP	53
38	Création XML en PHP	53
39	Exemple de lecture XML en Java	53
40	Exemple d'écriture XML en Java	53
41	Exemple d'utilisation XML en Java	53
42	Exemple de lecture d'objets Java	54
43	XML en PHP	54
44	XML en Java	54
45	Recherche de fichiers non modifiables	57
46	Bit s	57

Nous commençons une séquence de formation visant à réaliser des programmes s'appuyant sur les fonctions fondamentales d'un système d'exploitation. Le contenu des cours du **Bachelor Universitaire de Technologie Informatique** est décrit dans le **Programme National** (<http://www.iut-informatique.fr/>). Le programme est composé de ressources. Et celle que nous débutons est décrite sous la référence "R3.05 : Programmation système." Elle fait partie de la "Compétence 3 : Administrer des systèmes informatiques communicants complexes". Elle est présente dans tous les parcours actuels du BUT au semestre 3.

La ressource prévoit 30 h (15 h TD et 15 h TP). À l'IUT de Vannes nous ne disposons que de 2,5 créneaux de 1h30 pendant 6 semaines. Nous ne pourrons donc faire que : 4,5 h CM / 9 h TD / 9 h TP soit 22,5 h. Vous devrez combler ce manque par un travail personnel (préparation de vos TP en particulier). Des aspects de programmation système pourront également apparaître lors de SAÉ (**Situation d'Apprentissage et d'Évaluation**).

L'objectif de cette ressource est de comprendre la structure d'une application client-serveur et de comprendre les mécanismes bas niveaux, mis en œuvre dans une application multi-tâches. Cette ressource permettra de découvrir le développement d'applications multi-processeurs, de comprendre et de traiter les problèmes de synchronisation et d'utiliser des outils de communication internes aux processus, mais aussi externes, via les API de transport.

Le texte officiel précise que les savoirs de référence étudiés seront :

- Fonctionnement du système (par ex. : pagination, mémoire virtuelle, systèmes de fichiers...)
- Gestion de processus (par ex. : ordonnancement, synchronisation, threads...)
- Programmation client-serveur (par ex. : IPC, interface socket, protocoles applicatifs...)
- Les différents savoirs de référence pourront être approfondis

Les mots clefs de la ressource sont : Mécanismes bas niveaux, Processus, Client-serveur.

Si l'acronyme IPC évoque pour vous l'Indice des Prix à la Consommation, ou que le terme de pagination vous semble être réservé à l'imprimerie, il n'y a pas de doute : ce support va enrichir vos connaissances. Et même, si vous ne projetez pas briller en société en décrivant la beauté d'une synchronisation de processus, cela aura moins le mérite de vous aider à obtenir votre diplôme. Ce n'est pas rien !

Pour s'adapter au mieux au programme, ce support de cours se compose des parties suivantes :

- **Introduction** : sur la structuration de cette ressource enseignée et d'un survol des fonctions principales d'un système d'exploitation indispensable à l'exécution de programme,
- **Processus** : sur le partage des ressources et particulièrement l'ordonnanceur des processus,
- **Réseau** : sur la présentation de l'interconnexion des systèmes d'exploitation et la présentation d'un outil de communication : les "sockets",
- **Tâches** : sur la gestion de multiples activités au sein d'un processus (également appelé processus légers) et leur synchronisation,
- **Mémoire** : sur différentes organisations de mémoire pour une gestion efficace des ressources,
- **Entrée/Sortie** : sur la temporisation des écritures et le formatage des données,
- **Fichiers** : sur les systèmes de gestion de fichiers avec le cas particulier d'Unix.

A quoi peut bien servir ces compétences et l'université a-t-elle de l'expérience dans le domaine ?

Très simplement, le réseau d'éducation populaire Framasoft a lancé le CHATONS (Collectif d'Hébergeurs Alternatifs, Transparents, Ouverts, Neutres et Solidaires) dans l'objectif de "Dégoogliser Internet" (rien que ça). Il se trouve que dans le Morbihan le collectif Kaz (dont la moitié des fondateurs sont de l'UBS) fait partie du CHATONS. On peut ajouter que la moitié des administrateurs système de Kaz sont de votre département informatique. Donc d'expérience, oui ces connaissances servent au quotidien.

Première partie

Introduction

Le système universitaire dans lequel nous nous trouvons, s'appuie sur les notes plutôt que la transmission de connaissance. Il est donc indispensable en début de formation maximiser vos chances de réussite en prenant en compte le système dévaluation. L'objet n'est pas de prendre les étudiants en traite en leur posant des questions inattendues sur une phrase perdue au milieu du cours. Pour que chacun soit évalué de façon équitable, il ne doit pas subsister de quiproquo. Cette section aborde l'organisation du cours et son évaluation.

1 Structuration du cours

Le cadre national nous contraint à de courtes (45 minutes) séances en amphithéâtre. L'avantage est que cela correspond au temps d'attention que l'on peut avoir en continu. L'inconvénient est un manque de temps pour exposer des notions communes. En amphithéâtre, il ne sera pas possible de lire tout le contenu du support. Il vous faudra donc l'étudier en dehors de cours.

Concernant vos facultés d'apprentissage, les travaux de Michel Desmurget, chercheur en neurosciences cognitives, sont incontournables. Il a écrit "TV Lobotomie" (2011), où il collecte les résultats sans appel de toutes les études scientifiques qui montrent que les écrans : diminue les facultés mentales (10%), diminue l'espérance de vie, pousse au tabagisme, provoque des grossesses non désirées...

En particulier, les écrans nuisent à l'enseignement (à l'école et ailleurs). Pour mémoriser une leçon il faut que cela coûte "du sucre au cerveaux" (un effort). Cela ne veut pas dire que l'on doit apprécier la souffrance. Au contraire, jouer à un jeu de société ou de plein air demande de la concentration et de l'effort. Et c'est au final très agréable.

Ces cours vous paraîtront difficiles. Ils demanderont de la concentration. Mais vous apporteront un savoir en contrepartie.

Vous pouvez consulter des conférences en ligne sur "TV Lobotomie" (site <http://fls56.org/>) ou sur "Atelier Éducation 2016-2017 - 1 - L'école et le numérique".

Le découpage se fera en 6 thèmes suivant la table 1.

Semaine	CM	TD	TP
1	Processus	fork/exec	shell
2	Réseaux	asynchrone	Java
3	Tâches	synchronisation	Java
4	Mémoire	allocation	Java
5	Entrée/Sortie	formatage	Java
6	Fichiers	SF Unix	Python

TABLE 1: Découpage en chapitres

2 Évaluation du module

Au delà de la notation, l'évaluation consiste à mesurer les nouvelles connaissances acquises au cours de ce module. Vous trouverez dans ce document un Questionnaire à Choix Multiple (QCM) qui ne sera pas noté. Vous remplirez ce QCM avant le premier cours, puis à nouveau après le dernier. Vous serez à même de juger de votre progression. Vous restituerez ce QCM au dernier cours pour réaliser une évaluation de groupe. En revanche d'autres QCM (un par chapitre) seront notés.

À noter que l'on vous distribuera les QCM en deux exemplaires :

- le 1^{er} est pour vous (en souvenir)
- le 2nd à remettre à l'enseignant en début de chaque TD (fourni en amphi la semaine précédente)

Pensez à recopier les réponses sur votre exemplaire.

Il y aura également des évaluations chiffrées en contrôle continu :

- coef 1 : QCM en début de TD
- coef 1 : Assiduité en TD et TP (1 point si présent et à l'heure par créneau / 12)
- coef 1 : Régularité du travail (1 point par TP rendu / 6)
- coef 4 : 2 Comptes-rendus de TP
- Pénalité de pollution : -2 points par TP rendu via l'Internet (courriel, moodle ...)
- Bonus écologique en cas de rendu papier :
 - +1 point pour les TP rendus en binôme.
 - +1 point pour les TP rendus en 2 colonnes et recto-verso (4 pages par feuille)

- Pénalité de non-respect des règles :

- identité : -1 point par élément manquant (nom, prénom, groupe TD, identité du TP)
- retards : -1 point si non rendu le jour même et -1 par semaine supplémentaire

2.1 Calcul de QCM

Pour ceux qui apprécient la carotte et le bâton, disons que les bonnes réponses donnent des points positifs (ex 7 pts) et les mauvaises réponses des points négatifs (ex -3pt). Comme il serait aberrant de ne pas faire la différence entre quelqu'un qui s'abstient et quelqu'un qui fait exprès de mal répondre à toutes les questions, les notes sont ramenées à des valeurs positives. Dans la grille de calcul ci-dessous, la ligne du haut indique le nombre de bonnes réponses, la colonne de gauche indique le nombre de réponses fausses. Comme il peut y avoir des absences de réponses, la grille à une forme triangulaire.

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
points	0	7	14	21	28	35	42	49	56	63	70	77	84	91	98	105	112	119	126	133	140
0	0	6,0	6,7	7,4	8,1	8,8	9,5	10,2	10,9	11,6	12,3	13,0	13,7	14,4	15,1	15,8	16,5	17,2	17,9	18,6	19,3
1	-3	5,7	6,4	7,1	7,8	8,5	9,2	9,9	10,6	11,3	12,0	12,7	13,4	14,1	14,8	15,5	16,2	16,9	17,6	18,3	19,0
2	-6	5,4	6,1	6,8	7,5	8,2	8,9	9,6	10,3	11,0	11,7	12,4	13,1	13,8	14,5	15,2	15,9	16,6	17,3	18,0	
3	-9	5,1	5,8	6,5	7,2	7,9	8,6	9,3	10,0	10,7	11,4	12,1	12,8	13,5	14,2	14,9	15,6	16,3	17,0		
4	-12	4,8	5,5	6,2	6,9	7,6	8,3	9,0	9,7	10,4	11,1	11,8	12,5	13,2	13,9	14,6	15,3	16,0			
5	-15	4,5	5,2	5,9	6,6	7,3	8,0	8,7	9,4	10,1	10,8	11,5	12,2	12,9	13,6	14,3	15,0				
6	-18	4,2	4,9	5,6	6,3	7,0	7,7	8,4	9,1	9,8	10,5	11,2	11,9	12,6	13,3	14,0					
7	-21	3,9	4,6	5,3	6,0	6,7	7,4	8,1	8,8	9,5	10,2	10,9	11,6	12,3	13,0						
8	-24	3,6	4,3	5,0	5,7	6,4	7,1	7,8	8,5	9,2	9,9	10,6	11,3	12,0							
9	-27	3,3	4,0	4,7	5,4	6,1	6,8	7,5	8,2	8,9	9,6	10,3	11,0								
10	-30	3,0	3,7	4,4	5,1	5,8	6,5	7,2	7,9	8,6	9,3	10,0									
11	-33	2,7	3,4	4,1	4,8	5,5	6,2	6,9	7,6	8,3	9,0										
12	-36	2,4	3,1	3,8	4,5	5,2	5,9	6,6	7,3	8,0											
13	-39	2,1	2,8	3,5	4,2	4,9	5,6	6,3	7,0												
14	-42	1,8	2,5	3,2	3,9	4,6	5,3	6,0													
15	-45	1,5	2,2	2,9	3,6	4,3	5,0														
16	-48	1,2	1,9	2,6	3,3	4,0															
17	-51	0,9	1,6	2,3	3,0																
18	-54	0,6	1,3	2,0																	
19	-57	0,3	1,0																		
20	-60	0,0																			

TABLE 2: Grille d'évaluation de QCM

Dans le cas d'un QCM à 4 réponses, si vous répondez aléatoirement sur 4 réponses, vous aurez 1 bonne et 3 mauvaises. Ou sur 20, 5 bonnes et 15 mauvaises. Notez les valeurs particulières :

réponses	en points	note sur 20
20 mauvaises	-60	0
20 aléatoires	-10	5
pas de réponse	0	6
20 bonnes	140	20

TABLE 3: Valeurs caractéristiques de QCM

Pour ceux qui veulent voir ce barème de façon positive, il suffit de dire que vous avez 1 point par bonne réponse et 0,3 point par abstention (les mauvaises réponses ne sont pas comptées). Le tableau est strictement le même.

Une réponse au hasard ne vous donne que 0,25 points (en moyenne 1 chance sur 4). Vous n'avez donc aucun intérêt à répondre au hasard. Car, à chaque question, vous avez plus de point à vous abstenir que de répondre au hasard. Le mieux est évidemment de donner la bonne réponse.

2.2 Calcul de pollution

La question revenant chaque année, je donne ici quelques éléments sur le surcoût de la pollution induite par l'utilisation de l'Internet quand vous rendez vos TP de cette manière.

Sachant qu'un groupe de 2^e année est constitué de 4 groupes TD de 24 étudiants. Sachant qu'il y a 6 TP à rendre, de 2 feuilles de papier (8 pages). Sachant que les TP sont conservés 10 ans.

Calculez le nombre de feuilles stockées. (Quel formidable piège à carbone !)

Calculez la consommation énergétique consommée par ces feuilles pendant 10 ans. Il faut tenir compte des années bissextiles (un indice : $0 \times 365 = 0$).

Pour le rendu électronique, faites une estimation du poids en Ko des comptes-rendus (8 pages avec graphiques). Faites le même calcul de stockage sur 10 ans des rendus électroniques.

Calculez la quantité de disques durs nécessaire à acheter. Calculez la consommation électrique correspondante. Ajoutez les systèmes de sauvegarde et la redondance matérielle (n'oubliez pas que le papier est plus fiable et est insensible aux virus).

Donnez une estimation pour le passage à un enseignement "tout numérique" (on prendra 20% de la population française comme étant scolarisée). Donnez la consommation en nombre de réacteurs nucléaires de 1 300 mégawatts.

Vous n'aviez pas conscience de la pollution de l'Internet ? Vous n'aviez pas non plus conscience que le papier ne consommait pas d'énergie une fois imprimé ? Alors, vous allez apprendre bien d'autres choses dans ce cours.

Vous voici fin prêts à commencer ce cours... .

3 Système d'exploitation

Dans une représentation en couches de niveau d'abstraction d'un ordinateur (physique, logique ...), le système d'exploitation gère le matériel suivant les demandes des applications.

Définition 1 (système d'exploitation). Un système d'exploitation (en anglais *Operating System*) permet d'exploiter (de tirer parti) des ressources d'une machine. Il est composé d'un ensemble d'automatismes que l'on peut déclencher pour faire fonctionner tous les organes de la machine.

Définition 2 (périphérique). Un périphérique (en anglais *device*) est un composant matériel dédié à une fonction.

Définition 3 (pilote). Un pilote (en anglais *driver*) est un composant logiciel communiquant avec un périphérique suivant l'interface du constructeur (signaux électriques) et offrant toutes les fonctions disponibles pour les applications.

Définition 4 (gestion des droits). Les systèmes d'exploitation multi-utilisateurs garantissent une gestion des droits. Ils mettent en place un moyen d'authentification (mot de passe sous Unix). Pour chaque ressource, il est spécifié un propriétaire, ainsi que les groupes d'utilisateurs qui peuvent y avoir accès.

Définition 5 (groupe d'utilisateurs). Pour faciliter la gestion des droits, des profils génériques sont définis : des groupes (*/etc/group* sous Unix). Un mécanisme indique l'appartenance d'un utilisateur à un groupe (définitivement */etc/passwd* ou temporairement *newgrp* sous Unix).

Définition 6 (administrateur). Les groupes définissent des niveaux d'accès (ou priviléges) : disque, réseaux, messagerie... Un administrateur dispose de tous les droits (*root* sous Unix).

Pour réaliser ses propres fonctions, le système d'exploitation s'utilise lui-même. Les instructions définissant le programme du pilote de la mémoire sont enregistrés dans cette même mémoire. Le temps de calcul du programme d'ordonnancement des tâches est alloué par ce même Ordonnateur de tâches.

Il existe un nombre limité de fonctionnalités élémentaires.

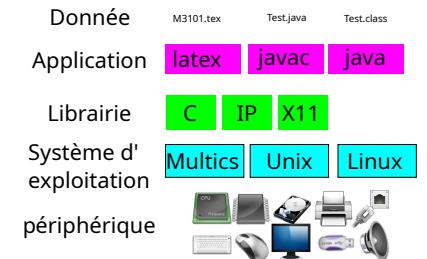


FIGURE 1: Architecture mémoire

Définition 7 (noyau). Le noyau (en anglais *kernel*) regroupe les fonctions élémentaires du système. C'est là que se réalise le contrôle d'accès et la répartition des ressources. Les processeurs peuvent fournir des instructions particulières pour offrir un mode spécifique d'exécution. Cela signifie que, même avec les instructions assembleurs appropriées, vous ne pourrez plus entrer dans le noyau une fois le système lancé.

Voici une liste d'équivalence pour les termes anglophones.

- *device* : périphérique
- *disk* : disque
- *driver* : pilote
- *kernel* : noyau
- *keyboard* : clavier
- *login* : identifiant de connexion
- *mouse* : souris
- *password* : mot de passe
- *root* : administrateur
- *screen* : écran
- *user* : utilisateur

Quelques exemples historiques :

- 1960 :
Premier ordinateur à lecteurs de cartes perforées (on y passe plus de temps à attendre qu'un périphérique termine son travail, qu'à exécuter des instructions).
- 1965 :
Le MIT crée Multics (MULTiplexed Information and Computing Service). Système multi-tâches et multi-utilisateurs préemptif (durée maximum d'exécution sans attendre qu'un programme daigne céder sa place).
- 1969 :
Dennis Ritchie des laboratoires Bell crée Unix pour lequel tous les périphériques sont considérés comme des fichiers. Les dates du système Unix commencent au 1/1/1970. Une généalogie des systèmes Unix se trouve sur https://fr.wikipedia.org/wiki/Berkeley_software_distribution.
Il invente également un langage pour le développer : le langage C.

À noter que les systèmes d'exploitation sont souvent accompagnés d'un langage de commande propre. Pour Unix, le langage d'origine est le "Borne Shell". La variante la plus souvent utilisée aujourd'hui sous linux est "bash".

Parmi tous les systèmes lequel choisir ?

Chacun a sa particularité. Il est possible aujourd'hui de ne pas choisir.

Définition 8 (virtualisation). La virtualisation permet de faire fonctionner plusieurs systèmes d'exploitation avec ses qualités sans nuire aux performances de la machine.

En effet, les processeurs actuels disposent d'un jeu d'instructions permettant de gérer leurs cohabitations (voir <https://fr.wikipedia.org/wiki/Virtualisation>).

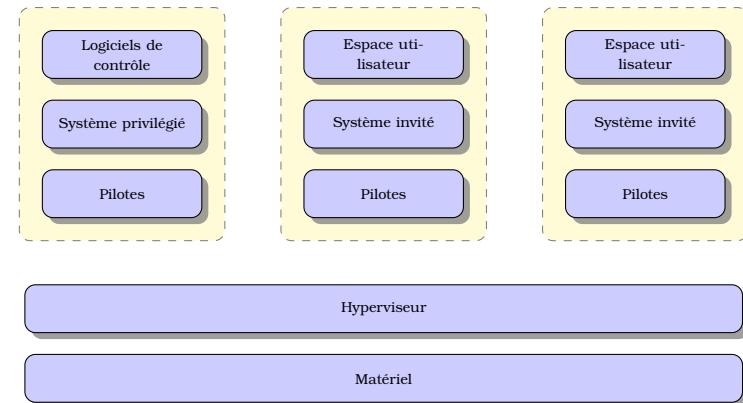


FIGURE 2: Virtualisation

Un exemple particulier notable est le logiciel libre Docker. Il permet de lancer des applications dans des conteneurs qui incluent un système d'exploitation. Il est particulièrement efficace, car il s'appuie sur LXC, cgroups et le noyau Linux de la machine hôte. Les puristes diront qu'il ne s'agit pas d'une virtualisation puisqu'il ne permet pas l'exécution de système autre que Linux.

L'installation se fait de la manière suivante :

```
1 $ sudo bash
# add-apt-repository "deb [arch=amd64] https://download.docker.com/linux/debian
# bullseye stable"
```



FIGURE 3: Logo docker

```
5 # apt-get update  
# apt-get install docker-ce docker-compose  
# systemctl status docker  
10 # usermod -a -G docker myLogin
```

Listing 1: Installation de Docker

Voici un exemple d'utilisation.

```
1 $ docker search debian  
$ docker pull debian  
5 $ docker images  
07:45:10$ docker images  
REPOSITORY                                     TAG      IMAGE  
ID      CREATED        SIZE  
debian          fe3c5de03486    2 weeks ago   124MB  
10 $ docker run -i -t debian /bin/bash  
root@197238dbc93:/#  
root@197238dbc93:/# apt-get update  
Get:1 http://security.debian.org/debian-security bullseye-security InRelease [44.1  
kB]  
...  
15 root@197238dbc93:/# apt-get install -y procps  
...  
20 root@197238dbc93:/# ps -aux  
USER      PID %CPU %MEM      VSZ      RSS TTY      STAT START  TIME COMMAND  
root         1  0.0  0.0    4092   3504 pts/0      Ss  17:46  0:00 /bin/bash  
root        379  0.0  0.0    6700   2952 pts/0      R+  17:48  0:00 ps -aux  
25 CTRL + P + Q  
$ docker ps -a  
CONTAINER ID     IMAGE      COMMAND      CREATED      STATUS      PORTS  
 NAMES  
197238dbc93     debian     "/bin/bash"  3 minutes ago  Up 3 minutes  
romantic_poincare  
30 $
```

Listing 2: Utilisation de Docker

Ce cours ne fera qu'effleurer la présentation des systèmes d'exploitation. Il y a tant de choses à dire ...

Deuxième partie

Processus

4 Ressource

4.1 Caractéristiques

Dans ce document, le terme *ressources* recouvre : soit des composants matériels, soit des sous éléments matériels que l'on manipule comme une abstraction identifiée par le système (fichier dans un disque, espace mémoire dans des composants électroniques, connexion via une carte réseau ...).

Concrètement des ressources sont des :

- processeurs (unité de calcul et de traitement)
- périphériques (connectés en interne à la carte mère ou en externe à l'unité centrale)
- fichiers (dans des périphériques de mémoire de masse)
- mémoires (gérées par des composants électroniques)
- canaux de communication (via un périphérique réseau)

Pour les systèmes d'exploitation de la famille Unix tout est fichier

- /proc/... : accès aux données des processus en cours d'exécution
- /dev/tty... : les écrans et claviers des terminaux
- /dev/sd... : les disques durs
- /dev/sr... : les lecteurs de disques optiques
- /dev/mem : la mémoire
- /dev/eth... : les cartes réseaux Ethernet
- y compris les...
répertoires et fichiers.

Donc, dans notre cas, il s'agit principalement d'un partage de fichiers, même si le partage de fichiers posera des contraintes différentes suivant leur nature.

Dès que l'on utilise un système d'exploitation multi-tâches (voire multi-utilisateurs), se pose des questions d'accès concurrents aux ressources. Ils sont de différentes natures :

- la cohérence des données
- l'inter-blocage
- la vérification des droits
- le contrôle des limites

4.2 Cohérence des données

Pour illustrer le problème de cohérence des données, prenons l'exemple suivant. Le client d'une banque a un compte courant (CC) et un livret d'épargne (LE). Chaque mois, il met de côté 10% de son compte (tâche 2). Disons qu'aujourd'hui, chacun des deux comptes a un solde de 50 €. Exceptionnellement ce mois ci, il sait qu'il devra consommer 10 € de plus et choisit de faire un virement supplémentaire (tâche 1).

- tâche 1
 - ① LE = LE - 10
 - ② CC = CC + 10
- tâche 2
 - ❶ tmp = 10%CC
 - ❷ LE = LE + tmp
 - ❸ tmp = 10%CC
 - ❹ CC = CC - tmp

Intuitivement, nous imaginons que l'argent ne s'évapore pas lors de transfert de compte. Il y a donc un invariant :

$$\text{CC} + \text{LE} = \text{constante} \text{ (100 € dans notre cas)}$$

Une exécution séquentielle des deux virements peut donner le résultat suivant :

Tâche 1	LE	CC	CC+LE	tâche 2
Solde	50	50	100	
❶ LE = LE-10	40			
❷ CC = CC+10		60		
fin	40	60	100	
			6	❶ tmp = 10%CC
			46	❷ LE = LE+tmp
			6	❸ tmp = 10%CC
			54	❹ CC = CC-tmp
			46	fin

L'invariant est respecté.

Cependant, lorsque le traitement est effectué de façon parallèle, l'imbrication des instructions pourrait donner le résultat suivant :

Tâche 1	LE	CC	CC+LE	tmp	Tâche 2
Solde	50	50	100		
				5	❶ tmp = 10%CC
❶ LE = LE-10	40				
❷ CC = CC+10		60			
	45			❸ LE = LE+tmp	
		6		❹ tmp = 10%CC	
	54			❺ CC = CC-tmp	
	45	54	99		fin
fin	45	54	99		

Définition 9 (intégrité des données). Un système d'exploitation doit garantir l'intégrité et la cohérence des données quel que soit l'ordre d'exécution des instructions.

Ce problème d'accès concurrent peut survenir avec tous les types de ressources. Par chance, cette situation n'arrive pas dans le cas de la mémoire vive, car le système d'exploitation isole l'espace d'adressage entre les processus au moment de leur exécution. Cependant, le cas est fréquent lorsque deux processus manipulent un même fichier. Par exemple, dans le cas de la modification simultanée de mots de passe (`/etc/passwd`) par plusieurs utilisateurs.

Un moyen de se prémunir de l'accès simultané à une même ressource est d'identifier les sections critiques et d'effectuer une réservation pour un usage exclusif en écriture.

Définition 10 (section critique). En programmation parallèle, une section critique est une portion de code qui accède à une ressource qui ne devrait pas être accédé par plus d'une activité à la fois. Elle doit être protégée par un mécanisme de synchronisation : `MUTEX`, sémaphore, ...

4.3 Inter-blocage

La réservation de ressources peut induire des inter-blocages. Pour l'illustrer, nous présentons l'exemple des 5 philosophes (voir https://fr.wikipedia.org/wiki/D%C3%A9iner_des_philosophes).

Cinq philosophes sont assis autour d'une table ronde sur laquelle se trouvent cinq plats de spaghetti et cinq fourchettes. Chaque philosophe a devant lui un plat de spaghetti et il lui faut deux fourchettes pour le manger. Un philosophe passe son temps à manger et à penser. Quand il a faim, il tente de prendre deux fourchettes, l'une à sa gauche et l'autre à sa droite. S'il obtient les deux fourchettes, il mange pendant un temps, puis repose les fourchettes et ensuite se remet à penser. Comment permettre à chaque philosophe de se donner à ses activités (manger et penser) sans jamais être bloqué ou privé ?

Nous comprenons vite que, si l'algorithme consiste à prendre la fourchette de gauche puis la fourchette de droite et que deux philosophes côté à côté décident de manger en même temps, l'un va être bloqué. Ce blocage peut assez vite se propager. Le cas extrême étant que les 5 philosophes aient leur fourchette gauche dans la main et attendent indéfiniment que celle de droite se libère.

Les phénomènes d'inter-blocage concernent toutes les ressources. Ce peut être le cas de la mémoire si des processus demandent plus de mémoire en cours d'exécution, ou cherchent à accéder à un fichier en cours d'utilisation par un autre.

Définition 11 (inter-blocage). Un système d'exploitation doit garantir que les fonctions dont il a la charge (tel l'ordonnancement de processus) ne provoque pas d'inter-blocage.

4.4 Gestion des droits

Dans un système multi-utilisateurs, les utilisateurs fictifs (`login`) sont :

- soit l'avatar d'un humain du mode réel,
- soit le représentant d'une fonction (`mail` : l'administrateur des courriels, `www-data` : l'administrateur de pages web... et `root` : l'administrateur suprême).

Les données ont un propriétaire : un utilisateur fictif responsable de leur gestion.

Les données peuvent être :

- communes (lisibles et modifiables par tous)

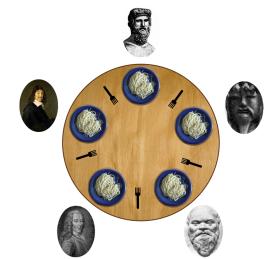


FIGURE 4: Dîner de philosophes

- publiques (lisibles par tous mais non modifiables)
- privées (uniquement accessibles par le propriétaire)

Dans tous les cas, un administrateur suprême a accès à vos données, sur votre machine ou sur des serveurs distants.

Il serait fastidieux de décrire toutes les combinaisons de droits entre tous les utilisateurs. Si le système compte n utilisateurs, il faudrait compter $n \times n$ descriptions de droits par ressource! En conséquence, on définit des profils génériques : les groupes.

À titre d'exemple, Unix fixe la description des droits aux ressources à 3 domaines :

- le propriétaire
- le groupe propriétaire
- les autres

Les droits sont :

- **r** : lecture
- **w** : écriture
- **x** : exécution (pour les fichiers) / traverser (pour les répertoires)
- **s** : *suid* ou *sgid* (pour les fichiers) / nouveaux fichiers au nom du propriétaire de répertoire (pour les répertoires)
- **t** : *Sticky Bit* historiquement l'utilisation fréquente (pour les fichiers) / suppression réservé au propriétaire du fichier (pour les répertoires)

Rappelons que **S** et **T** correspondent aux droits **s** et **t** sans la présence du droit **x**.

Droits

Un système d'exploitation multi-utilisateurs doit fournir un mécanisme de gestion de droits.

4.5 Limitation d'accès

Une erreur de programmation ou une anomalie conjoncturelle peut engendrer une demande incessante de ressources. Par exemple, le programme suivant va provoquer une infinité de création de processus.

1 \$0 &

Listing 3: Le shell de la mort

Le système va passer son temps à créer des processus et à les détruire. C'est une fonction qu'il sait faire rapidement mais qui doit être assurée par le noyau. Il ne pourra donc rien faire d'autre et ne sera pas interruptible.

Responsabilité

N'oubliez jamais que de grandes connaissances impliquent une grande sagesse.

(Pensez à sauvegarder votre travail avant de lancer une telle commande).

Vous constatez qu'un programme de 3 caractères seulement peut paralyser tout votre système d'exploitation. C'est gênant quand c'est un autre qui le lance pour vous nuire. C'est autant gênant quand c'est vous qui le faites par erreur. Vous souhaitez légitimement que le système vous offre la pleine puissance de ses ressources. Dans le même temps, vous souhaitez qu'il soit vigilant et contrôle l'accaparement des ressources (y compris par vous). Il est donc possible de fixer des limites y compris pour soi-même.

La commande *quota* permet de fixer par utilisateur (ou par groupe) des limites d'utilisation de disque (limites molles et dures) ainsi qu'un délai de grâce.

De la même façon, il est possible de limiter d'autres ressources (nombre de processus, temps d'utilisation du processeur, nombre de fichiers ouverts, taille des programmes ...) avec *limits.conf* (voir le manuel).

Limitation

Un système d'exploitation multi-tâches doit fournir un mécanisme permettant d'éviter l'accaparement des ressources par un processus.

5 Ordonnanceur

Il est fréquent en français de confondre le contenu avec le contenant (dire boire un verre au lieu de boire l'eau qu'il contient). Parce que leurs manipulations seront différentes, il faut savoir distinguer les :

- programmes
- processus

- processeurs

Définition 12 (programme). Un programme est une suite d'instructions imaginées pour réaliser un algorithme. Un programme a un nom (`/bin/bash`).

C'est un composant logiciel qui existe indépendamment de l'alimentation de la machine qui l'héberge. Un seul exemplaire d'un programme est nécessaire. Un programme n'a pas une date de début, de fin, ni de durée. Peut-être ne sera-t-il jamais exécuté.

Les programmes existent sous forme : de scripts qui sont interprétés par un interpréteur (`bash`, `python`, ...) qui le traduira en instructions d'un processeur ; ou de binaires (programmes compilés) pour être directement interprétés par le processeur. Il existe des situations hybrides (comme `Java`) où le programme source est traduit en instructions qui seront interprétées par un processeur virtuel (machine virtuelle `Java`) dont les instructions seront elles-mêmes interprétées par un processeur réel.

Définition 13 (processus). Un processus est l'exécution (une réalisation dans le temps) d'un programme. Un processus a un identifiant unique seulement au moment de l'exécution (exemple 1231).

Il peut y avoir simultanément plusieurs processus d'un même programme. Un processus débute à un instant et se termine (de préférence). Il comprend un environnement d'exécution, un état des mémoires, des canaux d'entrées/sorties et les positions d'avancement du programme.

Définition 14 (processeur). Un processeur est un composant matériel qui interprète les instructions d'un programme suivant la progression gérée par le processus. En programmation classique, un processeur n'est normalement jamais désigné ni par l'utilisateur ni par le développeur. Ce peut être le cas en programmation d'algorithmes parallèles pour répartir les lieux d'exécution.

Dans un système multi-tâches, un processeur traitera les processus à tour de rôle donnant l'illusion qu'ils progressent en parallèle. Si un ordinateur possède n processeurs, il ne pourra y avoir au plus que n processus actifs en même temps (les autres seront en sommeil).

Dans une première approximation nous dirons que les coeurs sont comparables à des processeurs. Dans les faits, contrairement aux coeurs, 2 processeurs ne partageront pas nécessairement les mêmes horloges et bus de données.

Définition 15 (ordonnanceur). L'ordonnanceur (en anglais *scheduler*) est

un programme du système d'exploitation (exécuté par un processus), qui contrôle le déroulement des autres processus.

Dans le cas de systèmes d'exploitation à plusieurs processeurs (ou plusieurs coeurs), il peut y avoir physiquement un accès simultané à une même ressource.

6 Les processus Unix

Nous usons généralement de métaphores anthropomorphiques pour décrire les processus : père, fils, vie, mort, testament, ... Nous continuons donc dans ce style en ayant bien conscience que ce monde virtuel est sans pitié, sans féminin et où les processus passent leur temps à tuer leur progéniture.

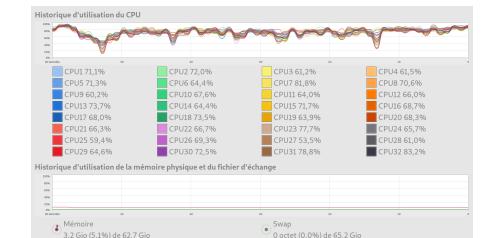


FIGURE 5: Exemple multi-cœurs

Définition 16 (PID). Les processus ont un identifiant, un entier strictement positif : le PID.

Au démarrage du système d'exploitation (après allumage de l'ordinateur), il y a un premier processus "**init**" de PID 1. Autrefois, fils de "**swap**" aujourd'hui fils de lui-même. La métaphore anthropomorphique a ses limites. En effet, il faudrait considérer que le premier être humain (i.e. Homo sapiens en Afrique il y a 300 000 ans) aurait été son propre fils.

Définition 17 (fork). Les processus sont créés par clonage. L'activité se sépare en 2 (comme une fourche en forme de Y) sous l'action d'une fonction du noyau : "**fork**". Seul le PID diffère entre le père et le fils.

En théorie, le processus est recopié en mémoire. En pratique, une optimisation évite cette opération, car la plupart du temps le programme associé au processus est remplacé immédiatement après la création du clone (car le père à une tâche spécifique à donner à son fils). Du fait qu'il n'y a pas recopie, mais référence sur le même programme, le noyau traite cette fonction de façon quasi instantané. Cela explique le "shell de la mort" vu ci-dessus.

Un processus père peut avoir autant de processus fils que nécessaire (dans la limite de la capacité du système).

Une tâche (ou opération au sens commun) peut nécessiter la réalisation de

sous-tâches (ou opérations intermédiaires). Il faut attendre la fin des sous-tâches (ou sous opérations) pour reprendre la tâche initiale. De la même façon, un processus créera des processus fils pour la réalisation de son but. Il attendra la fin des sous-but (processus fils) pour poursuivre son but principal.

Dans ce contexte, nous comprenons la logique des ancêtres qui vivent plus longtemps que leurs descendances. Dans le monde réel, l'évolution des générations futures n'est possible que par la mort des générations précédentes.

Lorsqu'une tâche est abandonnée, ses sous-tâches n'ont plus de raison d'être. De même, lorsqu'un processus est tué, tous les processus fils reçoivent le signal de se terminer.

Un processus se termine parce qu'il :

- a terminé son programme
- met un terme à son exécution
- a reçu un signal lui demandant de se "suicider"

La demande de terminaison d'un processus s'effectue par l'envoi d'un signal qui précise la motivation de cette fin. En voici quelques exemples :

- QUIT (^C) : arrêt de l'utilisateur
- HUP : rupture de connexion
- TERM : kill standard
- STOP (^Z) : fige le processus et CONT réactive le processus

Définition 18 (kill). En shell, c'est la commande "kill" qui permet d'envoyer un signal de demande de fin de processus. Exemple : `kill -TERM $(pidof emacs)`

Un programme peut prévoir un détournement du signal pour soit l'ignorer, soit réaliser une action. C'est utile pour demander confirmation à l'utilisateur avant de terminer réellement un processus. Ce moyen est contourné pour communiquer entre processus.

Définition 19 (trap). En shell, c'est la commande "trap" qui permet d'ignorer ou de détourner un envoi de signal. Exemple : `trap "rm /tmp/res-$@ 0 1 2 3 15"`.

Cependant, il existe un motif qui ne peut être détourné : le signal "KILL" ou "9".

Il est également possible de détacher un processus de sa lignée pour ne pas recevoir une demande de terminaison à la mort de son père. Il suffit d'ignorer le signal "HUP". À la mort de son père, le processus est adopté par le père de

tous les processus : "init". Il n'est plus attaché à un terminal et devient un démon ("deamond").

L'utilisation des signaux est en réalité généralisée comme un moyen de communication entre processus. Un processus peut ainsi signaler la présence de données ou tout autre information de synchronisation simple.

Il existe un autre moyen de communication spécifique au moment de la mort d'un processus pour délivrer une sorte de "testament" vers son créateur. Évidemment, sous Unix seuls les fils délivrent un testament vers leurs ancêtres. L'information est limitée à la valeur d'un entier. Cette valeur est en général le motif de terminaison du fils, 0 indique que tout s'est bien passé, sinon la valeur correspond à un code d'erreur. Attention : la convention est inverse de celle du langage C.

- en C : 0 = false / autres valeurs = vrai
- en shell : 0 = ok / positif = nombre de résultats / négatif = code d'erreur

Définition 20 (exit). En shell, c'est la commande "exit" qui permet de se suicider et d'envoyer un testament. Exemple : `exit -1`.

Un père n'est pas toujours dans un état compatible avec la lecture d'un testament. Un processus fils mort restera dans la table des processus tant que son père n'aura pas lu son testament. Il est à la fois vivant (dans la table des processus) et mort (exécution terminée). On l'appelle dans ce cas : "défunt" (en anglais "defunct") ou "zombie". Comme "init" est toujours à l'écoute des testaments de ses descendants (même s'il ne les lit pas), un démon (rattaché à "init") ne peut jamais devenir zombie.

Définition 21 (wait). En shell, c'est la commande "wait" qui permet d'attendre la mort d'un fils et retourne son testament. Exemple : `wait $!`

Le mécanisme de "fork" permet de créer une multitude de processus. Le seul problème est qu'ils seront tous identiques (des clones). Il est nécessaire de recourir à un autre moyen pour les différencier.

Définition 22 (exec). Le programme d'un processus peut être changé au cours de l'exécution par substitution avec un autre programme. L'activité précédente est alors arrêtée, toutes les données sont oubliées (sauf les variables d'environnement et les canaux de communication) sous l'action d'une fonction du noyau : "exec". La commande échoue s'il n'existe pas de programme exécutable indiqué en paramètre (ou que certaines limites sont atteintes).

Si l'appel réussit, l'instruction suivant la commande exec ne sera jamais exécutée.

Il ne faut pas confondre, dans des langages comme *C* ou *python*, les fonctions “exec” et “system”. La fonction “system” n'est que la composition des fonctions “fork”, “exec” et “wait”. Elle permet de cloner un processus, de différencier le père et le fils pour qu'ils réalisent des actions différentes :

- le fils va substituer son programme avec le nom fourni en argument.
- le père va attendre la fin du fils.

L'échange d'informations entre processus peut donc se faire par héritage des variables d'environnement (au moment du clôture et avant une substitution de programme), envoi de signaux ou envoi de testament.

Il est possible de communiquer les entrées/sorties standards. Au moment de la création de ses fils, un processus peut modifier après clôture les entrées/- sorties (avant une éventuelle substitution). En *shell*, voici quelques exemples de modification :

- `commande > file` redirige la sortie standard vers un fichier
- `commande 2> file` redirige la sortie d'erreur vers un fichier
- `commande < file` puise l'entrée depuis un fichier
- `commande1 | commande2` alimente l'entrée d'une commande avec la sortie de la première

Il existe d'autres redirections possibles que nous ne développerons pas ici.

Il est également possible d'utiliser des fichiers en mode file (en anglais *fifo* pour *first in first out*). Ils sont aussi appelés pipe nommé (pipe pour tuyau, en anglais *named pipe*).

Définition 23 (tuyau ou en anglais *pipe*). Un “*pipe*” est un canal de communication en mode file (en anglais *fifo*)

Pour créer une file on peut utiliser indifféremment l'une ou l'autre des commandes suivantes :

- `mkfifo maFile`
- `mknod maFile p`

Nous utiliserons enfin des sémaphores nommés avec les fonctions suivantes :

- initialisation :
`sem_t *sem_open(const char *name, int oflag, mode_t mode, unsigned int value);`
- suppression :
`int sem_unlink(const char *name);`

- Puis-je ? :
`int sem_post(sem_t *sem);`
- Vas-y ! :
`int sem_post(sem_t *sem);`

Pour vérifier l'existence d'un sémaphore nommé, il faut regarder le contenu du répertoire : `/dev/shm/`.

Nous donnons les programmes utilisables en shell en annexe à la fin de ce chapitre.

En plus de ce que nous venons de décrire (fichiers, signaux, pipes nommés), Le noyau offre d'autres moyens de communication entre processus : les IPC pour *Inter-Process Communication*. L'avantage de passer par le noyau est que le système a la parfaite connaissance de la dépendance entre processus. Il sait donc quel processus est en attente de l'évènement d'un autre et pourra le laisser dormir dans la table des processus jusqu'à ce que les conditions soient réunies pour l'activité. Les processus ne sont donc pas en attente active.

7 Communication inter processus

La commande “*ipcs*” fournit la liste des ressources exclusivement réservées pour la communication entre processus :

- sémaphore
- mémoire partagée
- file de message

Toutes ces ressources sont sous le contrôle de droits d'accès des fichiers. Cela signifie qu'elles ont un propriétaire et un groupe. Il est possible de préciser les droits de lecture et d'écriture de chacun. Pour pouvoir être désignées sans ambiguïté entre des processus qui n'ont pas de lien de parenté (via “*fork*”), elles sont associées à des clefs.

7.1 Les sémaphores

Les sémaphores sont indispensables pour éviter des attentes actives.

- `semget` crée ou récupère un bloc de sémaphore.
- `semop` applique les opérations P et V sur des sémaphores.
- `semctl` réalise des opérations de contrôle (statistiques, droits).

7.2 Les mémoires partagées

La mémoire partagée est sous le contrôle d'un utilisateur et peut être superposée à celle utilisée dans un processus pour des variables. Ainsi, la modification d'un entier dans un processus écrit en C peut être instantanément liée à une autre variable entière d'un processus écrit en python.

- `shmget` crée ou récupère une mémoire partagée
- `shmat` attache une mémoire partagée à un processus
- `shmdt` détache une mémoire partagée d'un processus
- `shmctl` réalise des opérations de contrôle.

7.3 Les files de messages

Les files de messages fonctionnent en fifo (premier arrivé, premier servi). Il y a la garantie que les messages n'interfèrent pas. Cela peut être utilisé pour envoyer des données au fil de l'eau (provenant de capteur) ou des commandes (d'un système graphique).

- `msgget` crée ou récupère une file de messages.
- `msgsnd` ajoute un message dans la file.
- `msgrcv` retire un message de la file.
- `msgctl` réalise des opérations de contrôle.

8 L'attente passive

Un ordinateur (et heureusement pour sa consommation et son ventilateur) tourne la plupart du temps au ralenti. Un programme passe beaucoup de temps à attendre : la saisie au clavier, le clic d'une sourie, une information provenant du réseau, la disponibilité d'un bloc d'information de la mémoire de masse...

Nous l'aborderont dans la section consacrée aux tâches, une première approche (inefficace) consiste à faire une boucle d'attente active. C'est à dire que l'on vérifie qu'une information est disponible, sinon on s'endort un laps de temps (plus ou moins courte) pour retenter sa chance plus tard.

Une autre approche est de dédier un processus pas événement bloquant (par exemple resté en attente de la lecture d'une ligne. Mais là encore il faudra la collecte d'informations simultanées.

8.1 La fonction "select"

Il vaudrait mieux indiquer au système la ou les ressources que nous attendons et demander à être réveillé au moment où cela arrive. D'autant le système, qui gère toutes les ressources, et celui qui va nous les présenter. Sous Unix, cette fonction se nomme "select".

```
1 #include <sys/select.h>
int select (int nfds, fd_set *readfds, fd_set *writefds,
            fd_set *exceptfds, struct timeval *timeout);
```

Listing 4: Prototype de la fonction select

Puisque toute ressource est un fichier sous Unix, on fournit à la fonction des tableaux de pointeur sur des descripteurs de fichier ouvert en lecture ou écriture (terminal, sourie, disque...). Il y a en paramètre trois tableaux de ressources pour lesquelles on attend : une lecture, une écriture ou un évènement exceptionnel. "nfds" indique le nombre maximum de descripteurs à considérer dans les 3 tableaux. "timeout" est un délai de garde.

En Java par exemple, on contourne ce problème d'attente active en écrivant une tâche par événement attendu (l'acceptation une connexion réseau réseau, la lecture d'un fichier sont bloquante). Et on laisse l'ordonnanceur de tâche le soin d'activer la tâche lorsque la ressource est débloquée. En ce qui concerne la gestion des événements graphiques (clavier, sourie), une tâche spécifique s'en charge. Cette tâche est automatiquement lancée dès l'affichage d'une première fenêtre.

8.2 La commande "inotifywait"

Il peut arriver de vouloir attendre en shell la modification d'un fichier (donc d'un répertoire également). La commande "inotifywait" s'avère précieuse. L'exemple suivant permet d'afficher la création et la suppression d'un fichier dans un répertoire.

```
1#!/bin/bash
# sudo apt-get install inotify-tools
#
# termA: inotify.sh
# termB: touch /tmp/queue/toto ; rm /tmp/queue/toto
#
# TMP_REP=/tmp
# TMP_QUEUE="${TMP_REP}/queue"
#
# mkdir -p "${TMP_QUEUE}"
#
# inotifywait -e create,delete --format '%f' --quiet "${TMP_QUEUE}" --monitor |
#   while read filename; do
#     [ -f "${TMP_QUEUE}/${filename}" ] &&
#       echo "new file: ${filename}" ||
#       echo "file deleted: ${filename}"
```

Listing 5: Attente passive en shell sur un répertoire

9 Les commandes shell

Voici une liste restreinte de commandes shell dans le contexte de ce chapitre que nous pourrons être amenés à utiliser en TD et TP. Il est vivement conseillé de compléter ces explications avec la commande "man" ou de les lancer avec l'argument "--help".

9.1 Liste des processus

Commandes concernant les processus :

- "ps" permet de voir l'ensemble de ses propres processus ou de ceux des autres. Les liens de parenté sont décrits et permettent de recréer la généalogie des processus. Il est également précisé le temps passé depuis sa création en distinguant celui passé dans le noyau. On peut également voir les raisons d'une mise en sommeil (attente d'entrée/sortie).
- "pstree" Une version simplifiée, mais plus visuelle.

9.2 Les signaux

Voici un rappel des commandes citées dans ce chapitre :

- "kill" envoie un signal.
- "nohup" neutralise la réception du signal "hangup" pour réaliser un "démon".
- "trap" détourne ou ignore la réception de signaux
- "exit" écrit un testament pour son père avant un suicide.
- "wait" attend le testament d'un fils.

9.3 L'arrière-plan

Il est possible de jouer à faire passer les processus d'un terminal en avant-plan ou en arrière-plan. Les commandes n'ont de sens que pour les shells en mode interactif :

- & lance une commande en arrière-plan
- "jobs" liste tous les processus en arrière-plan ou interrompu.

- ^Z pause du processus en avant plan (correspond au signal "STOP")
- "bg" mise en arrière-plan (correspond au signal "CONT")
- "fg" mise en avant plan (donc avec perte du prompt) (correspond au signal "CONT" et wait)

9.4 Les variables

Pour visualiser les différents processus et leurs testaments, nous utiliserons les variables suivantes :

- \$\$ (PID) du shell initial
- \$BASHPID PID réel de la partie de shell concerné
- \$PPID PID du père
- \$! PID du dernier fils lancé en arrière-plan
- \$? dernier testament reçu à la dernière terminaison d'un fils lancé en arrière-plan
- \$(cmd arg) substitut le texte par le résultat de la commande

9.5 La syntaxe

Voici enfin quelques rappels sur la syntaxe pour regrouper des commandes ou les ordonner séquentiellement.

- {} regroupe des commandes shell en tentant de ne pas créer de fils.
- () crée explicitement un shell fils pour exécuter une liste de commandes
- ; séparateur séquentiel de commandes (toutes les commandes sont censées être exécutées et une erreur interrompt le shell)
- || permet de lancer séquentiellement des commandes. La première est exécutée systématiquement. Les suivantes ne seront lancées que si une précédente échoue.
- && permet également de lancer séquentiellement des commandes. Toutes les commandes sont exécutées tant qu'elles réussissent.

10 Annexe : code source

Il n'existe pas de commande shell permettant la manipulation directe de sémaphores. En revanche, il existe une librairie C qui donne accès aux fonctions du noyau. Voici quelques programmes qui combinent ce manque.

Les programmes sont à compiler de la façon suivante :

```
1 #!/bin/bash
# (c) F. Merciol
# Plus d'informations sur http://r305.merciol.fr
for file in mkNamedSem rmNamedSem namedSemP namedSemV
do
    gcc ${file}.c -lpthread -o ${file}
done
```

Listing 6: Compilation des outils sémaphores

10.1 mkNamedSem

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#include <fcntl.h>
#include <sys/stat.h>
#include <semaphore.h>

void usage (char *prog) {
    fprintf (stderr, "Usage: %s semName initialValue\n", prog);
    exit (-1);
}

int main (int argc, char** argv, char** envp) {

    if (argc != 3)
        usage (argv [0]);

    int value = atoi (argv [2]);
    sem_t *sem;
    if ((sem = sem_open (argv[1], O_CREAT, S_IRUSR|S_IWUSR, value)) == SEM_FAILED) {
        perror ("can't create semaphore");
        exit (1);
    }
}
```

Listing 7: Crée un sémaphore nommé

10.2 rmNamedSem

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#include <semaphore.h>
```

```
10 void usage (char *prog) {
    fprintf (stderr, "Usage: %s semName\n", prog);
    exit (-1);
}

15 int main (int argc, char** argv, char** envp) {

    if (argc != 2)
        usage (argv [0]);

    20 if (sem_unlink (argv[1])) {
        perror ("can't remove semaphore");
        exit(1);
    }
}
```

Listing 8: Supprime un sémaphore nommé

10.3 namedSemP

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#include <semaphore.h>

void usage (char *prog) {
    fprintf (stderr, "Usage: %s semName\n", prog);
    exit (-1);
}

int main (int argc, char** argv, char** envp) {

    if (argc != 2)
        usage (argv [0]);

    20 sem_t *sem;
    if ((sem = sem_open (argv[1], 0)) == SEM_FAILED) {
        perror ("can't find semaphore");
        exit (1);
    }

    25 if (sem_wait (sem)) {
        perror ("can't P");
        exit (1);
    }
}
```

Listing 9: Puis-je ? sur un sémaphore nommé

10.4 namedSemV

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
#include <stdio.h>
#include <errno.h>
#include <stdlib.h>

#include <semaphore.h>
```

```
10 void usage (char *prog) {
11     fprintf (stderr, "Usage: %s semName\n", prog);
12     exit (-1);
13 }
14
15 int main (int argc, char** argv, char** envp) {
16     if (argc != 2)
17         usage (argv [0]);
18
19     sem_t *sem;
20     if ((sem = sem_open (argv[1], 0)) == SEM_FAILED) {
21         perror ("can't find semaphore");
22         exit (1);
23     }
24
25     if (sem_post (sem)) {
26         perror ("can't V");
27         exit (1);
28     }
29 }
```

Listing 10: Vas-y ! sur un sémaphore nommé

Troisième partie

Introduction à la programmation réseau

11 Brève histoire de l'Internet

- 1961 MIT : communication par paquets
- 1962 MIT : application possible entre ordinateurs
- 1962 DARPA : création d'ARPANET
- 1970 : TCP/IP UDP/IP
- 1971-1978 : Cyclades en France
- NFS généralise ARPANET en l'Internet.
- ~1990 BBS
- 2000 l'Internet devient le support commercial

D'où vient le réseau mondial que nous connaissons aujourd'hui ?

Le MIT rédigea en 1961 un texte théorique sur la communication par paquets, puis en 1962 sur des applications possibles entre ordinateurs.

Le département de la défense étasunienne (au DARPA) s'empara du projet en 1962. Le réseau militaire portera le nom d'ARPANET (adresse IP 10.0.0.0).

Dans les années 1970, TCP et UDP sont définis.

A noter le réseau expérimental français Cyclades (dirigé par Louis Pouzin) se crée à la suite de ARPANET. Les échanges entre Cyclades et ARPANET furent nombreux. «Les travaux de Pouzin nous ont beaucoup apporté, explique Vinton chef de ARPANET. Nous avons utilisé son système de contrôle de flux pour le protocole TCP/IP. C'était motivant de parler avec lui.»

La NSF (Fondation Nationale pour la Science étasunienne) généralise l'accès à ARPANET, qui devient l'Internet (l'interconnexion des réseaux : le réseau des réseaux).

Dans les années 1990, il y eut un projet alternatif : les BBS (Bulletin Board System) qui consistait en une multitude de serveurs, avec un ou des modems offrant les services d'échanges de messages et de stockage et d'échange de fichiers. Le modem (modulateur-démodulateur) est un dispositif traduisant les éléments d'informations (bits) en sons, pour être transmis par des lignes téléphoniques. Dans les années 2000, l'Internet supplanta les BBS.

Le réseau est mondial. En France, il existe plusieurs opérateurs privés et publics. Ils sont interconnectés par le "SFINX". Tous les échanges inter-opérateurs y transitent en France. L'université utilise son propre réseau : Rénater (**RÉ**seau **N**ational de télécommunications pour la **T**echnologie, l'**E**nseignement et la **R**esearch, voir la figure 7). Rénater n'est pas l'Internet. C'est un des réseaux de l'Inter connexion des Réseaux.

En 2013, l'Internet représente 10% de la consommation électrique mondiale, soit 1 000 Tw (l'équivalent de 90 réacteurs nucléaires de 1 300 Mw). L'électricité n'est qu'un vecteur de transport de l'énergie (pas une source). La source d'énergie de l'Internet est nucléaire en France. Dans le reste du monde, l'Internet utilise principalement du charbon et des gaz de schiste. Les calories dégagées par les centres de données participent évidemment également au réchauffement climatique.

En 2013, les GAFAM (Google, Apple, Facebook, Amazon) représentent un chiffre d'affaires de 200 milliards de \$ (figure 6 source <http://meta-media.fr/2014/02/07/si-les-gafa-etaient-des-etats-infographie.html>). Ils réduisent le commerce de proximité. Et par un modèle économique fondé sur la publicité, ils augmentent le prix des denrées, ce qui touche les pays pauvres et nous-mêmes. Les principales bénéficiaires sont des compagnies étasuniennes.

C'est enfin une source d'informations sur la vie privée des citoyens, mise en place sous forme ludique (moteur de recherche, exhibition de la vie privée, commerce en ligne). L'Internet représente un pouvoir politique.

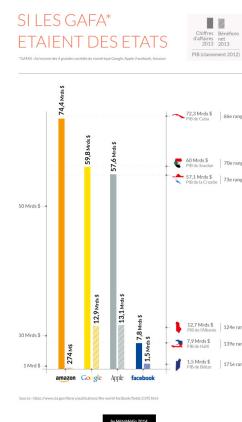
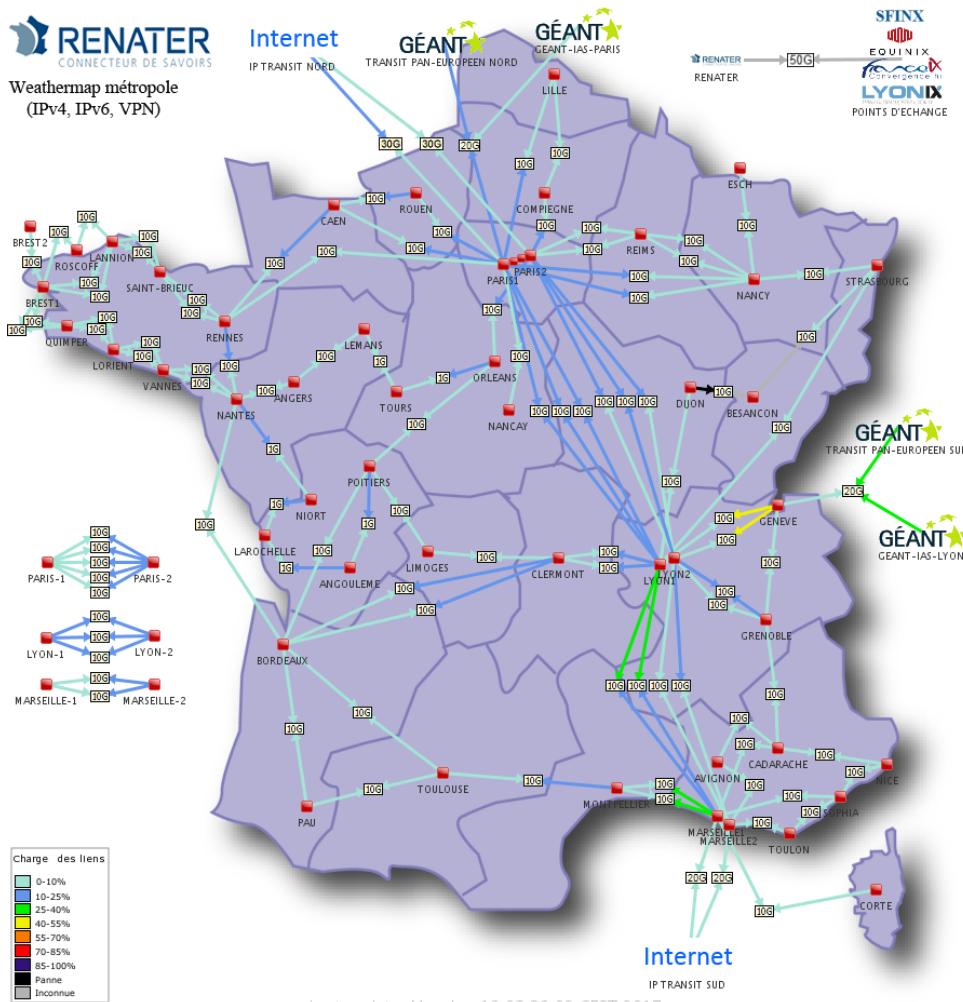


FIGURE 6: Chiffre d'affaires des GAFAM (source Mediapart)



L'Internet est un standard (il n'est pas régi par des accords entre nations). Dans le même temps, les nations définissent des normes. L'ISO normalise les communications informatiques. La table 4 représente une correspondance entre le standard de l'Internet et la norme ISO.

ISO	couche ISO	couche IP	structure	Protocoles IP
7	Application	Application	données	SMTP, POP3, telnet, FTP
6	Présentation			
5	Session			
4	Transport	Transport	segments	TCP, UDP
3	Réseau	Internet	paquets	IP
2	Liaison	Ethernet	trames (frames)	
1	Physique		bits	

TABLE 4: Correspondance ISO / l'Internet

12 Ethernet

La couche de liaison fonctionne sur un bus d'échanges de trames d'information. Sur le même brin d'un bus, toutes les machines envoient leurs trames de façon asynchrone.

Elles écoutent en même temps qu'elles envoient. Lorsqu'il y a collision, elles arrêtent l'émission et reprennent plus tard.

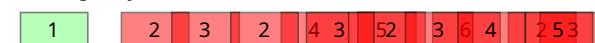
Pour limiter la ré-émission en même temps, elles attendent un temps aléatoire avant de recommencer.

En théorie... cela ne fonctionne pas.

Echange synchronie (ex systolique)



Echange asynchrone (ex Etheret)



→ t

FIGURE 8: Exemple de congestion de trames

La probabilité de collisions augmente au fur et à mesure que des collisions apparaissent. Le phénomène devient exponentiel.

Habituellement, les choses "tombent en panne" (on ne sait pas pourquoi la panne survient). Ici, les choses "tombent en marche" (on ne sait pas pourquoi la marche survient).

En fait, lorsqu'il y a saturation, un signal particulier est envoyé pour réinitialiser les tirages aléatoires. L'envoi systématique et automatique est une solution à ce problème.

ARP (Address Resolution Protocol) est un protocole permettant de découvrir les adresses des autres machines sur le brin (commande `arp -a`).

L'identification se fait par l'adresse MAC (Media Access Control). Elle est composée de 48 bits (6 octets affichés en chiffres hexadécimaux séparés par “:”):

- 1 bit : individuel ou groupe
- 1 bit : universel ou local
- 22 bits : identifiant du constructeur
- 24 bits : adresse unique pour un constructeur donné

Les trames Ethernet sont constituées de l'adresse MAC du destinataire, de l'adresse MAC de l'émetteur, de 2 octets de protocole (par exemple IP) et des données.

13 IP

Ethernet permet la connexion de proche en proche. L'Internet permet la connexion de bout en bout (du réseau).

La désignation se fait par une adresse IP composée de 4 octets (en IP V4) écrits sous forme décimale.

- Le premier bit à 0 désigne la classe
 - classe A 128 réseaux de 16 777 214 machines
 - classe B 16 384 réseaux de 65 534 machines
 - classe C 2 097 152 réseaux de 254 machines
 - classe D multidiffusion
 - classe E réservée pour usage futur
- Les suivants, l'identifiant de réseau
- Les derniers, l'identifiant des machines dans le réseau

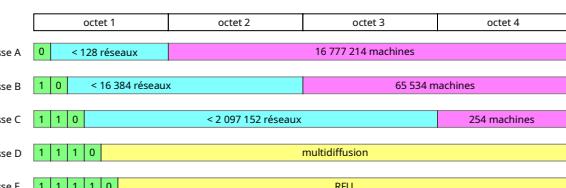


FIGURE 9: Classes IPV4

Le masque réseau permet de séparer les 2 éléments d'une adresse et de réaliser du sous-adressage. Il s'agit d'une suite de 1 à gauche (points forts) pour le réseau et de 0 à droite (points faibles) pour les machines.

Il existe 2 adresses de machine spécifiques :

- tous les bits à zéro : désigne le réseau
- tous les bits à un : désigne la diffusion sur le sous-réseau.

On notera par exemple le masque générique d'une classe C 255.255.255.0. Ou plus rapidement un “/” suivi du nombre de 1. “/24” en classe C. Il est possible de découper une classe en augmentant le nombre de bits réseau. On perd alors deux adresses (réseau et diffusion).

Certains réseaux sont réservés pour chaque classe pour un usage privé. Tout le monde peut les utiliser à condition de ne pas sortir sur l'Internet (de toute façon ces adresses sont bloquées) :

- classe A : 10/8 de 10.0.0.0 à 10.255.255.255
le réseau de défense étasunienne (donc non routé sur l'Internet)
- classe B : 172.16/12 de 172.16.0.0 à 172.31.255.255
- classe C : 192.168.0.0/16 de 192.168.0.0 à 192.168.255.255

14 TCP/IP UDP/IP

Il n'est pas possible de monopoliser le bus pour l'envoi d'une trame. Les trames sont limitées en taille. Pour l'envoi de données volumineuses les données contenues dans une trame sont découpées en segments.

Les segments sont numérotés et peuvent être ré-émis en cas de corruption. Ils sont automatiquement réordonnés à leur arrivée. Dans les 2 cas, il y a échange entre les deux machines tout au long de l'échange des segments.

Il existe 2 sortes de segments.

- TCP pour l'envoi en mode dit connecté.
- UDP pour l'envoi en mode dit déconnecté ou datagramme.

La différence entre les deux modes provient du fait qu'en TCP les ressources restent réservées pour des échanges au niveau applicatif. Dans le cas d'UDP, il faudra établir une nouvelle connexion à chaque envoi.

15 Les Sockets

Pour échanger des informations entre applications, celles-ci utilisent des "prises" (en anglais *sockets*) avec soit TCP soit UDP. C'est l'interface logiciel de communication. Bien que l'on utilise le même terme de *socket* et les mêmes trames IP, les manipulations sont différentes.

Un socket possède un identifiant sur 2 octets : le "port". Chaque port correspond à un usage d'application. L'usage se nomme service et la correspondance entre port et service se trouve dans le fichier `/etc/services`

Port inférieur à 1024

Les numéros inférieurs à 1024 ne sont utilisables que par l'administrateur. Cela garantit que le service n'est pas rendu par un utilisateur quelconque. Par exemple, il est préférable que les messages ne puissent pas être reçus par un invité sur une machine multi-utilisateur.

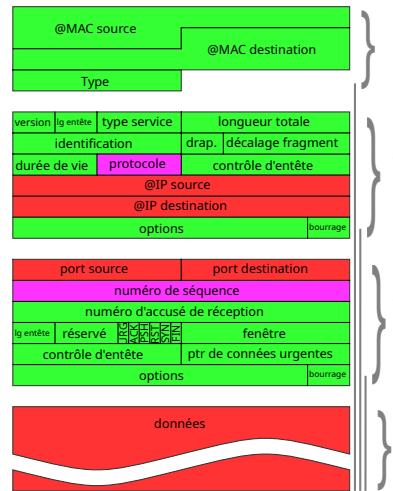


FIGURE 10: Trames IP

Lorsque l'utilisateur n'a pas de préférence de numéro, il utilise la valeur 0, un port disponible (au-dessus de 1024) lui est fourni.

L'application serveur web (par exemple apache) a un port (en général 80). L'application navigateur (par exemple firefox) en a un également (il changera à chaque connexion).

15.1 TCP/IP

En mode connecté (TCP), l'application crée un socket particulier : le serveur de sockets (une sorte de standard d'appel).

Au moment de la connexion, le standard va créer une nouvelle liaison pour pouvoir libérer la ligne. Elle aura le même numéro de port côté serveur. Les liaisons sont différencierées par le fait qu'elles associent un numéro de port client et un numéro de port serveur.

Il y a 3 fonctions pour établir une connexion client/serveur :

- "bind" une seule fois côté serveur pour lier l'application à un port.
- "accept" à chaque nouveau client pour établir la liaison.
- "connect" côté client pour établir la connexion vers le serveur.

En Java c'est la classe `Socket` qui représente un socket. La méthode `connect` est invoquée en fournissant l'adresse IP et le numéro de port sur la machine serveur.

La classe `ServeurSocket` représente le "standard" d'appel. Le `bind` est réalisé à la construction en fournissant le numéro de port (l'adresse IP est trouvée automatiquement localement). La méthode `accept` est bloquante. Elle sera libérée par une connexion d'un client. Elle retourne un `socket` déjà connecté.

Voici un exemple de code Java pour un serveur qui attend un client unique. Il attend une ligne puis reste connecté pour lui envoyer l'entrée standard.

```
1 int port = Integer.parseInt (args [0]);
  ServerSocket serverSocket = new ServerSocket (port);
  System.out.println ("Start server on port "+port);
```

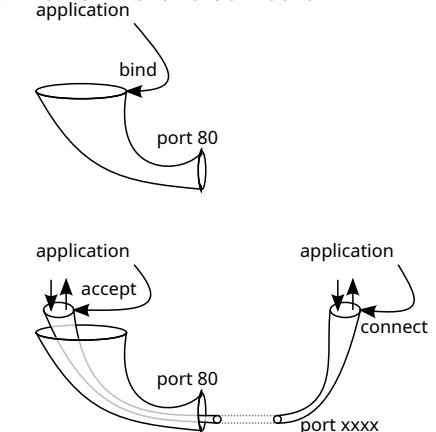


FIGURE 11: TCP/IP

```

5   for (;;) {
6     Socket call = serverSocket.accept ();
7     System.out.print ("Client coming from " +
8       call.getRemoteSocketAddress ());
9     BufferedReader in =
10      new BufferedReader
11        (new InputStreamReader (call.getInputStream ()));
12     PrintStream out =
13       new PrintStream (call.getOutputStream (), true);
14     BufferedReader sysin =
15       new BufferedReader (new InputStreamReader (System.in));
16     for (;;) {
17       String line1 = in.readLine ();
18       if (line1 == null)
19         break;
20       System.out.println (line1);
21       System.out.flush ();
22       String line2 = sysin.readLine ();
23       out.println (line2);
24       out.flush ();
25     }
26     System.out.println ("Socket closed");
27   }

```

Listing 11: Code Java d'un serveur de sockets

La ligne 2 crée le serveur. La ligne 6 reçoit une connexion. Les lignes 9 et 12 récupèrent les entrées/sorties du socket.

Voici un exemple de code Java pour un client de socket. Il se connecte et envoie l'argument du programme. Ensuite, il reste connecté tant que le serveur lui envoie du texte.

```

1   Socket clientSocket =
2     new Socket (args [0], Integer.parseInt (args [1]));
3     PrintStream out =
4       new PrintStream (clientSocket.getOutputStream (), true);
5       out.println (args[2]);
6       out.flush ();
7       clientSocket.shutdownOutput ();
8       BufferedReader in =
9         new BufferedReader
10        (new InputStreamReader (clientSocket.getInputStream ()));
11       for (;;) {
12         String line = in.readLine ();
13         if (line == null)
14           break;
15       System.out.println (line);
16       System.out.flush ();
17       out.close ();
18     }

```

Listing 12: Code Java d'un client de socket

La ligne 1 établit une connexion. Les lignes 3 et 8 récupèrent les entrées/-sorties du socket.

15.2 UDP/IP

En mode non-connecté (UDP), l'application crée un socket particulier : le serveur de datagramme (message de données).

C'est le même type de socket pour les serveurs et les clients.

L'adresse IP et le port de l'émetteur des messages se trouvent dans les messages reçus eux-mêmes.

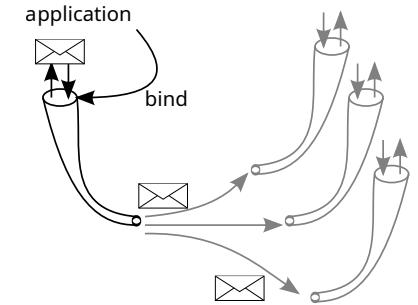


FIGURE 12: UDP/IP

Voici un exemple de code Java pour la réception d'un datagramme. Il récupère le port dans l'argument de la commande. Il faut créer un tampon mémoire capable de recevoir le paquet de données en entier. Les données reçues sont sous forme d'octets. Il faut prévoir un mécanisme d'interprétation.

```

1   int port = Integer.parseInt (args [0]);
2   DatagramSocket socket = new DatagramSocket (port);
3
4   byte[] buf = new byte[256];
5   DatagramPacket packet = new DatagramPacket (buf, buf.length);
6   socket.receive (packet);
7
8   String received = new String (packet.getData (), 0, packet.getLength ());
9   String srcIP = packet.getAddress ().getHostAddress ();

```

Listing 13: Code Java de réception d'un datagramme

La ligne 2 crée le socket d'échanges de datagramme. La ligne 5 crée un datagramme d'accueil. La ligne 6 puise dans le datagramme reçu depuis la création. La ligne 9 affiche le message reçu et son expéditeur.

Voici un exemple de code Java pour l'envoi d'un datagramme. Il récupère les arguments, crée le datagramme et l'envoie.

```

1   InetAddress address = InetAddress.getByName (args[0]);
2   int port = Integer.parseInt (args [1]);
3   byte[] buf = args[2].getBytes ();
4   DatagramPacket packet =
5     new DatagramPacket (buf, buf.length, address, port);
6
7   DatagramSocket socket = new DatagramSocket ();
8   socket.send(packet);

```

Listing 14: Code Java de l'envoi d'un datagramme

La ligne 4 crée le datagramme. La ligne 8 l'envoie.

16 Protocoles IP

La plupart des protocoles de l'Internet datent des origines. À l'époque, on a choisi d'échanger les informations en mode texte. C'est donc des codes ASCII qui transiting par le réseau. Cela facilite la compréhension et la mise au point. L'origine de l'Internet étant étasunienne, les mots sont en anglais.

Un outil particulièrement simple a été développé. Il s'agit de la commande `telnet host port`. Elle permet d'établir une connexion avec un serveur donné en argument, de connecter l'entrée du socket au clavier et la sortie sur l'écran.

Les protocoles comme : SMTP (envoi de message), POP3 (consultation de message), FTP (transfert de fichiers), HTTP (navigation web)... échangent en clair sur le réseau.

Les outils graphiques que vous utilisez aujourd'hui, ne sont que des interfaces pour masquer ces échanges.

Vous pouvez sans difficulté vous connecter et discuter directement avec l'un de ces serveurs.

17 Annexe : code source

Voici le code complet des extraits donnés dans ce chapitre.

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
package network;

5 import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;
import java.net.ServerSocket;
import java.net.Socket;

10 public class RecvCmd {

    static public void main (String[] args) {
        if (args.length != 1) {
            System.err.println ("Usage: RecvCmd port");
            System.exit (1);
        }
        try {
            15     int port = Integer.parseInt (args [0]);
            ServerSocket serverSocket = new ServerSocket (port);
            System.err.println ("Start server on port "+port);

            for (;;) {
                20     Socket call = serverSocket.accept ();
                System.err.println ("Client comming from "+
                    call.getRemoteSocketAddress ());
                BufferedReader in =
                    new BufferedReader
                    (new InputStreamReader (call.getInputStream ()));
30 }
```

```
35         PrintStream out =
            new PrintStream (call.getOutputStream (), true);
        BufferedReader sysin =
            new BufferedReader (new InputStreamReader (System.in));
        for (;;) {
            String line1 = in.readLine ();
            if (line1 == null)
                break;
            40             System.out.println (line1);
            System.out.flush ();
            String line2 = sysin.readLine ();
            out.println (line2);
            out.flush ();
        }
        45             System.err.println ("Socket closed");
    } catch (Exception e) {
        e.printStackTrace ();
}
50             System.err.println ("Server closed");
        System.err.flush ();
    }
}
```

Listing 15: Réception de datagramme en Java

```
1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
package network;

5 import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.io.IOException;
import java.io.PrintStream;
import java.net.Socket;

10 public class SendCmd {

    static public void main (String[] args) {
        if (args.length != 3) {
            15             System.err.println ("Usage: SendCmd host port request");
            System.exit (1);
        }
        try {
            20             Socket clientSocket =
                new Socket (args [0], Integer.parseInt (args [1]));
            PrintStream out =
                new PrintStream (clientSocket.getOutputStream (), true);
            out.println (args [2]);
            out.flush ();
            clientSocket.shutdownOutput ();
            BufferedReader in =
                new BufferedReader
                (new InputStreamReader (clientSocket.getInputStream ()));
                25             for (;;) {
                    String line = in.readLine ();
                    if (line == null)
                        break;
                    System.out.println (line);
                    System.out.flush ();
                    out.close ();
                }
        } catch (IOException e) {
            30             System.err.println ("Socket closed");
        }
}
```

```

40     System.err.flush ();
    } catch (Exception e) {
        e.printStackTrace ();
    }
}

```

Listing 16: Envoi de datagramme en Java

```

1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
package network;

5 import java.io.IOException;
import java.net.DatagramPacket;
import java.net DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;

10 public class RecvData {
    static public void main (String[] args) {
        if (args.length != 1) {
            System.out.println ("Usage: RecvData port");
            System.exit (1);
        }
        try {
20            int port = Integer.parseInt (args [0]);
            DatagramSocket socket = new DatagramSocket (port);

            byte[] buf = new byte[256];
            DatagramPacket packet = new DatagramPacket (buf, buf.length);
            socket.receive (packet);

            String received = new String (packet.getData (), 0, packet.getLength ());
            String srcIP = packet.getAddress ().getHostAddress ();
            int scrPort = packet.getPort ();
            System.out.println (srcIP+":"+scrPort+" send : "+received);
        } catch (SocketException e) {
            e.printStackTrace ();
        } catch (IOException e) {
            e.printStackTrace ();
        }
    }
}

```

Listing 17: Réception de datagramme en Java

```

1 // (c) F. Merciol
// Plus d'informations sur http://r305.merciol.fr
package network;

5 import java.io.IOException;
import java.net.DatagramPacket;
import java.net DatagramSocket;
import java.net.InetAddress;
import java.net.SocketException;
import java.net.UnknownHostException;

10 public class SendData {
    static public void main (String[] args) {
        if (args.length != 3) {
            System.out.println ("Usage: SendData host port data");
            System.exit (1);
        }
}

```

```

20     }

try {
    InetAddress address = InetAddress.getByName (args [0]);
    int port = Integer.parseInt (args [1]);
    byte[] buf = args [2].getBytes ();
    DatagramPacket packet =
        new DatagramPacket (buf, buf.length, address, port);

    DatagramSocket socket = new DatagramSocket ();
    socket.send (packet);
} catch (UnknownHostException e) {
    e.printStackTrace ();
} catch (SocketException e) {
    e.printStackTrace ();
} catch (IOException e) {
    e.printStackTrace ();
}
}

```

Listing 18: Envoi de datagramme en Java

*La théorie c'est quand on sait tout et que rien ne fonctionne.
 La pratique c'est quand tout fonctionne et que personne ne sait pourquoi.
 Ici, nous avons réuni théorie et pratique : rien ne fonctionne ...
 et personne ne sait pourquoi!*

Albert Einstein

Quatrième partie

Les tâches

Nous avons déjà abordé la question des activités avec la notion de processus. Dans ce chapitre, il s'agit d'un niveau plus fin d'activité. On parle alors : de tâches, de processus légers ou allégés, de fil d'activité, d'exécution, d'instructions ou bien encore d'unité de traitement ou d'exécution.

Définition 24 (tâche). Une tâche (en anglais *thread*) est une suite séquentielle d'instructions qui s'écoulent dans le même temps (donc indépendamment) que d'autres traitements au sein d'un même espace d'adressage.

En première approximation les tâches sont des "processus" dans un processus. Alors, à quoi servent-elles et quelles sont les différences avec des processus ?

18 Comparaison entre tâches et processus

Il est fréquent d'avoir besoin de réagir à des événements désynchronisés. Habituellement, nous envisageons des déroulements en séquence pour un programme. Mais, l'environnement extérieur impose d'autres rythmes. Par exemple, en cas de sollicitations du réseau, un serveur n'a pas connaissance a priori du nombre ni de la fréquence des requêtes des clients. De même, lors de développement d'une interface graphique, les interactions avec l'utilisateur (clavier, souris) et avec le gestionnaire de fenêtres (recouvrement, transparence avec d'autres applications) s'entrelacent dans une logique différente de celle du programme.

On trouvera donc fréquemment des tâches pour gérer les :

- demandes de connexions
- requêtes de clients connectés
- événements graphiques
- modifications du modèle de données

Une différence importante entre tâche et processus provient du mode de communication entre eux.

- Entre processus, les espaces d'adressage sont clairement isolés. Le système est l'intermédiaire incontournable et contrôle les interactions. Nous les avons cités au chapitre "Partage des ressources" : IPC, pipe, signaux, fichiers ...
- Entre tâches d'une même application, il y a partage du même espace d'adressage. Le système ne contrôle pas l'utilisation simultanée des échanges d'informations entre tâches. L'accès est plus rapide, mais c'est au développeur d'en gérer le contrôle et la cohérence. Nous aborderons donc dans ce chapitre les questions de synchronisation.

18.1 Mécanismes historiques

Les mécanismes permettant de gérer des tâches en parallèle sont les mêmes que pour les processus. Ils sont apparus en même temps que les systèmes d'exploitation.

- En premier le plus simple, la notion de co-routines. Il s'agit en fait d'un traitement dont on pense qu'il se termine en un temps court. Au lieu de faire une boucle sans fin sur le traitement d'un évènement graphique, on ne traite que le premier et on retourne. L'application passera son temps à donner la main à un ensemble de co-routines. Le fait de ne pas bloquer induit une attente active.
- Vient ensuite la notion de préemption. On déclare auprès du système une alarme déclenchée par un chronomètre. Il faut alors sauvegarder le contexte d'exécution interrompu et vérifier que les données sont dans un état stable compatible à un changement de traitement. L'attente active est réduite par des périodes de sommeil. Mais dans le cas où il ne se passe rien, l'application est tout de même réveillée.
- En dernier, on a souhaité bénéficier de l'ordonnanceur de processus du système et de sa performance. Ce phénomène s'accroît avec l'arrivée des multiprocesseurs et multicoeurs. Lorsque vous êtes seuls sur une machine octo-processeurs, que vous avez conçu un programme en Java avec des traitements parallèles, il serait dommage de ne pas jouir pleinement de cette architecture.

À titre d'exemple, l'ordonnanceur d'une Machine Virtuelle Java (JVM) peut déclarer au système Linux, les tâches engendrées par un processus (tâches elles-mêmes décrites dans un programme Java ou Scala).

Il existe plusieurs niveaux d'intégration des tâches pour le développeur.

- Extérieur au système. Le développeur écrit des co-routines. Mais cette solution est inefficace du fait de l'attente active.

- Au niveau d'appels au système d'exploitation. Le développeur écrit des fonctions et doit les enregistrer auprès du système. Il devra mettre en place le contrôle nécessaire et prendre les décisions au niveau le plus bas.
- Au niveau de librairie. Dans ce cas, des composants logiciels ont été réalisés pour les cas les plus fréquents. Le développeur doit alors spécialiser des mécanismes génériques.
- Au niveau du langage. Là, le traitement parallèle est intégré au développement. C'est le cas de Java ou de Scala, deux langages qui produisent du "bytecode" interprété par une même Machine Virtuelle Java (JVM). Des mots clefs et une syntaxe facilitent la rédaction et permettent la manipulation de concepts au plus haut niveau (exceptions, piles séparées d'exécution, synchronisations, priorités, ramasse-miettes ...).

Sous Linux, vous pouvez voir les tâches avec l'option `-T` de la commande `ps`. L'option fait apparaître une colonne supplémentaire "SPID". Lorsque que le processus n'a qu'une tâche le "PID" et "SPID" sont identiques. On peut considérer que le "SPID" est un entier unique obtenu à partir d'un compteur et que le "PID" prend la valeur du "SPID" de la tâche principale.

Pour une approche plus concrète, la suite du chapitre sera illustrée par des exemples en Java. Pour voir les processus et tâches Java vous pouvez lancer la commande Unix : `jps`.

19 Création de tâches

Il n'existe qu'une seule façon de créer une tâche en Java :

- créer un objet de la classe `Thread`. Cet objet sera le point de contrôle de la tâche.
- invoquer sur cet objet la méthode `start`. Cette méthode lancera une activité sur la méthode `run` du même objet.

La séquence d'instructions que doit réaliser l'activité est rassemblée dans la méthode de nom `run`. Toute tâche possède une méthode `run`. L'invocation de cette méthode ne crée aucune tâche.

Méthode run

Vous ne devez jamais invoquer directement la méthode `run` d'un objet réalisant une tâche. Seule, la méthode `start` fera cette invocation dans une tâche séparée.

La méthode `start` n'est bien évidemment pas bloquante. L'appelant n'attend donc pas le résultat du nouveau traitement engagé. Aussi la méthode `run` ne retourne aucune valeur (type `void`).

De façon symétrique, pour que la méthode `start` soit générique, la méthode `run` ne prend également aucun paramètre formel. Il est possible d'utiliser des attributs dans la classe définissant la méthode `run` pour initialiser le traitement réalisé dans la tâche.

Il existe plusieurs façons d'attacher une méthode `run` à une tâche :

- par héritage de la classe `Thread`
- par réalisation de l'interface `Runnable`

Nous allons maintenant détailler ces différentes façons.

19.1 La classe Thread

La façon la plus directe de créer une tâche est de spécialiser la classe "Thread".

Dans le listing 19, la classe `SimpleThread` est une nouvelle classe d'activités. Son comportement est spécialisé à la ligne 4. Dans l'exemple, il n'y a qu'une instruction. Bien évidemment l'ensemble du corps de la méthode `run` est exécuté dans la tâche quelle que soit sa longueur et sa durée.

Pour tester la tâche, nous avons placé dans une méthode `main` les 2 étapes :

- création de l'objet tâche en ligne 9 et
- lancement de l'activité proprement dite en ligne 12.

```

1 public class SimpleThread extends Thread {
2
3     public void run () {
4         System.err.println ("Here is "+getName ());
5     }
6
7     static public void main (String[] arg) {
8         System.err.println ("new");
9         Thread thread = new SimpleThread ();
10
11        System.err.println ("start");
12        thread.start ();
13    }
14 }
```

Listing 19: Héritage de Thread

19.2 L'interface Runnable

Java n'offre pas d'héritage multiple. Si un objet est déjà issu d'une classe provenant d'un héritage et que l'on souhaite qu'il ait une activité, il est toujours possible de lui faire réaliser (implanter) l'interface "Runnable".

Dans le listing 23, la classe *SimpleRunnable* n'est pas une activité. Elle n'est donc pas contrainte par un héritage spécifique. En revanche, l'implantation de l'interface *Runnable* lui impose de décrire une méthode *run* qui sera éventuellement invoquée par une tâche.

Pour tester notre tâche, nous avons placé dans une méthode *main* les 3 étapes :

- création de l'objet de notre application qui pourra dérouler une tâche en ligne 9,
- création d'une tâche générique qui activera le comportement de notre objet en ligne 10 et
- lancement de l'activité proprement dite sur la tâche qui déroulera le comportement de notre objet en ligne 13.

```
1 public class SimpleRunnable implements Runnable {  
2  
3     public void run () {  
4         System.err.println ("Here is "+Thread.currentThread ().getName ());  
5     }  
6  
7     static public void main (String[] arg) {  
8         System.err.println ("new");  
9         Runnable runnable = new SimpleRunnable ();  
10        Thread thread = new Thread (runnable);  
11  
12        System.err.println ("start");  
13        thread.start ();  
14    }  
15}
```

Listing 20: Implantation de Runnable

Nous pouvions lancer directement une tâche générique anonyme avec l'instruction : `(new Thread (runnable)).thread.start ()` ;

19.3 Les tâches anonymes

Une solution intermédiaire consiste à ne pas hériter de la classe "Thread" et ne pas créer une nouvelle classe dans l'arbre de nos développements. On peut utiliser une classe anonyme spécialisée, qui hérite de la classe "Thread" mais redéfinit à la volée la méthode "run".

Dans le listing 21, commençons par décrire la ligne 5. Les parenthèses vides indiquent un appel au constructeur d'une tâche. Nous aurions pu fournir des paramètres au constructeur (pour donner un nom à la tâche). La particularité de cette syntaxe est que juste après la liste des paramètres réels on trouve une paire d'accolades qui permet de spécialiser l'objet créé. Ce qui est mis dans les accolades correspond à ce que l'on aurait mis dans une classe dérivée : attributs et méthodes. La classe anonyme ne contiendra qu'un seul objet.

```
1 public class AnonymousThread {  
2     static public void main (String[] arg) {  
3         System.err.println ("new & start");  
4  
5         (new Thread () {  
6             public void run () {  
7                 System.err.println ("Here is "+getName ());  
8             }  
9         }).start ();  
10    }  
11}
```

Listing 21: Tâche en classe anonyme

L'avantage de cette syntaxe est de pouvoir disposer d'un nombre indéfini de tâches, par exemple en reprenant dans une boucle cette instruction de création anonyme. Cette technique est particulièrement adaptée à la définition à la volée de rétroaction (en anglais *callback*) sur des éléments d'interface graphique (boutons, champs de saisie ...).

19.4 Autres propriétés

La classe "Thread" dispose d'autres propriétés :

- un constructeur et une méthode *setName* qui permettent de fixer un nom symbolique de la tâche.
- une méthode *getName* qui permet de connaître le nom de la tâche.
- une méthode de classe *sleep* qui permet de demander à s'endormir pendant un temps approximatif.
- une méthode de classe *currentThread* permettant de connaître la tâche qui exécute cette invocation.

Il arrive souvent que l'on souhaite ne voir qu'un seul exemplaire de la tâche que l'on vient de lancer. Par exemple, lorsque l'utilisateur clique frénétiquement sur sa souris, il faut savoir réagir en temps réel.

Définition 25 (temps réel). Pour qu'un développement soit "temps réel", il faut qu'il ignore les actions devenues obsolètes. L'application est "temps réel",

si elle est capable d'abandonner ce qui ne peut plus être réalisé dans les temps.

Bien entendu, la rapidité du traitement doit être le plus proche possible de ce qui est nécessaire dans le cas général. En revanche, il ne sert à rien de savoir traiter une image d'un film en moins de 1/20^e de seconde lorsque la fréquence de défilement passe de 20 à 60 images par seconde. Il est plus utile de savoir abandonner le traitement une fois sur 3 pour conserver la fluidité du mouvement, autrement dit un rendu "en temps réel".

L'abandon d'un travail est donc utile. Combien de fois un utilisateur re-clic sur le démarrage d'un traitement de texte pensant que sa première demande n'avait pas été satisfait. Il se retrouve ensuite avec deux fenêtres ouvertes.

Voici un exemple de code (listing 22) pour ne lancer qu'une seule fois une même tâche. Le principe est de mémoriser la dernière tâche lancée. Toutes les tâches ne poursuivent leurs traitements que si elles sont bien la dernière lancée.

```
1 public class SingleThread {  
2  
3     private Thread lastThread;  
4     public void stopSingle () {  
5         lastThread = null;  
6     }  
7     public void startSingle () {  
8         new Thread () {  
9             public void run () {  
10                 for (lastThread = currentThread ();  
11                     lastThread == currentThread ();  
12                 ) {  
13                     System.err.println ("Here is "+getName());  
14                     pause (1);  
15                 }  
16             }  
17         }).start ();  
18     }  
19  
20     static public void pause (int seconds) {  
21         try {  
22             Thread.sleep (seconds*1000);  
23         } catch (InterruptedException e) {}  
24     }  
25     static public void main (String[] arg) {  
26         SingleThread activeObjet = new SingleThread ();  
27         for (int i = 0; i < 4; i++) {  
28             activeObjet.startSingle ();  
29             pause (1);  
30         }  
31         activeObjet.stopSingle ();  
32     }  
33 }
```

Listing 22: Limitation à une tâche

Vous percevez tout l'intérêt de la tâche anonyme. En effet, *SingleThread* ne

peut hériter de *Thread* sans impliquer l'unicité de la tâche. Car, un nouvel appel à *start* n'aurait aucun effet. Nous souhaitons ici recommencer tout le processus en initialisant la tâche avec les informations du moment (mises à jour par l'utilisateur) et en demandant l'arrêt des anciennes versions.

Enfin, le recours à une implantation de l'interface *Runnable* aurait rendu complexe la phase de lancement d'une nouvelle tâche, en initialisant des identifiants uniques pour comprendre de quelle filiation provenait la tâche pour l'associer à un objet de contrôle.

Notons pour finir, que la ligne 32 non seulement termine les tâches créées, mais arrête du même coup le programme Java.

Terminaison Java

Le processus d'une application Java se termine lorsque toutes ses tâches sont terminées.

La tâche principale d'une application Java invoque la méthode *main* de la classe principale. Après l'exécution de la méthode *main*, elle attend que toutes les tâches secondaires se terminent. Lorsque l'application demande la création d'une fenêtre, une tâche de gestion des événements graphiques est créée. Comme on ne peut déterminer quand l'utilisateur cessera de jouer avec une application, cette tâche ne se termine jamais et l'application Java de même. Il faut prévoir une action spécifique (bouton *quitter*) pour mettre un terme au programme.

20 Synchronisation

La création de tâche va de pair avec le contrôle de sections critiques. Nous avons vu au chapitre "partage des ressources" comment gérer de telles sections à l'aide de sémaphores. Java a fait un choix plus fin que celui des sémaphores pour synchroniser les méthodes ou l'accès à des attributs d'un objet. Il utilise le concept de "moniteur". L'équivalence entre moniteur et sémaphore a été démontrée. Il ne s'agit donc que d'un choix de facilité d'usage et d'un niveau d'intégration avec le langage.

Commençons par parler de synchronisation Java hors moniteur.

20.1 L'instruction join

Il y a un premier niveau de synchronisation en Java, celui de la vie des tâches : au démarrage et à la fin.

- La synchronisation de démarrage est la conséquence de la méthode "start".
- La synchronisation de terminaison se fait explicitement avec l'appel de la méthode "join".

```
1 public class Join {  
2     static public void pause (int seconds) {  
3         try {  
4             Thread.sleep (seconds*1000);  
5         } catch (InterruptedException e) {  
6         }  
7     }  
8     static public void main (String[] arg) {  
9         Thread thread = new Thread () {  
10             public void run () {  
11                 System.err.println ("Here  
12                     is "+getName ());  
13                 pause (2);  
14             }  
15         };  
16         thread.start ();  
17         pause (1);  
18         try {  
19             thread.join ();  
20         } catch (InterruptedException e) {  
21     }  
22 }
```

Listing 23: Synchronisation sur la vie d'une tâche

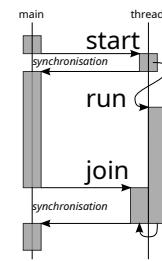


FIGURE 13:
Synchronisation

20.2 Moniteur Java

Définition 26 (moniteur). Un moniteur est un dispositif permettant le contrôle de l'exécution de méthodes particulières d'un objet, en vue de résoudre des problèmes posés par des sections critiques. Les méthodes ne doivent pas s'exécuter simultanément. Pour autant, le dispositif prévoit d'interrompre l'une des méthodes en donnant la main à une autre, puis de revenir sous certaines conditions à la première.

Dans une première approximation, nous dirons que les moniteurs permettent la déclaration de méthodes en exclusions mutuelles. Nous verrons ensuite que cela peut être affiné.

Définition 27 (synchronized). Le mot clef "synchronized" permet de déclarer des traitements en exclusion mutuelle à l'exception de la tâche qui entre en section critique.

Lorsqu'une synchronisation est contrôlée par un moniteur la tâche qui prend la main a un usage exclusif sur l'objet. Cela a deux conséquences :

- Les autres tâches se trouvent bloquées au moment d'invoquer une exécution synchronisée sur le même objet.
- La tâche ayant la main dans le moniteur peut invoquer librement d'autres exécutions synchronisées sur le même objet. En d'autres termes les méthodes synchronisées sont ré-entrantes (elles peuvent être récursives).

C'est en particulier indispensable pour un mécanisme lecteurs/écrivains. Imaginons un objet synchronisé sur lequel le nous avons développé 2 méthodes :

- *insert* en exclusion mutuelle En écriture, elle ne doit s'exécuter avec aucune autre.
- *dicoSearch* partageable En lecture, elle est utilisable par plusieurs lecteurs simultanément.

Nous souhaitons bien un mécanisme lecteurs/écrivains. Malheureusement, nous souhaitons réutiliser la méthode *dicoSearch* pour savoir où réaliser l'insertion. C'est exactement ce que permettent les moniteurs en autorisant une tâche déjà autorisée à poursuivre ses traitements sur l'objet. Le contrôle lecteurs/écrivains étant différent pour chaque objet (plusieurs objets ayant plusieurs lecteurs en même temps que d'autres objets ont un écrivain).

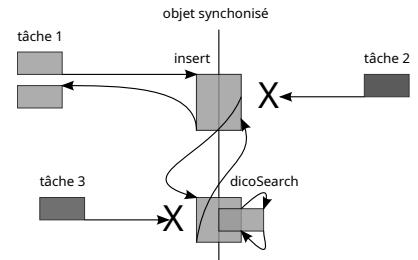


FIGURE 14: Ré-entrée

20.3 Le qualificatif synchronized

Il y a deux syntaxes différentes : objet complet ou liste de méthode.

Au niveau de l'objet complet, la synchronisation permet de demander au moniteur de l'objet un usage exclusif. Certains parlent de synchronisation par bloc, mais c'est bien le moniteur d'**un** objet qui verrouille l'exécution et pas

un bloc. Il n'y a aucune obligation à ce que les méthodes de l'objet soient synchronisées. Si le moniteur est libre, la tâche prend l'exclusivité. Si le moniteur est déjà pris par une autre tâche (présence d'une autre synchronisation d'objets ou de méthodes) le moniteur place la tâche en attente. Elle sera réveillée dès que le moniteur sera libéré.

```
1 synchronized (this) {  
    synchronized (unAutreObjet) {  
        // suite d'instruction en exclusion mutuelle  
        // aucune autre tâche ne pourra utiliser this et unAutreObjet  
    }  
}
```

Listing 24: Synchronisation d'objets

Au niveau d'une méthode, la synchronisation permet de demander au moniteur un usage exclusif pour l'objet sur lequel la méthode est invoquée. L'exclusion est donc mutuelle entre toutes les méthodes déclarées *synchronized* dans la classe de l'objet. Si le moniteur est libre, la tâche prend l'exclusivité. Si le moniteur est déjà pris par une autre tâche (synchronisation d'objets ou de méthodes) la tâche est en attente. Elle sera réveillée dès que le moniteur sera libéré.

```
1 public class UneClasse {  
    public void methodeNonSynchronise1 () { /* */ }  
    public void methodeNonSynchronise2 () { /* */ }  
    public synchronized void methodeEnExclusionMutuelle1 () { /* */ }  
    public synchronized void methodeEnExclusionMutuelle2 () { /* */ }  
}
```

Listing 25: Synchronisation de méthodes

Il existe un 3^e cas, la synchronisation des méthodes de classe. Elle revient à la synchronisation en considérant la classe comme un objet. Dans ce cas, le moniteur n'est pas attaché à un exemplaire de la classe mais à la classe elle-même.

Le mot clef *synchronized* seul ne permet que l'exclusion mutuelle, or nous avons vu en TD du chapitre "Partage des ressources" qu'il existe bien d'autres configurations de sections critiques. C'est là qu'intervient le dialogue avec le moniteur d'un objet pour un réglage plus fin.

20.4 Les instructions *wait*, *notify* et *notifyAll*

Lorsque qu'une tâche déroule en exclusivité une méthode d'un objet, elle a le loisir de vérifier les conditions particulières d'utilisation de la ressource. Le calcul peut révéler que son traitement est prématué. La tâche peut alors demander au moniteur de mettre en sommeil en section critique.

Le moniteur fait donc la distinction entre :

- **La** tâche active qui a autorisation d'exécution sur un objet
- Les tâches en sommeil qui seront réveillées à l'endroit où elles se sont endormies, à la condition de se retrouver seule tâche synchronisée active sur l'objet.
- Les tâches à évaluer qui font la demande d'entrer en section critique mais qui sont bloquées en attente de terminaison ou de sommeil de la tâche active.

Pour communiquer avec le moniteur d'un objet, il existe 3 méthodes : *wait*, *notify* et *notifyAll*.

Définition 28 (*wait*). La méthode *wait* permet à une tâche déjà entrée en synchronisation de s'endormir pour laisser une autre tâche entrer en section synchronisée. Dans tous les cas, il n'y a qu'une seule tâche active en section synchronisée.

Une tâche endormie en section synchronisée n'est pas réveillée par la terminaison de la tâche active du même moniteur. Il faudra un acte explicite de programmation pour vérifier que les conditions sont réunies pour réveiller cette tâche.

Il est possible d'indiquer un délai de garde à la méthode *wait*. C'est à déconseiller car cela a une fâcheuse tendance à sombrer dans l'attente active. Il faut toujours préférer la production d'un événement de réveil à l'endroit où les conditions changent.

Définition 29 (*notify*). La méthode *notify* permet de signaler au moniteur d'un objet qu'il devra réveiller une tâche en sommeil en section synchronisée. Dans tous les cas, il n'y a qu'une seule tâche active en section synchronisée.

L'instruction *notify* peut être en premier ou en dernier dans une méthode. Dans tous les cas, l'effet du *notify* n'aura lieu que lorsqu'il n'y aura plus de tâche en section synchronisée sur l'objet qui reçoit le *notify*.

Notez que *wait* et *notify* peuvent être invoqués en désignant explicitement un objet (qui n'est donc pas nécessairement *this* de la méthode qui invoque).

Définition 30 (*notifyAll*). La méthode *notifyAll* permet de signaler au moniteur d'un objet qu'il devra réveiller *toutes* les tâches en sommeil en section synchronisée. Dans tous les cas, il n'y a qu'une seule tâche active en section synchronisée. Donc les tâches seront réveillées à tour de rôle.

20.5 Une classe sémaphore avec un moniteur

Puisqu'il y a correspondance entre moniteur et sémaphore, nous donnons le listing 28 qui présente une réalisation d'un sémaphore avec un moniteur.

```

1 public class Semaphore {
2
3     private int value;
4
5     public Semaphore (int value) {
6         if (value < 0)
7             throw new IllegalArgumentException ("Negative value !");
8         this.value = value;
9     }
10
11    public synchronized void p () {
12        if (value <= 0)
13            try {
14                wait ();
15            } catch (InterruptedException e) {
16            }
17        value--;
18    }
19
20    public synchronized void v () {
21        value++;
22
23        notify ();
24    }
25 }
```

Listing 26: Sémaphore avec un moniteur

Notez que l'exclusion mutuelle fournie par le mot clef *synchronized*, sans les méthodes *wait* et *notify*, rend impossible le blocage de la méthode *p* tant que la condition *value > 0* n'est pas remplie.

Pour l'utilisation de *notify* et *notifyAll*, toutes les situations sont à regarder au cas par cas. Le tableau suivant est purement indicatif. Il donne une tendance sur les utilisations les plus probables.

	notify	notifyAll
if - wait	OK	risqué
while - wait	très prudent	prudent

TABLE 5: Comparaison notify/notifyAll

Imaginez les différentes configurations avec la classe *Sémaphore* précédente :

- En commentant la bonne ligne dans la méthode *p*

```

11   public synchronized void p () {
12       /* if || while */
13       // if
14       while
15           (value <= 0)
16           try {
17               wait ();
18           } catch (InterruptedException e) {
19           }
20       value--;
21   }
```

Listing 27: Réglage *wait* de la classe *Semaphore*

- En commentant la bonne ligne dans la méthode *v*

```

20   public synchronized void v () {
21       value++;
22       /* notify || notifyAll */
23       // notify ();
24       notifyAll ();
25   }
```

Listing 28: Réglage *notify* de la classe *Semaphore*

Cinquième partie

La mémoire

La mémoire d'un ordinateur revêt des formes multiples :

- mécanique (parties mobiles d'un disque dur)
- physique (encombrement)
- thermique (chauffe)
- électrique (consomme)
- rémanente (persiste après alimentation)
- usage (écriture ou non)
- capacité (taille stockée)
- rapidité (temps d'accès) ...

Citons quelques exemples : RAM, ROM EEPROM, USB, disque, CD, DVD, vive, virtuelle, segmentée, paginée, partagée, d'échange ...

Nous allons aborder différents aspects de la mémoire, mais ce chapitre sera loin d'en faire le tour.

Malgré tout, Portons une attention particulière sur une mémoire à *tore magnétique* qui pendant 20 ans (de 1955 à 1975) permettait de conserver des données sans consommation électrique (voir https://fr.wikipedia.org/wiki/Tore_magn%C3%A9tique). Comme quoi, il est possible de trouver des solutions si on décide de se donner des contraintes (zéro consommation).

Un carré de 11 cm de côté pouvait contenir 512 octets !



FIGURE 15: RAM

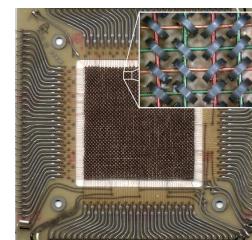


FIGURE 16: Tore magnétique

Profitons de l'occasion pour rappeler qu'à l'origine la mémoire de masse se trouvait sur bande magnétique donc d'un accès séquentiel. Pour lire une information en bout de bande, il fallait dérouler toute la bande. La mémoire vive a permis un accès dit aléatoire (non contraint d'être séquentiel). C'est ce qui a donné leur nom. Elles sont aujourd'hui électroniques.

Pour rappel, une liste de mémoires électroniques

- RAM (*Random Access Memory*) mémoire à accès aléatoire en opposition à mémoire à accès séquentiel.
- SRAM (*Static Random Access Memory*) mémoire chère et volumineuse, mais rapide et économique en énergie. Elle est utilisée dans les caches des microprocesseurs.
- DRAM (*Dynamic Random Access Memory*) s'appuyant sur un pico condensateur et un transistor, cette mémoire nécessite d'être rafraîchie au bout de quelques millisecondes du fait de la fuite prévue du condensateur.
- ROM (*Read-Only Memory*) mémoire morte dont les informations sont figées (peuvent être câblées).
- PROM (*Programmable Read Only Memory*) mémoire programmable une seule fois. Les bits sont stockés par des fusibles. Une surtension les grille. Ils passent à 0 définitivement.
- EPROM (*Erasable Programmable Read Only Memory*) mémoire programmable et effaçable. L'effacement peut se faire par UV (Ultra-Violet), ce qui peut amener à la débrocher pour être effacé.
- EEPROM (*Electrically Erasable Programmable Read Only Memory*) mémoire programmable et effaçable (on dit aussi E²PROM). En clair, on peut inscrire une information (souvent avec une tension électrique plus forte qu'à la lecture) et débrancher ensuite son alimentation. Elle conserve ses données.
- Flash est une variété de mémoire EEPROM rapide et effaçable par secteur complet. On les trouve dans les périphériques devant être personnalisés (BIOS sur une carte mère, adresse MAC sur une carte réseau ...).

21 Caractéristique de mémoires

21.1 Mémoire vive

Définition 31 (mémoire vive). La mémoire est dite vive si elle n'existe que tant qu'elle est alimentée électriquement.

21.2 Mémoire des processus

Historiquement, dans un processus, on distingue 4 espaces mémoires en fonction de leurs usages, les zones de :

- code, contient les instructions de programme
- données, contient les données globales du programme
- pile, contient les données intermédiaires de calcul
- tas, contient des données dont la vie n'est pas liée à la durée d'exécution d'une opération

Définition 32 (codes). La zone de code contient

les instructions d'un programme. Elle est composée des fonctions C, des méthodes Java... Elle comprend des constantes qui se retrouvent dans les instructions assembleur. Sa taille est connue à la compilation. Au moment de l'exécution elle doit être protégée en écriture.

Définition 33 (données). La zone de données contient

les données globales du programme. Elle est composée des variables "externes" aux fonctions C ou des variables statiques dans les fonctions, des attributs statiques de classe Java... Elle comprend des constantes élaborées (initialisées par un calcul au démarrage). Sa taille est connue à la compilation.

Définition 34 (pile). La zone de pile contient

les données intermédiaires de calcul. Elle est composée des variables automatiques (locales) des fonctions, des paramètres formels, des valeurs de retour. Sa taille n'est pas connue a priori, car elle dépendra de l'exécution. En revanche, sa taille croît à chaque appel de fonction ou de méthode et revient au même endroit au retour.

Définition 35 (tas). La zone de tas contient

des données dont la vie n'est pas liée à la durée d'exécution d'une opération. Ces données seront créées par une fonction et continueront d'être accessibles après l'appel. Elle est composée des variables obtenues par la fonction *malloc* en C, l'opérateur *new* en Java ... Pour éviter qu'elle ne croisse indéfiniment, on cherche à mettre des solutions de "ramasse-miettes" pour recycler les cellules mortes (également appelé GC pour "garbage collector" ou "glanage de cellules").

Au moment du lancement d'un processus, nous pouvons influencer sur la taille des mémoires. Par exemple, les arguments fournis à la JVM (Machine Virtuelle Java) fixent ces limites. 1^{er} caractère : "s" pour Stack, "m" pour Memory et 2^d caractère : "s" pour minimum et "x" pour maximum. Voici des exemples :

- -Xss512M taille minimum de 512 Mo pour la pile
- -Xsx1G taille maximum de 1 Go pour la pile
- -Xms6G taille minimum de 6 Go pour le tas
- -Xmx36G taille maximum de 36 Go pour le tas

Vérifiez que vous disposez de la mémoire vive suffisante pour les paramètres.

Les premiers systèmes d'exploitation allouaient le maximum de mémoire exigée par le processus et plaçaient tête-bêche la pile et le tas, en espérant qu'ils ne grandissent pas en même temps.

Aujourd'hui, la mémoire est plus souple (ce sera l'objet des sections suivantes), il n'y a plus de contraintes de placement.

Concernant l'accès en écriture par d'autres processus, on comprend que si un programme pouvait connaître, au moment de l'exécution d'un processus, la plage d'adresse qu'il utilise en mémoire, il pourrait modifier le contenu des variables et voir modifier les instructions initialement prévues. Nous verrons qu'aujourd'hui les adresses sont traduites et que les processus manipulent des adresses virtuelles (voir https://fr.wikipedia.org/wiki/M%C3%A9moire_virtuelle). Ils n'ont pas eux-mêmes accès à cette connaissance.

Définition 36 (mémoire virtuelle). Mise au point dans des années 1960, la mémoire virtuelle se fonde sur la traduction à la volée (dispositif électronique) des adresses utilisées par les processus. Elle peut être segmentée ou paginée.

La mémoire virtuelle permet une diminution de besoin en mémoire réelle, car elle offre :

- le recours à la mémoire d'échange (extension de mémoire)
- le partage de code entre processus (économie de mémoire)
- le partage de données entre processus (possibilité algorithmique).

Par ailleurs, la mémoire virtuelle garantit l'étanchéité entre programme. Ils ont tous des adresses qui commencent par 0 et ne savent pas désigner la mémoire réelle.

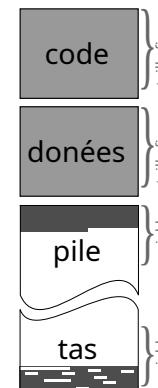


FIGURE 17:
Zone mémoire

21.3 Mémoire partagée

Il est cependant possible de partager des informations entre différents processus. Nous avons déjà abordé cette question dans notre chapitre sur les ressources du point de vue de la communication.

Revenons sur le partage du point de vue de l'espace mémoire.

Au moment de la création ou de la récupération de la mémoire partagée, le processus reçoit un pointeur sur sa zone de tas. En réalité, la mémoire associée réelle est ailleurs.

La mémoire partagée se trouve physiquement dans un espace alloué et géré par le système d'exploitation au moment de la demande sa création par un processus. Les adresses de cette zone dans l'espace d'adressage du processus sont détournées, comme si la mémoire partagée était superposée à son espace habituel.

La position manipulée par le processus et la position réelle de cette zone sont dissociées. Nous le verrons par la suite, qu'il n'y a pas de mémoire continue et donc pas d'espace mémoire perdu.

21.4 Mémoire d'échange

Tous les programmes ne sont pas actifs au même moment en mémoire. Certains sont bloqués en attente de conditions qui n'arriveront peut-être jamais.

L'idée est donc d'être optimiste et de penser que l'on peut servir plus de processus que la mémoire vive réelle ne permet d'en héberger en même temps. Pour cela, quand un processus dort (et que l'on a besoin de place), les données du processus sont recopiées dans une zone d'extension.

Définition 37 (mémoire d'échange). Une mémoire d'échange (en anglais “swap”) est une extension sur un périphérique de mémoire de masse de la mémoire vive. Son utilisation est coûteuse en temps.

Il faut réaliser que les choix seront sûrement différents d'une époque à une autre. C'est parce que la mémoire vive rapidement économiquement coûte cher, que l'on a recours à une mémoire de masse lente pour l'étendre.

Bien entendu, des heuristiques sont développées pour déterminer quelle mémoire est à écarter du processeur.

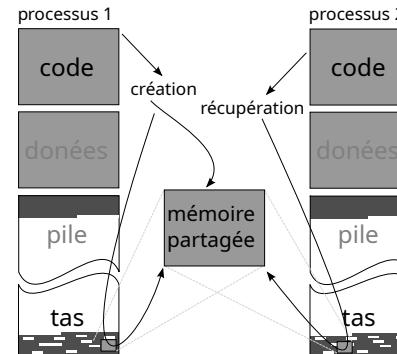


FIGURE 18: Mémoire partagée

Il y a des limites à cet optimisme. Il ne serait pas raisonnable d'accepter l'accueillir des milliers de processus sur un processeur en se disant qu'il y a une mémoire d'échanges. Car, l'inconvénient majeur est le temps d'accès au “swap”. Même si les mémoires du disque et de la RAM étaient équivalentes, il faut dans tous les cas inclure le temps de transfert sur les bus de données. À l'heure, où les mémoires centrales se comptent en gigaoctets, ce temps est loin d'être négligeable.

En revanche, il existe une autre utilisation qui pousse à déclarer une mémoire d'échange sur un système d'exploitation. C'est l'hibernation. Un ordinateur personnel peut être éteint pour consommer moins. Mais le redémarrage est coûteux en temps. On peut vouloir sauvegarder l'état de la mémoire vive pour revenir dans les mêmes conditions de travail après un sommeil. Ce sommeil peut être une simple “mise en veille” et dans ce cas la mémoire vive reste alimentée (donc il persiste une consommation). On peut vouloir une veille prolongée, sans consommation électrique. Dans ce dernier cas, la mémoire vive est enregistrée dans la zone d'échange. On conseille alors de la dimensionner à 1,5 fois la taille de la mémoire vive. Si vous disposez de 64 Go de mémoire vive, il faut prévoir 96 Go de partition d'échange sur votre disque. Sans ce cas, un arrêt complet peut parfois être plus rapide que la mise en veille prolongée.

21.5 Morcellement de la mémoire

Nous nous rappelons le problème d'inter-blocage posé par le dîner des philosophes. Nous allons présenter un cas semblable de “famine” (les processus n'ayant plus de mémoire à “manger” pour pouvoir vivre).

Prenons la situation suivante. Nous disposons d'un serveur avec une mémoire de taille 12 unités. 2 groupes (G1 et G2) de 4 étudiants lancent alternativement des processus sur le serveur. Le premier groupe (G1) a besoin de mémoire de taille 1 et le second (G2) de taille 2.

La moitié des étudiants terminent leur processus. Une partie de mémoire se libère. Sa taille est de $2 * (1 + 2) = 6$.

Un nouveau groupe de 2 étudiants veut lancer un processus (respectivement A et B) qui utiliseront chacun une mémoire de taille 3.

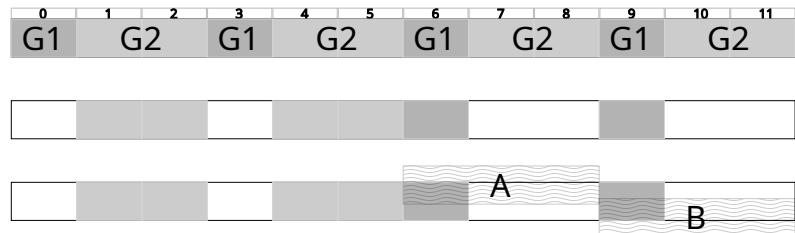


FIGURE 19: Exemple de répartition de mémoire

En théorie, il y a assez de place pour accueillir les processus A et B. En pratique, les espaces libres contigus ne suffisent pas.

Ne peut-on pas découper la mémoire ?

Ce n'est pas si simple. Imaginez que vous avez écrit un document de 10 pages. Vous devez y ajouter un graphique de 10 cm, mais il ne reste plus que 1 cm de libre par page. Allez-vous découper le graphique en 10 morceaux de 1 cm ? Autre problème, les références vers le texte qui se trouvait en page 2, se trouve peut-être maintenant en page 3.

Il en est de même pour les processus. Au moment de la compilation les sauts conditionnels, les sauts de boucle et les appels à des sous-routines ont été figés. Le début du programme est supposé commencer à l'adresse 0. (Nous passons sous silence le problème des données ou du tas).

Et lorsque l'on déplace un programme en mémoire, il faut prendre garde à repérer tous les sauts dans un programme (comme les référence au texte précédent) pour les décaler.

Le problème est encore plus complexe si le processus se présente en morceaux éparpillés.

C'est pourtant ce que l'on envisage de faire avec 2 techniques :

- la segmentation,
- la pagination

22 Mémoire segmentée

Le problème de découpage d'un processus ne se pose pas si on trouve un emplacement mémoire suffisamment grand pour répondre à ses besoins. Seul ceux qui "débordent" nous intéressent.

La première approche est de découper au minimum le processus en commençant par la place la plus adaptée (la plus proche de sa taille et qui pourrait le

contenir). Si le processus ne rentre pas, on coupe ce qui dépasse et on réitère l'opération avec ce qui reste à placer. Si le reste rentre on a terminé.

Pendant ce découpage on maintient une table de correspondance. Voici celle de l'exemple donné :

processus	adresse processus	taille segment	adresse mémoire
1 (G2)	0	2	1
2 (G1)	0	1	6
3 (G2)	0	2	4
4 (G1)	0	1	9
5 (A)	0	2	7
5 (A)	2	1	0
6 (B)	0	2	10
6 (B)	2	1	3

TABLE 6: Table de segmentation

Ce qui nous donne pour le même exemple simplifié une organisation comme celle de la figure suivante.

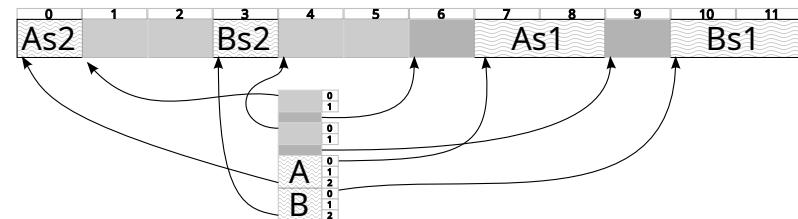


FIGURE 20: Mémoire segmentée

Nous n'avons placé que quelques chiffres dans cet exemple. Cela ne doit pas cacher la complexité de l'algorithme pour retrouver une adresse de processus en mémoire. Voici l'algorithme pour trouver une mémoire à l'adresse A du processus P

- Recherche dans la table de segment S qui possède la plus grande "adresse processus" et qui soit inférieur à A
- La mémoire se trouve à "adresse mémoire" de $S + A - \text{adresse processus}$

En voici une illustration :

processus	adresse processus	taille segment	adresse mémoire
P	0x10543	0x567	0x67321

TABLE 7: Calcul de segment

Pour le calcul de l'adresse 0x12345 on vérifie $0x12345 > 0x10543$ et $0x12345 < (0x10543 + 0x567)$ puis on calcule $0x67321 + 0x12345 - 0x10543$.

Faisons quelques remarques.

- Cet exemple est trivial. En réalité, les adresses de processus sont représentées par des nombres hexadécimaux sur 32 voire 64 bits. La taille des programmes est quelconque. Les trous qu'ils feront en disparaissant ou en allant en zone d'échange, sont également de taille quelconque.
- Les segments qui résultent des trous peuvent avoir n'importe quelle taille eux aussi (impair, non multiple d'une puissance de 2 ...).
- lors de l'utilisation de la mémoire d'échange, nous imaginons sauver un segment complet. Mais au moment de la récupération en mémoire vive, il se peut qu'il n'y ait plus l'espace continu suffisant pour remettre le segment et qu'il faille segmenter à nouveau le processus.

Bref, la mémoire peut devenir rapidement un vrai gruyère. En envisageant le pire, on pourrait imaginer des segments de l'ordre de l'octet. Ce ne serait vraiment pas efficace.

Le système a une charge supplémentaire, il doit lors de l'adressage d'un processus calculer sa translation. Pour cela, il doit en premier déterminer quel segment est concerné. Il doit donc parcourir tous les segments précédents pour déterminer en fonction du cumul des tailles, la position du segment.

processus	adresse processus	adresse mémoire
1 (G2)	0	4
1 (G2)	1	5
2 (G1)	0	6
3 (G2)	0	1
3 (G2)	1	2
4 (G1)	0	9
5 (A)	0	7
5 (A)	1	8
5 (A)	2	0
6 (B)	0	10
6 (B)	1	11
6 (B)	2	3

TABLE 8: Table de pagination

Ce qui nous donne pour le même exemple simplifié une organisation comme celle de la figure suivante.

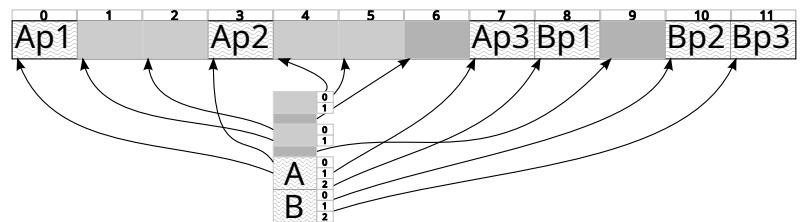


FIGURE 21: Mémoire paginée

De nouveau, nous comprenons qu'il y a un compromis entre rapidité et perte de mémoire. En segmentation, la taille mémoire de l'ensemble des segments est exactement celle demandée par le processus. En pagination, la taille mémoire de l'ensemble des pages est l'arrondi supérieur à la page de celle demandée par le processus.

A priori, il y aurait un peu de perte avec la pagination. Mais ce gaspillage sera vite compensé.

Suivant la base utilisée pour l'adressage (donc en base hexadécimale), on utilisera les 2 derniers chiffres pour le contenu de la page et les précédents pour désigner la page. En voici une illustration :

23 Mémoire paginée

La pagination part du même constat que la segmentation : il faut découper les processus.

La différence majeure est qu'en pagination tous les "bout de mémoire" ont la même taille. On parle alors de "page".

Définition 38 (cadre). Un cadre est une portion de mémoire réelle qui contient une page.

Tous les processus sont systématiquement découpés en page et l'on maintient une table de correspondance. Voici celle de l'exemple donné :

adresse processus	découpage	numéro de page	décalage dans la page
0xXXXXY	0xXXX 0xYY	0xXXX	0xYY
0x12345	0x123 0x45	0x123	0x45

TABLE 9: Calcul de page

Nous remarquons la simplicité de l'algorithme de calcul par rapport à celui de la segmentation.

En réalité, la taille des pages n'est pas nécessairement de 8 bits (1/4 de Ko). Il suffit qu'elle soit une puissance de 2 (des mots de 9, 11 ou 12 bits).

Faisons quelques remarques.

- Il y a des pertes de mémoire pour arrondir à des pages entières (négligeables par processus).
- La recherche de pages libres est rapide (il suffit qu'elles ne soient pas utilisées dans la table).
- Il n'y a pas d'apparition de "trou" qui ne pourrait pas être utilisé comme une page.
- Il n'y a pas de sur-segmentation lors d'un passage en mémoire d'échange.

24 Optimisations

Le passage à la mémoire virtuelle qu'elle soit segmentée ou paginée a apporté des caractéristiques nouvelles.

24.1 Taille mémoire des processus

Par exemple, la mémoire physique peut être plus petite que la mémoire utilisée par le processus.

Imaginons que l'on demande au système la réservation de mémoire dans le tas pour des objets de la taille d'une page. La libération de certains objets va libérer les pages correspondantes en mémoire qui pourront être réutilisées pour d'autres processus. Cependant, il n'est pas utile de compacter la mémoire (il serait même impossible de décaler les adresses des objets déjà utilisés par le processus). La présence de ces trous fait que, l'intervalle entre l'adresse la plus basse du programme et son adresse la plus haute, peut être plus grand que la mémoire réellement utilisée.

Nous pouvons imaginer généraliser le procédé. Lorsqu'un processus demande la réservation d'un objet de plusieurs pages (implicitement initialisé à 0), ce ne sera qu'au moment de l'accès à l'objet que les pages seront créées.

24.2 Calcul d'adresse

Des composants électroniques spécialisés participent au calcul à la volée de la translation d'adresse de la mémoire. Le processeur ne fait en réalité aucun calcul qui retarderait de plusieurs cycles d'horloge la manipulation des données. L'ordonnanceur qui gère l'affectation des processus va déclarer auprès de composants la répartition de la mémoire. La translation des adresses se fera sur le bus de données de manière transparente pour le processeur.

24.3 Partage de codes

Lorsque des pages ne sont accessibles qu'en lecture par plusieurs processus, il devient pratique, avec le mécanisme de pointeurs de la mémoire virtuelle vers la mémoire réelle, de partager des informations communes.

C'est le cas pour l'instruction "exec". Le système d'exploitation tient à jour la liste des programmes déjà lancés et quels processus les utilisent. Il va compter le nombre d'utilisations.

- À la première demande le pointeur est initialisé et le compteur mis à 1.
- À chaque lancement supplémentaire du même programme, le pointeur est réutilisé et le compteur incrémenté.
- À chaque mort de processus, le compteur est décrémenté.
- Arrivé à 0 la mémoire est libérée.

24.4 Chargement à la demande

Il n'est pas toujours nécessaire de charger tout un processus en mémoire. Les parties de code d'un programme sont utilisées différemment tout au long de la vie d'un processus. Certaines ne sont utilisées qu'au début (initialisation), d'autres se voient occupées par période (interaction à l'utilisateur, échange par fichiers...).

Les pointeurs de mémoire virtuelle ne réfèrent pas toujours la mémoire vive, ils peuvent faire référence à une information sur disque (partie de code non encore chargée) ou sur la mémoire d'échange (données libérées pour un autre processus).

Au moment de l'accès, l'électronique va produire une interruption "défaut de page" qui sera interceptée par le système pour résoudre la recherche d'informations selon les stratégies qu'il a implantées.

24.5 Partage de données

Il n'y a pas que le code qui puisse être partagé entre processus. Étonnamment, bien qu'elles soient modifiables et potentiellement uniques, les données aussi peuvent être partagées pendant une partie de leur vie. En effet, l'instruction "fork" doit virtuellement cloner (dupliquer) tout un processus : code, donnée, pile, tas.

Plutôt que de passer du temps à recopier réellement les informations, le système va marquer les pages de code comme étant utilisées une fois de plus. Il va aussi marquer toutes les pages de données comme nécessitant une copie à l'écriture et augmenter le compteur de copies. Supposons qu'un programme déclare une variable "i" puis fork 2 fois. Il y aura sur la page de données un compteur de copies en écriture à 3. Le premier processus qui modifiera sa variable voyant que le compteur n'est pas unique, va recréer une page par copie et mettre le compteur à 1. Il va également décrémenter le compteur de la page d'origine. Le dernier processus qui modifiera sa variable aura un compteur nul et ne fera donc aucune copie.

Vous comprenez que les concepts les plus coûteux en théorie (copie de tout un processus utilisant beaucoup de mémoire) peuvent dans les faits être efficaces.

24.6 Mise en mémoire secondaire

Nous avons vu qu'il est possible de placer des informations en mémoire d'échange (zone de *swap*). C'est particulièrement pertinent si l'information se trouve dans une portion du processus qui ne sera plus utilisée avant longtemps. En réalité, cette réflexion pose un problème générique. Comment choisir l'information qu'il faut faire passer d'une mémoire de grande capacité mais lente, vers une mémoire réduite mais très rapide. C'est la préoccupation de tous les mécanismes de mémoire cache d'un processeur.

La difficulté est bien entendu pour disposer de la page utile à l'instant "t", de l'échanger avec une autre page que l'on n'utilisera plus avant longtemps. Ou autrement dit, de ne conserver que les pages qu'on utilisera le plus souvent par la suite.

Définition 39 (Défaut de page). On appelle "défaut de page" le fait de ne pas disposer de l'information indispensable en mémoire principale et de devoir la charger à partir d'une mémoire secondaire.

Il existe plusieurs algorithmes pour choisir la page à supprimer. Nous allons en comparer 4 avec la séquence suivante de 24 demandes de page de processus pour un système qui dispose de 3 pages de cache :

ordre	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24
proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2

TABLE 10: Séquence de pages de référence pour comparaison

- **optimal** : "on refait le match". On enregistre toutes les demandes de page et on réfléchit en fonction de l'avenir que l'on peut prédire (puisque l'on a tout enregistré). Lorsque l'on doit faire sortir une page, on choisit celle qui ne sera plus utilisée pendant la plus grande période de temps. C'est un cas théorique que l'on ne peut jamais atteindre (la divination n'est pas une démarche très scientifique).

proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2
def.	9	6	7	4				3				5	2					1	4		3			
sup.								4				3	6					7	1	4		3		5

TABLE 11: Algorithme "Optimal"

- **FIFO** : "premier arrivé, premier sorti". Les pages sont placées dans l'ordre d'arrivée et sont sorties dans le même ordre.

proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2
def.	13	6	7	4					3	6		5	2					7	1	5	4		3	2
sup.									4			3	6					6	5	2	7		1	5

TABLE 12: Algorithme "FIFO"

- **LRU** (en anglais *Least Recently Used*) : "la moins récemment utilisée". Si la page est déjà dans le cache, on la remet en tête de file. Sinon on supprime la dernière page de la file et on place en tête la nouvelle page qui manquait. Cet algorithme est le plus souvent utilisé.

proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2
def. 13	6	7	4				3		6		5		2					7	1	5	4		3	2
sup.							6	7		4				6	5	2	7	1	5	4		1	5	

TABLE 13: Algorithme "LRU"

- **aléatoire** : comme son nom l'indique. Si la page est dans le cache on l'utilise. Sinon on tire aléatoirement la page à supprimer pour la remplacer par la nouvelle. Évidemment, conserver des pages aléatoires est plus efficace que de ne pas conserver de pages. Et curieusement, on ne perd pas à tous les coups statistiquement. Cependant, les résultats ne sont pas reproductibles.

proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2			
def. 12	6	7	4				3		2	1	0	2	1	1	2		2	1	0	7	1	5	4		3	2	
aléa	1	2	1	1	0	2	1	2	1	0	2	1	1	0	2	1	0	1	0	2	1	2	2	0			
sup.														3	6	5	2	4	7	1	5	2	4	5	1		

TABLE 14: Algorithme "Aléatoire"

- **LFU** (en anglais *Least Frequently Used*) : "la moins souvent utilisée". Plus complexe, il faut un compteur à chaque page du cache. Si la page est dans le cache, on incrémente son compteur. Sinon on choisit de supprimer le cadre correspondant au compteur le plus faible (il peut y en avoir plusieurs) et on met la page à la place en remettant le compteur à zéro. L'inconvénient est que des pages peuvent être fréquemment utilisées au lancement du processus et que leur compteur soit si élevé qui ne soit plus possible de la remplacer.

proc.	6	7	4	7	4	4	3	3	6	6	3	5	6	2	5	2	2	7	1	5	4	4	3	2
def. 14	6	7	4				3		6		5	6	2	5	2		7	1	5					2
sup.							6	7		4	5	6	2	5		2	7	1						5
compteur	1	1	1	1	1	1	1	2	2	2	3	3	3	3	3	3	3	3	3	3	3	3	4	4
0	1	1	2	2	2	2	2	1	2	2	1	1	1	1	2	1	1	1	1	1	1	1	1	1
0	0	1	1	2	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	4	5	5	5	5
LFU	6	6	6	6	6	6	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3	3
X	7	7	7	7	7	7	6	6	6	5	6	2	5	2	2	7	1	5	5	5	5	5	5	2
X	X	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4	4

TABLE 15: Algorithme "LFU"

24.7 Anomalie de Bélády

Il ne faut pas croire qu'en augmentant la mémoire réelle on accélère systématiquement la résolution du problème de défaut de page virtuelle. Bien sûr, en allouant plus de mémoire virtuelle que vos processus en ont besoin, vous devriez ne pas avoir de défaut de page (peut-on encore parler de cache dans ce cas?).

Cependant, László Bélády (informaticien hongrois) a démontré que des situations particulières contredisaient l'utilisation de l'algorithme FIFO.

En effet, la séquence (3, 2, 1, 0, 3, 2, 4, 3, 2, 1, 0, 4) produit 9 défauts de page avec 3 cadres et 10 défauts de page avec 4 cadres.

Voici une illustration de l'anomalie avec la même représentation que précédemment :

demandé	3	2	1	0	3	2	4	3	2	1	0	4
défaut	9	3	2	1	0	3	2	4		1	0	
supprimée		3	2	1	0				3	2		
FIFO 3	D	X	X	3	2	1	0	3	3	3	2	4
	X	3	2	1	0	3	2	2	2	4	1	1
	P	3	2	1	0	3	2	4	4	4	1	0
défaut	10	3	2	1	0			4	3	2	1	0
supprimée		3	2	1	0			3	2	1	0	4
FIFO 4	D	X	X	X	3	3	3	3	2	1	0	4
	X	3	2	1	2	2	2	1	0	4	3	2
	X	3	2	1	1	1	0	4	3	2	1	0
	P	3	2	1	0	0	0	4	3	2	1	0

TABLE 16: Anomalie de Bélády

Sixième partie

Les entrées-sorties

La communication entre sa partie calculatrice et le monde extérieur est indispensable pour donner un sens à l'utilisation d'un ordinateur. Les premiers modes d'échanges consistaient en des cartes perforées. Elles étaient perforées sur des sortes de machines à écrire. Il y eut ensuite des imprimantes, des écrans (uniquement en texte). Aujourd'hui, nous sommes à l'ère de l'USB et du sans-fil.

Les entrées/sorties peuvent être minimalistes. Pour l'asservissement optique d'un appareil photo, les entrées peuvent se résumer à un interrupteur et un capteur donnant une valeur numérique, et les sorties à l'alimentation d'un moteur pas à pas.

Nous parlerons dans ce chapitre des entrées/sorties d'un système d'exploitation classique. Pour autant, nous ne nous limiterons pas à la communication entre l'ordinateur vu comme une boîte noire et le reste du monde. Mais nous engloberons dans cette notion d'échanges toutes les communications avec un processus. Rappelons celles que nous avons déjà mentionnées précédemment :

- les communications entre processus en commençant par les signaux et en incluant les IPC (sémaphores, files de messages, mémoires partagées)
- les échanges avec un être humain : écran, clavier, souris et tous les dispositifs de pointage (surface tactile, manette de jeux ...)
- les communications via les réseaux filaires ou électromagnétiques
- tous les périphériques en général
- et enfin tous les fichiers en général

La mémoire est sans doute le premier média d'échanges. Pour le grand public, notons que la mémoire vive est passée de 1 Ko en 1981 (avec le ZX81 de 8 bits à 3 MHz) à aujourd'hui plusieurs Go sur des architectures multi-cœurs (64 bits à plusieurs GHz). De même, dans les années 80, pour diminuer les coûts de la micro-informatique grand public, on trouvait des entrées/sorties utilisant la prise Péritel du tube cathodique du poste de télévision et des lecteurs de cassettes audio pour la mémoire de masse.

25 Les périphériques

Bien que les développeurs manipulent les entrées/sorties de manière logique, il existe nécessairement une connexion physique des périphériques.

Certaines connexions sont cachées dans l'unité centrale. À commencer par les composants implantés sur la carte mère qui utilisent des connecteurs (en anglais *slot*) pour :

- les processeurs
- les bancs de mémoire vive
- une pile bouton (pour le fonctionnement d'une horloge temps réel)
- les alimentations (carte mère elle-même, CPU)
- le contrôle des ventilateurs (châssis, CPU)
- une carte graphique de base
- des émetteurs/récepteurs réseaux sans fils
- des extensions génériques (ISA, PCI)
- des extensions graphiques (AGP, PCI express)
- des mémoires de masse (IDE, SATA)
- des prises USB

D'autres connecteurs sont apparents (principalement au dos de l'unité centrale). Ils servent aux branchements des périphériques extérieurs.

La connaissance des connexions physiques permet la manipulation via le système des périphériques.

Les fichiers spéciaux par caractères (*c*) ou par bloc (*b*) identifient les périphériques par 2 numéros : le majeur et le mineur.

Le **numéro majeur** correspond à un périphérique ou type de connecteur. Il change d'un système à l'autre. Par exemple :



FIGURE 22: Connecteurs

n° majeur	type de périphérique
1	carte mère
4	terminaux virtuels
5	terminaux réels
8	disques durs
11	lecteur CD
29	vidéo
99	port parallèle

TABLE 17: Exemple de n° majeur

Le **numéro mineur** correspond à une subdivision ou un sous-adressage du périphérique. De même, il varie d'un système à l'autre. La table 18 en donne un exemple.

n° majeur	n° mineur	périphérique
1	1	mémoire
1	3	"dev nul"
1	5	produit des zéros
8	0	sda : le 1 ^{er} disque en entier
8	1	sda1 : la 1 ^{re} partition du 1 ^{er} disque
8	2	sda2 : la 2 ^e partition du 1 ^{er} disque
8	16	sdb : le 2 ^e disque en entier

TABLE 18: Exemple de n° majeur/mineur

Utilisation des périphériques

Attention ! Les périphériques par caractère ou par bloc fournissent un accès direct à l'information. Si vous modifiez un octet sur le disque et que dans le même temps le système modifie un fichier, cette dernière aura une répercussion sur une unité d'allocations et donc sur le disque. Il y aura incohérence entre les données que vous et le système modifiez par des canaux différents.

Le système ne peut gérer cette situation, qui revient à synchroniser 2 représentations, dont pour l'une d'elle vous ne donnez pas la signification de vos interventions.

Nous avons vu un cas où le système gère une double représentation : lorsqu'une image ISO (élément de l'arbre des fichiers) est montée sur un répertoire du même arbre avec la commande `mount -loop`. Mais dans cette situation, le système est maître des actions. Il peut les séquencer et mettre à jour l'une en fonction de l'autre.

Heureusement, habituellement nous n'utilisons pas des périphériques directement. Cela peut arriver avec la commande `dd` (vu au chapitre VII), qui permet la copie entre périphériques pour des sauvegardes de disque ou la création de clé USB bootable.

26 Lecture/Ecriture sous Unix

Un système d'exploitation met en place des pilotes adaptés à chaque type de périphérique. Il présente à l'utilisateur un haut niveau d'abstraction pour que l'interface ne se limite pas à lire ou écrire des bits dans des mots de communication avec le matériel.

Sous Unix cette encapsulation se fait en utilisant des fichiers. Il n'y a que de rares cas où la communication est différente (répertoire, échange de datagramme par socket).

Pour limiter les accès disques, qui sont lents et qui peuvent faire vieillir prematurely le support (nombre de réécritures limitées pour certaines mémoires), Unix n'écrit pas les données lorsqu'elles sont modifiées.

Par exemple, les fichiers temporaires (justement nommés) sont créés pour un traitement intermédiaire. Leur durée de vie peut être de quelques fractions de seconde. La plupart est détruite avant qu'Unix ait décidé de les enregistrer véritablement sur la partition `/tmp` du disque.

La commande `sync` indique au système qu'il doit synchroniser la représentation des disques en mémoire avec les périphériques (écriture des inodes et des unités d'allocation).

26.1 Bufferisation

De même, les modifications intermédiaires d'un fichier ne sont pas immédiatement répercutées. On parle de "bufferisation" (en anglais *bufferization*). Les modifications d'un fichier peuvent ainsi être stockées par ligne (jusqu'à l'arrivée d'un caractère `CR`) ou par bloc d'unités d'allocation (habituellement 1 Ko).

Sous Unix, la fermeture des fichiers est automatique à la fin d'un processus. Les tampons (en anglais *buffer*) sont vidés.

Bufferisation et fork

Le fork est un clonage fidèle de l'ensemble du processus. Les tampons mémoires en font partie. Il est donc utile de s'assurer que tous les tampons mémoires sont vides avant d'invoquer la fonction fork.

Remarquez la ligne 8 du programme au listing 29. Elle permet de choisir entre vider les tampons ou non.

```
1 int main (int argc, char** argv, char** envp) {
  int doFlush = strcasecmp ("avec", argv[1]);
  fprintf (stdout, "%s flush : ", doFlush ? "Avec" : "Sans");
  fflush (stdout);
5
  for (int i = 0; i < 3; i++) {
    fprintf (stdout, "%d", i);
    if (doFlush)
      fflush (stdout);
    if (fork () < 0)
      error ();
  }
10
  return 0;
}
```

Listing 29: Effet de bord des tampons avec fork

Le résultat suivant illustre bien le problème décrit.

```
Sans flush : 012012012012012012012012012
Avec flush : 0112222
```

Listing: Trace

Figure 23, lorsque nous lançons un programme (sans redirection dans un pipe ou un fichier) la sortie standard est connectée à l'écran. Elle est donc bufferisée en ligne (attente de retour chariot). Le processus réserve donc dans ses données un espace pour stocker ces caractères. Au moment du fork, le buffer est recopié avec son contenu. Comme il n'y a pas de *println*, les tampons seront vidés à la fin des processus. Paradoxalement, à la dernière itération après le fork, les processus ne font rien avant de quitter. C'est pourtant le simple fait de duplication qui fait passer le nombre de "012" de 4 à 8.

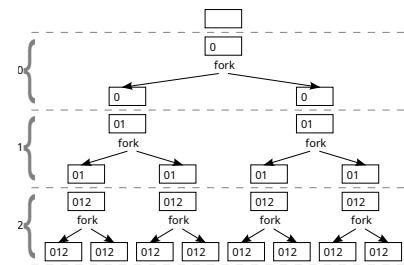


FIGURE 23: Fork avec tampon (sans flush)

Figure 24, lorsque nous prenons garde à vider les tampons avant le clonage, il n'y a aucune question à se poser sur le mode de bufferisation. Bien qu'il y ait au final 8 processus (1 père, 3 fils, 3 petits-fils et 1 arrière-petit-fils), il n'y aura que 7 chiffres écrits.

Pour compter rapidement le nombre de processus, il suffit de se souvenir qu'un fork les double. Si on double 3 fois de suite on en obtient huit fois plus ($2^3 = 8$).

Dans le cas où les tampons sont vidés, le dernier fork qui ne fait rien n'aura aucun effet de bord sur le résultat.

26.2 Les fichiers

La manipulation d'un fichier s'effectue en 3 étapes :

- l'ouverture qui initialise une position dans le fichier
- les lectures/écritures qui modifient la position
- la fermeture qui videra les tampons d'écriture s'ils existent

L'ouverture se fait suivant un mode qui déterminera entre autres la position du pointeur de lecture. Ce mode se retrouvera dans l'encapsulation faite par des langages de programmation comme C ou PHP. On ne le retrouve pas en Java qui cherche à masquer la dépendance au système d'exploitation sous-jacent.

mode	droit	création	pointeur
r	lecture seule	doit exister	au début
r+	lecture/écriture	doit exister	au début
w	lecture/écriture	efface ou crée	au début
w+	lecture/écriture	efface ou crée	au début
a	lecture/écriture	création éventuelle	à la fin
a+	lecture/écriture	création éventuelle	lecture au début / écriture à la fin

TABLE 19: Mode d'ouverture de fichier

La section 2 du manuel fournit les explications sur les fonctions de lecture, d'écriture et de modification du pointeur dans le fichier : *read*, *write* et *lseek*, ainsi que sur les caractéristiques du fichier (taille, droits) avec *stat*.

Cela a déjà été noté dans le document à plusieurs reprises. Il ne faut jamais recourir à une attente active même si elle est encadrée par une mise en sommeil. D'ailleurs quelle valeur donner ? Trop courte, on passe du temps processeur pour rien et l'on retarde l'utilisateur. Trop longue, on retarde le traitement et donc l'utilisateur.

Il est souvent possible de faire une lecture bloquante (sur un fichier ou un socket). Mais l'on peut vouloir en langage C attendre un évènement sur un ensemble de fichiers sans pour autant mettre en œuvre des tâches.

Unix prévoit un appel système spécifique : *select*. Il prend en paramètres des ensembles de fichiers dont on scrute la modification ou la disponibilité d'écriture.

Depuis plusieurs décennies, chaque problème rencontré lors de développement a amené des solutions et des mécanismes appropriés. Il faut avoir une bonne connaissance du système et de ses capacités pour développer des programmes efficaces.

26.3 Les répertoires

La manipulation de répertoires est particulière. Cela provient du fait que même en faisant abstraction des droits d'accès utilisateurs au répertoire, le système conserve la maîtrise du contenu de ceux-ci.

- La lecture est libre, car elle ne modifie pas l'arborescence de fichiers.
- Les modifications se font de manière explicite : création de nouveau fichier, re-nommage, modification de structure.

Commençons par les modifications des feuilles de l'arbre. Elles sont la conséquence d'effets de bord d'autres commandes.

Lorsque l'on demande à accéder à un fichier en écriture qui n'existe pas encore, on provoque la création d'une entrée dans un répertoire.

La suppression d'une entrée fichier dans un répertoire se fait par l'appel système *unlink*, au travers de la commande shell *rm*.

La modification d'un nom de fichier dans un répertoire se fait par l'appel système *rename* au travers de la commande *mv*.

Regardons maintenant les modifications de structure. La création et la suppression d'un nœud de l'arbre des fichiers se font avec les appels *mkdir* et *rmdir*.

Il ne reste plus qu'à détailler la lecture d'un répertoire. De nouveau, c'est le même principe dans la plupart des langages (*C, PHP, Java*). Une commande

fournit un descripteur, en l'occurrence l'appel système *opendir*. Ensuite, un pointeur géré par ce descripteur permettra de lister une à une chaque entrée avec l'appel système *readdir*.

27 Les formats de fichier

Il y a en général deux manières de lire un fichier au niveau le plus bas :

- caractère par caractère. C'est long et demande de faire une analyse (doit-on découper en fin de ligne ?) à chaque endroit du programme.
- en fournissant un tableau de caractères à remplir. Mais d'autres questions se posent : le tableau est-il assez grand ou trop grand ? Tombe-t-il à cheval sur plusieurs structures (plusieurs lignes) ?

En général, les langages fournissent des fonctions de plus haut niveau pour la lecture de fichiers textes. Les fichiers binaires nécessitent plus de précisions : quelle taille de structure ? Quelle décomposition ?

Il n'est pas possible d'utiliser l'une pour l'autre. La lecture texte permet de gérer la fin des lignes et en particulier la différence entre l'utilisation d'un marqueur simple ou d'un marqueur double.

marqueur	contrôle	hexadécimal	signification
\n	^J	0x0A	nouvelle ligne
\r\n	^M ^J	0x0D 0x0A	retour chariot / nouvelle ligne

TABLE 20: Marqueur de fin de ligne

Pour illustrer cette différence, nous allons lire en mode texte un fichier binaire en utilisant *readLine* en Java

```

1 static public void main (String[] arg) {
2     try {
3         File file = new File ("/etc/alternatives/emacs-48x48.png");
4
5         BufferedReader in =
6             new BufferedReader (new InputStreamReader
7                             (System.in));
8
9         try {
10            for (;;) {
11                String line = in.readLine ();
12                if (line == null)
13                    break;
14                System.out.println (line);
15            }
16        } finally {
17            in.close ();
18        }
19    }
20 }
```

Listing 30: Code Java utilisant *readLine*

et en utilisant *fscanf* en C.

```
1 int main (int argc, char** argv, char** envp) {
 2     char tmp [8192];
 3     while (fscanf (stdin, "%s", tmp) != EOF)
 4         fprintf (stdout, "%s", tmp);
 5     return 0;
}
```

Listing 31: Code C utilisant *fscanf*

Voici le résultat de la lecture du fichier “/etc/alternatives/emacs-48x48.png”

```
/etc/alternatives/emacs-48x48.png
00000000: 8950 4e47 0d0a 1a0a 0000 000d 4948 4452 .PNG.....IHDR
00000010: 0000 0030 0000 0030 0806 0000 0057 02f9 ...0...0....W.

readline en Java
00000000: efbf bd50 4e47 0a1a 0a00 0000 0a49 4844 ...PNG.....IHD
00000010: 5200 0000 3000 0000 3008 0600 0000 5702 R...0...0....W.

fscanf en C
00000000: 8950 4e47 1a49 4844 5249 e8ee f79e adcf .PNG.IHDR.....IHDRI
00000010: fb7d e75e dd2b 300e 8e73 675e 9d73 cf3d }.^+0..sg^..s.=
```

Listing: Trace

On remarque que Java a tenu compte des retours chariots mais les a ignorés à la recopie. Des caractères hors ASCII (au-delà de la valeur 128) ont été transcrit en UTF8.

En C, tous les caractères nuls sont ignorés puisqu'il s'agit dans ce langage du marqueur de fin de chaîne de caractères.

À l'inverse la lecture binaire de fichiers textes impose un traitement supplémentaire. Il faut donc toujours utiliser le mode de lecture et d'écriture le plus adapté. Par exemple pour la lecture en Java choisir entre : *BufferedReader*, *DataInputStream*, *FileInputStream*, *ObjectInputStream* ...

28 Fichiers textes

La lecture de caractères est plus complexe qu'il n'y paraît.

28.1 Codage des caractères

Dans le monde, nous n'utilisons pas tous le même alphabet. Ce document est écrit en latin. Nous ne sommes pourtant qu'une minorité à utiliser cette représentation (la majorité utilise les caractères chinois).

Avant d'invoquer une fonction de lecture ou d'écriture sur un fichier il faut donc préciser le codage utilisé. Du fait qu'une machine ne pense pas, elle ne

peut donner de sens à ce qui est lu et ne sera pas faire la différence entre un texte correctement décodé et un autre.

La phrase Gaëlle, où êtes vous ? encodée en *UTF8* donnera Ga  lle, o  t  s vous   ? si on croit qu'elle avait été encodée en *latin1*. Ce genre de chose arrive souvent lorsque les méta-données d'un page HTML ne décrivent pas correctement l'encodage utilisé.

Exemple	alphabet	codage
This text is written in latin	latin	ASCII
C��te texte est crit en latin accentu��	latin accentu��	ISO 8859-1
Этот текст написан на кириллице	cyrillique	KOI8-R

TABLE 21: Codage de caractères

ISO-8859-1 est devenu ISO-8859-15 après l'euro.

La raison de disposer de plusieurs encodages est de réduire la taille des caractères. Entre le nombre de lettres (majuscules et minuscules), le nombre de chiffres, les symboles et les divers caractères de contrôle, il faut environ un octet. C'est suffisant pour un alphabet mais pas tous en même temps. Il y a recouvrement des différents encodages.

La norme *UTF* (pour *Unicode Transformation Format*) regroupe tous les alphabets. Elle fonctionne en encodant les caractères les plus fr  quents sur les bits de poids faible et en utilisant le dernier pour un cha  nage vers une extension. En *UTF8* le code ASCII prend 1 octet (compatible avec l'ASCII), mais cela peut aller jusqu'à 4 octets avec des alphabets plus grands. Il existe des variantes *UTF16* et *UTF32* plus constantes dans l'interpr  tation des valeurs (m  me taille pour tous les caract  res) mais qui peuvent quadrupler la taille du fichier texte. De pr  f  rence, utilisez *UTF8*.

En cas de difficult  , Unix propose un outil permettant toutes les conversions d'encodage pour des fichiers avec la commande *recode*.

28.2 1 champ par ligne

On pourrait croire qu'un fichier binaire est structur   et qu'un fichier texte ne l'est pas. En r  alit  , il y a la m  me diversit   d'organisation. Un premier niveau de structuration des fichiers textes consiste  reconnaître les lignes.

Pour cela, on empile des filtres qui ajoutent des fonctionnalit  s  un objet. *FileInputStream* offre la lecture d'un ensemble d'octets `int read (byte[] b, int off, int len);`

À partir de cette fonctionnalité, *InputStreamReader* offre la lecture d'un ensemble de caractères `int read (char[] cbuf, int offset, int length);`.

À partir de cette fonctionnalité, *BufferedReader* peut offrir la lecture d'une ligne `String readLine();`

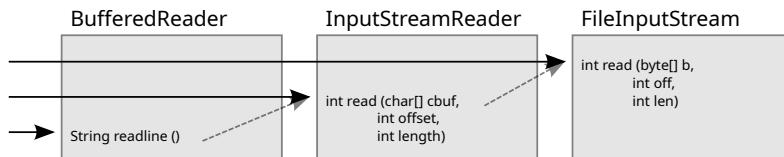


FIGURE 25: Empilement de fonctionnalité

28.3 2 champs par ligne

Les lignes sont parfois utilisées comme déclaration : variable = valeur. C'est le cas des fichiers *properties* de Java. Ces fichiers sont utilisés pour enregistrer des données de configuration ou de sauvegarde. Ils peuvent être inclus dans l'arbre des classes d'un fichier ".jar".

Le listing 32 donne un exemple de création de propriété.

```
1 static public void main(String[] args) {
  try {
    Properties prop = new Properties();
    prop.setProperty ("universite", "Bretagne-Sud");
    prop.setProperty ("diplome", "DUT Informatique");
    prop.setProperty ("module", "M3101");
    OutputStream out = null;
    try {
      out = new FileOutputStream ("demo.properties");
      prop.store (out, null);
    } finally {
      out.close ();
    }
  } catch (FileNotFoundException e) {
  } catch (IOException e) {
  }
}
```

Listing 32: Création d'un fichier propriété Java

Il produit le fichier propriété (voir listing 33).

```
1 #Sat Aug 16 18:23:28 CEST 2014
diplome=DUT Informatique
universite=Bretagne-Sud
module=M3101
```

Listing 33: Création d'un fichier propriété Java

Le listing 34 donne un exemple de lecture.

```
1 static public void main(String[] args) {
  try {
    Properties prop = new Properties();
    prop.load (new FileInputStream ("demo.properties"));
    for (Object key : prop.keySet ())
      System.out.println (key+"="+prop.getProperty ((String)key));
  } catch (FileNotFoundException e) {
  } catch (IOException e) {
  }
}
```

Listing 34: Création d'un fichier propriété Java

28.4 n champs par ligne

La structure peut être une table de base de données avec un délimiteur de champs pour constituer des "n-uplet". Unix utilise ce format pour ses propres bases par exemple avec le caractère ":" comme délimiteur (passwd, group, shadow, gshadow).

```
root:x:0:0:root:/root:/bin/bash
```

Listing: Trace

La section 1 du manuel fournit les explications sur les commandes *cut*, *sort*, *grep*, *head*, *tail*, *wc* ...

Par exemple, la liste des shell de connexion est obtenue avec la commande :

```
cut -d : -f7 /etc/passwd | sort -u
```

et donne le résultat suivant :

```
/bin/bash
/bin/false
/bin/sh
/bin/sync
/usr/sbin/nologin
```

Listing: Trace

28.5 Découpage en jetons

Les éléments d'analyse ne sont pas forcément limités à une ligne. Le fichier en entier peut être une succession de mots. C'est le cas d'un programme informatique que l'on souhaite découper en éléments lexicaux. On appelle alors ces éléments : des jetons (en anglais *token*). Les jetons sont : des identifiants, des chaînes de caractères, des nombres, des ponctuations ...

Pour reconnaître les jetons on réalise une analyse lexicale. Pour comprendre le texte suivant un langage, il faut définir en plus une grammaire et réaliser une analyse syntaxique.

Java fournit le premier niveau d'analyse (lexical) avec deux classes :

- *StringTokenizer* qui s'applique à une chaîne de caractères. Elle découpe de façon rudimentaire des phrases en donnant pour chaque jeton la chaîne de caractères reconnue.
- *StreamTokenizer* qui s'applique à un texte entier. Elle est plus élaborée et reconnaît des commentaires, des nombres, des textes entre guillemets... Elle fournit pour chaque jeton son type et sa valeur.

28.6 Modèle de texte

Un fichier de textes est certes constitué d'une suite de mots, mais il peut être chargé d'informations que l'on peut formater (en anglais *format*) pour une meilleure compréhension ou que l'on peut extraire (en anglais *parse*).

Java fournit un ensemble de classes et de méthodes dans ce but dans le paquetage `java.text`.

Voici les classes les plus significatives :

- `DateFormat`, pour les dates et les heures.
- `NumberFormat`, pour tous les nombres (`int`, `long`, `float`, `double`).
- `DecimalFormat`, pour les nombres à virgule.
- `ChoiceFormat`, pour la correspondance entre des intervalles de nombres et du texte.
- `MessageFormat`, pour des messages élaborés incluant des formats simples.

Le listing 35 donne une rapide utilisation de l'utilisation de quelques classes.

```

1 static public void main(String[] args) {
2     try {
3         SimpleDateFormat sdf = new SimpleDateFormat ("dd MM HH:mm ");
4         System.out.println (sdf.format (new Date ()));
5
6         ChoiceFormat cf = new ChoiceFormat ("-1# negatif|0#nul|1#un|1<
7             plusieurs");
8         for (int i : new int []{-1, 0, 1, 2}) {
9             System.out.println (i+" => "+cf.format (i));
10
11         MessageFormat mf =
12             new MessageFormat ("Phrase avec "+
13                 "un nombre {0,number,integer}, "+
14                 "un mot {2} une date {1,date}, "+
15                 "une heure {1,time}.");
16
17         Object[] objs = {42, new Date (), "super"};
18         String s = mf.format (objs);
19         System.out.println (s);
20
21         Object[] result = mf.parse (s);
22         for (Object obj : result)
23             System.out.println (" => "+obj);

```

```

25     } catch (ParseException e) {
26     }

```

Listing 35: Exemple de message formaté en Java

Notez qu'en lignes 14 et 15 , un même objet “temps” peut être affiché en tant que date (ligne 14) et en tant qu’heure.

Le résultat est assez surprenant :

```

1 17 08 11:51
-1 => negatif
0 => nul
1 => un
2 => plusieurs
Phrase avec un nombre 42, un mot super une date 17 aout 2014, une heure 11:51:59.
=> 42
=> Thu Jan 01 11:51:59 CET 1970
=> super

```

Listing: Trace

Notez qu'en ligne 8 de cette trace d'exécution, le même objet “temps” a été interprété

- une 1^{re} fois en tant que date (mais sans heure il a fourni “Sun Aug 17 2014 00:00:00”) et
- une 2^{de} fois en tant qu’heure (mais sans date il a fourni la date d’origine d’Unix “Thu Jan 01 1970”)

La 2^{de} a écrasé la 1^{re}.

Pour l’interprétation d’un texte en langage naturel ce mécanisme est rudimentaire. Il ne fait qu’une analyse lexicale et non syntaxique. Ne lui demandez pas l’impossible. Il ne pourra pas lever des ambiguïtés, telle l’analyse d’un mot suivi d’un nombre si le 1^{er} mot est nombre lui-même.

28.7 Expression régulière

Au niveau le plus paramétrable des outils proposés par Java, se trouve un paquetage de gestion des expressions régulières (`java.util.regex`).

Il est composé de 2 classes :

- *Pattern* qui permet de définir un modèle avec des sélections de caractères et des répétitions.
- *Matcher* qui permet d’appliquer le modèle sur une chaîne à analyser. Elle fournit toutes les interprétations possibles du modèle sur la chaîne sous forme de groupe d’analyse.

Voici un extrait de la documentation java sur les expressions régulières :

```

1 Characters
x          The character x
\\          The backslash character
\0ooo      The character with octal value 0ooo
5 \xhh      The character with hexadecimal value 0xhh
\xhhh     The character with hexadecimal value 0xhhh
\t          The tab character ('\u0009')
\n          The newline (line feed) character ('\u000A')
\r          The carriage-return character ('\u000D')
10 \f         The form-feed character ('\u000C')
\`          The alert (bell) character ('\u0007')
\`          The escape character ('\u001B')
\cx        The control character corresponding to x

15 Character classes
[abc]       a, b, or c (simple class)
[^abc]      Any character except a, b, or c (negation)
[a-zA-Z]    a through z or A through Z, inclusive (range)
[a-d[m-p]]  a through d, or m through p: [a-dm-p] (union)
20 [a-z&&[def]] d, e, or f (intersection)
[a-z&&[^bc]] a through z, except for b and c: [ad-z] (subtraction)
[a-z&&[^m-p]] a through z, and not m through p: [a-lq-z](subtraction)

Predefined character classes
.          Any character (may or may not match line terminators)
25 \d         A digit: [0-9]
\d         A non-digit: [^0-9]
\s         A whitespace character: [\t\n\x0B\f\r]
\S         A non-whitespace character: [^s]
\w         A word character: [a-zA-Z_0-9]
\W         A non-word character: [^w]

POSIX character classes (US-ASCII only)
30 \p{Lower}   A lower-case alphabetic character: [a-z]
\p{Upper}   An upper-case alphabetic character:[A-Z]
\p{ASCII}   All ASCII:[\x00-\x7F]
\p{Alpha}   An alphabetic character:[\p{Lower}\p{Upper}]
\p{Digit}   A decimal digit: [0-9]
40 \p{Alnum}   An alphanumeric character:[\p{Alpha}\p{Digit}]
\p{Punct}   Punctuation: One of !"#$%&()*+,-./:;<=>?@[\]^_`{|}-_
\p{Graph}   A visible character: [\p{Alnum}\p{Punct}]
\p{Print}   A printable character: [\p{Graph}\x20]
\p{Blank}   A space or a tab: [\t]
45 \p{Cntrl}   A control character: [\x00-\x1F\x7F]
\p{XDigit}  A hexadecimal digit: [0-9a-fA-F]
\p{Space}   A whitespace character: [\t\n\x0B\f\r]

Boundary matchers
~          The beginning of a line
50 $          The end of a line
\b          A word boundary
\b          A non-word boundary
\A          The beginning of the input
\G          The end of the previous match
55 \Z          The end of the input but for the final terminator, if any
\z          The end of the input

Greedy quantifiers
X?         X, once or not at all
60 X*         X, zero or more times
X+         X, one or more times
X{n}       X, exactly n times

```

X{n,}	X, at least n times
X{n,m}	X, at least n but not more than m times
Reluctant quantifiers	
X??	X, once or not at all
X*?	X, zero or more times
X+?	X, one or more times
70 X{n}?	X, exactly n times
X{n,}?	X, at least n times
X{n,m}?	X, at least n but not more than m times
Possessive quantifiers	
X?+	X, once or not at all
X*+	X, zero or more times
X++	X, one or more times
X{n}+	X, exactly n times
X{n,}+	X, at least n times
80 X{n,m}+	X, at least n but not more than m times
Logical operators	
XY	X followed by Y
X Y	Either X or Y
85 (X)	X, as a capturing group
Back references	
\n	Whatever the nth capturing group matched
\k<name>	Whatever the named-capturing group "name" matched
90	Special constructs (named-capturing and non-capturing)
(?:name)X	X, as a named-capturing group
(?:X)	X, as a non-capturing group
(?idsuxU-idmsuxU)	Nothing, but turns match flags i d m s u x U on - off
95 (?idsuxU-idmsux:X)	X, as a non-capturing group with the given flags i d m s u x on - off
(?=X)	X, via zero-width positive lookahead
(?!X)	X, via zero-width negative lookahead
(?<=X)	X, via zero-width positive lookbehind
(?<!X)	X, via zero-width negative lookbehind
100 (?>X)	X, as an independent, non-capturing group

Listing 36: Extrait documentation expression régulière

28.8 Internationalisation

Nous voulons des programmes génériques qui séparent fond et forme.

Comment permettre l'internationalisation (en anglais *localization*) d'une application ?

Nous devons rechercher toutes les occurrences des chaînes de caractères et les remplacer par un message qui puisse être paramétré par la langue.

En clair, au lieu d'écrire : `System.out.println ("Bonjour");`

Il faut écrire : `System.out.println (localized (msgHello));`

Et au lieu de : `System.out.println (MessageFormat.format ("Il y a {0} objets", nbObjs));`

Il faut écrire : `System.out.println (localized (msgNbObj, nbObjs));`

C'est à vous d'écrire la fonction de traduction `localized`.

Rechercher toutes les chaînes du programme veut dire : tous les titres de fenêtres, les titres des boîtes de dialogue, les champs de formulaires, les textes des boutons ...

Ce travail paraît fastidieux a posteriori, quand le programme est terminé et que rien n'a été prévu pour le multi-linguisme.

En réalité, rien n'empêche de prévoir des classes qui allègent la réécriture et qui en fin de compte factorise le formatage des textes. Nous venons de voir des composants élémentaires qu'il suffit d'assembler.

Les messages sont à enregistrer dans des propriétés que Java sait associer avec la langue choisie (voir la classe `ResourceBundle`). Ainsi, en même temps que les messages de l'application sont mis à jour, la machine virtuelle Java change le format de la date et des décimaux.

28.9 XML

Les fichiers textes peuvent enfin s'éloigner du langage naturel pour avoir une structure à la syntaxe rigoureuse pour accueillir des bases de données. C'est le cas de XML. Ce chapitre est trop court pour inclure une présentation complète d'XML. Nous supposons ce concept déjà acquis.

Les listings 37 et 38 donnent respectivement des exemples de manipulation d'XML en PHP pour la navigation dans la structure ou sa création. Le listing complet est donné en fin de chapitre.

```
1 $requete = new DOMDocument ("1.0", "utf8");
2 $requete->loadXML ($xml);
3 $racineRequete = $requete->documentElement;
4 $info = $racineRequete->getAttribute ("info");
5 $nom = $racineRequete->getElementsByTagName ("personne")->item (0)->nodeValue;
6 $texte = $racineRequete->getElementsByTagName ("tache")->item (0)->nodeValue;
```

Listing 37: Navigation XML en PHP

```
1 $reponse = new DOMDocument ("1.0", "utf8");
2 $racineReponse = $reponse->createElement ("reponse");
3 $reponse->appendChild ($racineReponse);

5 if ($info == "nouvelle")
6   $racineReponse->setAttribute ("action", "cree");
7 else
8   $racineReponse->setAttribute ("action", "supprime");

10 $racineReponse->appendChild ($reponse->createElement ("personne", $nom));
11 $racineReponse->appendChild ($reponse->createElement ("tache", $texte));
```

Listing 38: Création XML en PHP

De même en Java, les listings 39 et 40 fournissent des exemples de lecture et d'écriture de structure XML.

```
1 // =====
2 static public Document readDocument (InputStream stream)
3   throws java.io.IOException {
4     try {
5       DocumentBuilder documentBuilder =
6         DocumentBuilderFactory.newInstance ().newDocumentBuilder ();
7       Document document =
8         documentBuilder.parse (new NoWaitingNoCloseInputStream (stream));
9       document.normalizeDocument ();
10      return document;
11    } catch (javax.xml.parsers.ParserConfigurationException e) {
12      throw new IOException (e);
13    } catch (org.xml.sax.SAXException e) {
```

Listing 39: Exemple de lecture XML en Java

```
1 }
2 // =====
3 static public void writeDocument (Document document, OutputStream stream) {
4   try {
5     Source source = new DOMSource (document);
6     Result result = new StreamResult (stream);
7     Transformer xformer =
8       TransformerFactory.newInstance ().newTransformer ();
9     xformer.setOutputProperty (OutputKeys.INDENT, "yes");
10    xformer.setOutputProperty
11      ("{http://xml.apache.org/xslt}indent-amount", "2");
12    xformer.transform (source, result);
13    stream.write ("\n".getBytes ());
14 }
```

Listing 40: Exemple d'écriture XML en Java

Le listing 41 fournit un exemple réel d'utilisation des méthodes précédentes.

```
1 public void print (File file)
2   throws FileNotFoundException {
3   try {
4     DocumentBuilder documentBuilder =
5       DocumentBuilderFactory.newInstance ().newDocumentBuilder ();
6     Document challengeDoc = documentBuilder.newDocument ();
7     challengeDoc.setXmlStandalone (true);
8     Element challengeTag = challengeDoc.createElement (challengeToken);

10    // creation du document XML ...

11    XML.writeDocument (challengeDoc, new FileOutputStream (file));
12  } catch (ParserConfigurationException e) {
13    Log.keepLastException ("XMLChallenge::print", e);
14  }
15 }

16 public void parse (File file)
17   throws FileNotFoundException, IOException {
18   Document challengeDoc = XML.readDocument (new FileInputStream (file));
19   Element challengeTag = challengeDoc.getDocumentElement ();

20   challengeName = challengeTag.getAttribute (nameToken);
```

```
// lecture du document XML ...
```

Listing 41: Exemple d'utilisation XML en Java

29 Fichiers binaires

La lecture des fichiers binaires dépend de la structure du fichier. Il n'y a pas de solution générique pour toutes les situations. Cependant, dans certains cas des classes de solutions ont été développées.

Les sons sont intégrés à Java (voir le paquetage *javax.sound*).

Les images simples sont également intégrées dans le langage (voir le paquetage *javax.imageio.ImageIO*). Certains cas peuvent poser problème, comme les formats multi-spectraux ou des pixels sur 48 bits. Des extensions au langage sont disponibles avec le paquetage *javax.media.jai*. JAI (pour *Java Advanced Imaging*) qui pourront être trouvées à cette URL <http://www.oracle.com/technetwork/java/iio-141084.html>.

Les vidéos ne sont pas dans le langage mais un paquetage *com.xuggler.mediatool* permet la manipulation image par image d'un film. Le développement est disponible à l'URL <http://www.xuggler.com/>. Il s'appuie sur le logiciel libre FFmpeg (voir <http://www.ffmpeg.org/>).

Il y a enfin un moyen d'écrire et de relire des objets Java, grâce à des mécanismes de sérialisation (mise en série des attributs). De nouveau ce n'est pas si simple. Il ne faut pas que l'objet enregistré forme un graphe qui tire l'ensemble des Gigas de données de l'application. Les attributs de l'objet sont annotés de façon à indiquer, si la sauvegarde de leurs attributs est pertinente ou si par exemple on peut les recalculer à partir d'autres attributs. Un attribut qui ne participe pas à la persistance de l'objet est dit *transient*.

Le listing `listing:readObjectJava` est extrait de la javadoc de l'API Java.

```
1  FileInputStream fis = new FileInputStream ("objects.sav");
  ObjectInputStream ois = new ObjectInputStream (fis);
  int i = ois.readInt ();
  String today = (String) ois.readObject ();
  Date date = (Date) ois.readObject ();
  ois.close();
```

Listing 42: Exemple de lecture d'objets Java

30 Annexe : code source

Voici le code complet des extraits donnés dans ce chapitre.

```
1  <?php
  /*
   * (c) F. Merciol
   * Plus d'informations sur http://r305.merciol.fr
   */
 10
 15
 20
 25
 30
 35
 40
  ?>
```

Listing 43: XML en PHP

```
1  package io;
  import misc.*;
  import java.io.IOException;
  import java.io.InputStream;
  import java.io.OutputStream;
  import java.util.Hashtable;
  import javax.xml.parsers.DocumentBuilder;
  import javax.xml.parsers.DocumentBuilderFactory;
  import javax.xml.transform.OutputKeys;
  import javax.xml.transform.Result;
  import javax.xml.transform.Source;
  import javax.xml.transform.Transformer;
  import javax.xml.transform.TransformerFactory;
  import javax.xml.transform.dom.DOMSource;
  import javax.xml.transform.stream.StreamResult;
  import org.w3c.dom.Document;
  import org.w3c.dom.Element;
  import org.w3c.dom.Node;
  import org.w3c.dom.NodeList;
  import org.w3c.dom.Text;
```

```

public class XML {
    // =====
    /** une idee de Kohsuke Kawaguchi
     * http://weblogs.java.net/blog/kohsuke/archive/2005/07/socket_xml_pitf.html
     */
    static public class NoWaitingNoCloseInputStream
        extends java.io.FilterInputStream {
        public NoWaitingNoCloseInputStream (InputStream in) { super (in); }

        public int read (byte[] b, int off, int len) throws IOException {
            if (super.available () <= 0)
                return -1;
            int nb = super.read (b, off, len);
            return nb;
        }
        public void close () throws IOException {}
    }

    // =====
    static public Document readDocument (InputStream stream)
        throws java.io.IOException {
        try {
            DocumentBuilder documentBuilder =
                DocumentBuilderFactory.newInstance ().newDocumentBuilder ();
            Document document =
                documentBuilder.parse (new NoWaitingNoCloseInputStream (stream));
            document.normalizeDocument ();
            return document;
        } catch (javax.xml.parsers.ParserConfigurationException e) {
            throw new IOException (e);
        } catch (org.xml.sax.SAXException e) {
            throw new IOException (e);
        }
    }

    // =====
    static public void writeDocument (Document document, OutputStream stream) {
        try {
            Source source = new DOMSource (document);
            Result result = new StreamResult (stream);
            Transformer xformer =
                TransformerFactory.newInstance ().newTransformer ();
            xformer.setOutputProperty (OutputKeys.INDENT, "yes");
            xformer.setOutputProperty
                ("{http://xml.apache.org/xslt}indent-amount", "2");
            xformer.transform (source, result);
            stream.write ("\n".getBytes ());
            stream.flush ();
        } catch (Exception e) {
            Log.keepLastException ("XML::writeDocument", e);
        }
    }

    // =====
    static public void writeElement (Element element, OutputStream stream) {
        try {
            DocumentBuilder documentBuilder =
                DocumentBuilderFactory.newInstance ().newDocumentBuilder ();
            Document document = documentBuilder.newDocument ();
            document.setXmlStandalone (true);
            document.appendChild (document.importNode (element, true));
            XML.writeDocument (document, stream);
            stream.flush ();
        }
    }
}

```

```

85         } catch (Exception e) {
86             Log.keepLastException ("XML::writeElement", e);
87         }
88     }

89     // =====
90     public final static void putToken (Hashtable<String, String> hashtable,
91                                         String token, String value) {
92         hashtable.put (token, (value == null) ? "" : value);
93     }

94     // =====
95     public final static Hashtable<String, String> node2hashtable (Node child) {
96         Hashtable<String, String> hashtable = new Hashtable<String, String> ();
97         for (; child != null; child = child.getNextSibling ()) {
98             if (child.getNodeType () == Node.ELEMENT_NODE) {
99                 Element elementTag = (Element) child;
100                String token = child.getNodeName ();
101                NodeList nodeList = ((Element) child).getChildNodes ();
102                if (nodeList.getLength () > 0)
103                    hashtable.put (token,
104                               ((Text) nodeList.item (0)).getWholeText ());
105            }
106        }
107        return hashtable;
108    }

109    // =====
110    public final static void hashtable2node (Document document, Element container,
111                                             Hashtable<String, String> hashtable)
112    {
113        for (String token : hashtable.keySet ()) {
114            Element tag = document.createElement (token);
115            tag.appendChild (document.createTextNode (hashtable.get (token)));
116            container.appendChild (tag);
117        }
118    }
119}

120

```

Listing 44: XML en Java

Septième partie

Système de gestion de fichiers

Les premiers systèmes de gestion de fichiers sont apparus dans les années 60 (voir https://fr.wikipedia.org/wiki/Système_de_fichiers). Il y en a aujourd'hui un nombre important. Chaque système d'exploitation cherche souvent à définir sa gestion de ses fichiers en fonction de ses caractéristiques propres.

31 Besoin de fichiers

Un Système de Gestion de Fichiers (SGF) répond à plusieurs besoins :

- désignation compréhensible par un humain (utilisation de l'alphabet)
- hiérarchisation des informations (arbre, voire graphe)
- typage des informations (extension et/ou étiquette embarquée)
- intégrité (droits d'écriture)
- confidentialité (droits de lecture)
- fiabilité (redondance des données, stabilité du système)
- disponibilité (réplication en réseau)

Il n'est pas toujours nécessaire de mettre en place toutes ces caractéristiques en fonction de l'usage du système d'exploitation. À quoi sert de définir un système complexe de propriétés et de droits d'accès quand il n'y a qu'un seul utilisateur ? C'est le cas pour un enregistreur numérique de film de salon, un appareil photo ou un appareil électroménager.

À quoi sert d'utiliser des lettres de l'alphabet pour désigner des fichiers quand les noms donnés automatiquement sont des nombres ? De nouveau, c'est le cas d'enregistreurs automatiques, qui n'ont pas le temps d'interagir avec l'utilisateur et vont donner un nom en fonction de la date ou d'un numéro d'ordre.

31.1 Désignation

Dans la mesure où un utilisateur doit désigner des fichiers, on préférera utiliser des symboles signifiants. Il vaut mieux désigner le répertoire d'un utilisateur par son nom ("jean.martin") plutôt que par "e42".

Les systèmes peuvent utiliser différents alphabets (pour nous le latin accentué). Parfois, ils permettent l'utilisation des majuscules et des minuscules. Il reste cependant des caractères interdits qui sont utilisés dans la syntaxe du langage de commande du système.

Certains caractères spéciaux sont réservés comme :

- ? remplace un caractère
- * remplace plusieurs caractères
- / séparateur de chemin
- \$ nom de variable
- & lancement en arrière-plan
- ...

En règle générale, évitez tous les caractères de ponctuation. Par ailleurs, évitez les caractères accentués qui ne seront pas nécessairement encodés de la même manière d'un système à l'autre et compliquent la sauvegarde de répertoire ou l'échange entre systèmes.

Il existe également des systèmes de gestion de fichiers qui limitent la taille des noms (par exemple 8 caractères pour les noms et 3 pour les extensions).

Enfin, certains fichiers de configuration sont présents dans un contexte (un répertoire) mais nuisent à la lecture du contenu du répertoire. Sous Unix, les fichiers commençant par un point sont dit "cachés" (mais pas secrets). C'est le cas de ".." le répertoire courant et "../" le répertoire précédent dans l'arbre. Par défaut, la commande "ls" ne les affiche pas. Il suffit d'ajouter l'option "-a" pour qu'ils apparaissent (ls -a).

31.2 Typage

Certains systèmes ont besoin de connaître la nature des données pour réaliser des opérations appropriées.

Principalement les systèmes distinguent la

nature des données contenues dans les fichiers par :

- L'extension. C'est une suite de caractères qui suit le dernier point dans le nom d'un fichier (.c pour un fichier source C, .out pour un exécutable ...).
- Un "nombre magique" (voir https://fr.wikipedia.org/wiki/Nombre_magique_%28programmation%29). C'est une suite de caractères, mais cette fois incluse en tête du fichier (#! pour les scripts, GIF pour les images GIF ...)

- Un répertoire dédié. Des données particulières sont regroupées de façon à ce que le système les identifie (des variables dans /proc/, des scripts dans cgi-bin/ ...).

31.3 Hiérarchie

La représentation hiérarchique est de loin la plus utilisée par les systèmes de gestion de fichiers.

Parfois elle se limite à 2 niveaux : les répertoires ne peuvent contenir que des fichiers. Le plus souvent sans limitation logique (il y a toujours une limitation physique de capacité de stockage).

- Les feuilles sont les fichiers (d'accord, sous Unix les répertoires sont également des fichiers).
- Les nœuds de l'arbre sont appelés : dossier, catalogue ou répertoire.

Un fichier est désigné par un chemin (en anglais *path*).

Le chemin débute par la racine de l'arbre sur la mémoire de masse concernée (il n'y a qu'une racine sous Unix "/").

On indique la succession des noms des nœuds (des répertoires) jusqu'au fichier inclus. Les noms sont séparés par un caractère indiquant le changement de répertoire ("/" sous Unix).

Définition 40 (arbre). Un arbre est un "graphe" "orienté" "connexe" "sans cycle" avec une seule "racine".

En réalité, les systèmes de gestion de fichiers autorisent des liens (ou raccourcis) qui rompent le caractère sans cycle. Il peut donc exister plusieurs chemins pour désigner une même donnée. Il peut également exister des boucles lorsqu'un lien fait référence à un nœud plus proche de la racine.

Sous Unix on distingue 2 types de liens :

- physique (commande `ln`). Qui ne peuvent se faire qu'au sein d'un même volume (pour simplifier une même partition d'un disque dur). Nous verrons que dans ce cas, deux chemins désignent le même "inode" sous Unix.
- logique (commande `ln -s`). Qui peuvent se faire y compris avec des espaces disques non présents au moment de la création du lien. Il s'agit simplement d'un fichier qui contient un chemin absolu ou relatif et étiqueté comme étant un lien.

31.4 Intégrité

Définition 41 (intégrité). L'intégrité est la propriété de garantir la cohérence, la fiabilité et la pertinence des données. Un moyen consiste à interdire la modification des fichiers.

Il serait logique de penser que les fichiers sont faits pour être écrits. Pourtant, il arrive que l'on souhaite les protéger de toute modification, y compris de la part de leurs propriétaires. C'est par exemple le cas des fichiers gérés par "subversion" (un logiciel de gestion de versions de développement). Il crée un répertoire ".svn" qui nous appartient, mais que l'on ne peut pas modifier.

Pour trouver les fichiers protégés sur votre compte vous pouvez lancer la commande :

```
1 | find ~ ! -perm /a+w -print
```

Listing 45: Recherche de fichiers non modifiables

Vous trouverez par exemple les fichiers ".svn/entries"

Un autre moyen est de déléguer le traitement des données à un utilisateur fictif ("mail" pour la messagerie, "root" pour la gestion des mots de passe ...). Seul cet utilisateur pourra réaliser les modifications en votre nom sous le contrôle des règles mises en place. Par exemple, le fichier "/etc/passwd" ne peut être écrit directement par l'utilisateur.

On écrit alors des programmes qui font ce travail au nom de l'utilisateur fictif. Sous le système Unix, c'est le "bit s" qui indique cette faculté. Bien entendu, seul le propriétaire peut indiquer que l'on prend sa personnalité en lançant ce programme. Par exemple, lorsque vous exécutez le programme "/usr/bin/passwd", vous devenez "root"! (mais uniquement pour ce pourquoi le programme a été écrit ☺.)

```
1 | $ ls -l /etc/passwd /usr/bin/passwd
-rw-r--r-- 1 root root 1548 sept. 1 08:32 /etc/passwd
-rwsr--r-x 1 root root 42824 sept. 13 2012 /usr/bin/passwd
```

Listing 46: Bit s

Il est indispensable que les programmes avec "bit s" ne puissent pas être modifiables par tout le monde.

31.5 Confidentialité

Certaines données ne doivent pas être lues par d'autres utilisateurs. C'est le cas des mots de passe ou des clefs secrètes de communication. Par exemple, les clefs utilisées par SSH ("Secure SHell") ne doivent pas être lues par d'autres (fichier "`~/.ssh/id_dsa`").

Cette confidentialité est toute relative, puisque l'administrateur a toujours la possibilité de lire tous les fichiers. Si vous avez un secret à garder, le mieux est de ne jamais le numériser.

31.6 Fiabilité

Un système de gestion de fichiers doit être fiable, y compris en cas de coupure de courant.

Le système d'exploitation passe son temps à modifier des fichiers. On peut distinguer des différences de fréquences d'utilisation en fonction des parties de l'arbre des fichiers :

- La racine ("boot"). Utilisée en lecture est nécessaire au démarrage, il évolue rarement. Mais, sa dégradation empêche tout démarrage.
- Le système. Utilisé principalement en lecture, il n'est mis à jour qu'au gré des changements de versions.
- Les programmes. Utilisés principalement en lecture, ils sont mis à jour au gré des changements de versions. Ils peuvent être conservés, même en cas de mise à jour du système, mais ne nécessitent pas de sauvegarde.
- Les données des utilisateurs. Fréquemment écrites, il faut leur adjoindre un système de sauvegarde.
- Les données des programmes. Contiennent principalement des informations de configurations que l'on peut souvent reconstruire.
- Le répertoire temporaire. En écriture permanente, il sera le premier à être en état instable en cas de coupure de courant.
- La zone d'extension de la mémoire vive ("swap"). Il est donc inutile de la conserver d'un arrêt à l'autre.

C'est pourquoi, on peut conseiller de les séparer physiquement dans des partitions distinctes du disque :

- /boot
- /

- /opt
- /home
- /var
- /tmp
- swap

Le mieux est bien entendu de ne pas perdre les données en cas de crash du système. Un système de gestion de fichiers met en place des stratégies pour que les écritures se fassent de façon atomique. Par exemple, lors de la modification de fichier, le système conserve pendant un temps très court les deux versions du fichier. Il prévoit une modification minime (un pointeur, voire un bit) pour passer de l'un à l'autre. De cette façon, les fichiers restent dans un état stable.

Il existe également des mécanismes de journalisation, où le système décrit les modifications qu'il s'apprête à effectuer. En cas d'interruption, il reprend la même séquence (elle donnera le même résultat). Quand elle est terminée, il efface le journal.

31.7 Disponibilité

D'autres mécanismes, se fondant sur la réplication des disques, permettent de pallier à des défaillances, y compris de disque et de les remplacer en cours de fonctionnement du système. Cela nécessite de multiplier les disques durs. Les fichiers (ou parties de fichiers) se retrouvent répliqués sur plusieurs disques. En cas de panne, on arrête le disque défectueux (mais pas le système), on le remplace et le système recopie les éléments manquants pendant que l'utilisateur les modifie.

C'est le cas de regroupements redondants de disques indépendants (ou RAID pour "Redundant Array of Independent/Inexpensive Disks", voir https://fr.wikipedia.org/wiki/RAID_%28informatique%29)

- RAID 1 : Disques en miroir
- RAID 5 : Volume agrégé par bandes à parité répartie

Ces mécanismes se trouvent également dans des caches, comme avec le système NFS (pour "Network File System"), où les fichiers se trouvent sur un serveur distant. Ils sont copiés sur une machine locale et verrouillés. Les modifications sont renvoyées en fin d'utilisation.

32 Support physique

Nous venons de décrire les principes logiques de fonctionnement. Il faut les appliquer avec des contraintes physiques (voir https://fr.wikipedia.org/wiki/Disque_dur).

La gestion de masse de données utilise encore des supports mobiles (tournant) car plus efficaces en rapport de leur coût. Les disques magnétiques et optiques (voire les deux) sont structurés en blocs de données (1 Ko ou 1/2 Ko). Ces blocs forment de façon logique une suite linéaire d'informations.

Ils doivent avoir un diamètre raisonnable. Les données sont situées sur des cercles concentriques, leurs circonférences augmentent avec le rayon. Il existe donc une relation entre vitesse de rotation, espacement des données et vitesse de lecture.

Les concepteurs se sont trouvés face aux choix suivants :

- soit avoir des vitesses (rotation et lecture) constantes, mais dans ce cas les bits sont plus longs et espacés à mesure que l'on s'éloigne du centre.
- soit distance régulière entre bits et lecture constante sur le périmètre du cercle et donc faire varier la vitesse de rotation des disques.
- soit enfin, distance régulière entre bits et rotation constante et avoir une vitesse de rotation constante mais une vitesse de lecture/écriture différente.

L'inertie mécanique rend difficile la variation instantanée des disques (déplacement des têtes du centre vers la périphérie). La modification de vitesse d'écriture rend aléatoire le temps d'accès aux données. Le choix a été fait de conserver la même quantité de données par rayon. C'est donc plus facile à gérer pour les informaticiens. C'est en revanche une perte considérable de support et donc une dépense inutile pour le propriétaire des données.

Un disque est donc divisé en :

- Tête. Elle correspond à une surface d'écriture.

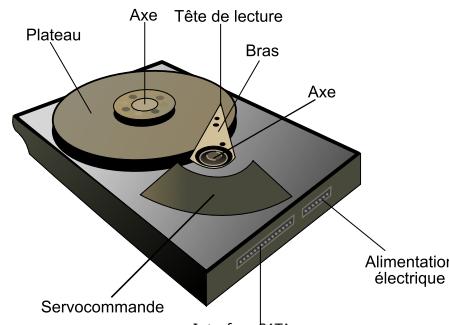


FIGURE 26: Architecture d'un disque

- Cylindre. Il correspond à l'empilement à rayons constants de cercle des différentes têtes.

- Secteur. Il correspond à un arc de cylindre (le périmètre d'une portion de camembert pour être imagé).

(voir <https://fr.wikipedia.org/wiki/Cylindre/T%C3%A4te/Secteur>)

L'intersection d'une tête et d'un cylindre fournit une piste. L'intersection d'une piste et d'un secteur fournit un bloc de données.

À noter, qu'autrefois (époque des disquettes à une face), on parlait plutôt de piste et secteur. Il y avait confusion entre un bloc et un secteur (potentiellement sur plusieurs faces).

Définition 42 (bloc de données). Le bloc est un composant élémentaire de conservation de données. Sur un support tournant, il est l'intersection d'une tête d'un cylindre et d'un secteur. Ils ont en général une taille de 1 024 ou 512 octets (1 Ko ou 1/2 Ko).

Vous pouvez déterminer la géométrie de votre disque avec les commandes /sbin/fdisk ou /usr/sbin/gparted.

Il y a deux façons de désigner un bloc :

- CHS (cylinder-head-sector) comme son nom l'indique
- LBA (Logical Block Addressing) utilisant le numéro du bloc dans une vision linéaire.

L'adressage LBA a remplacé celle du CHS, car elle généralise la désignation des blocs sur tous les supports (bandes, CD, clefs usb, ...).

33 Partition

Définition 43 (partition). Une partition est une partie logique du disque dur.

Le fait de pouvoir découper le disque dur en différentes partitions permet d'isoler des arborescences de fichiers

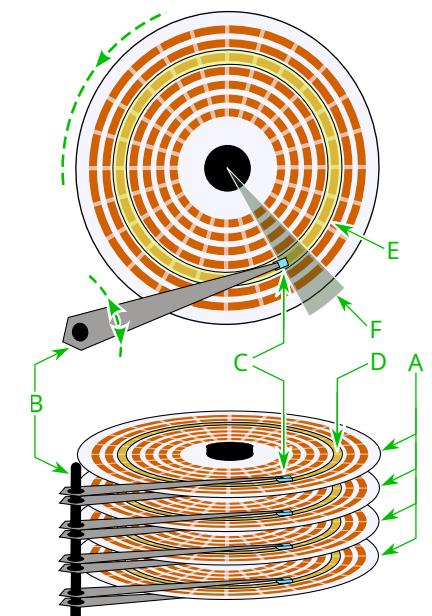


FIGURE 27: Géométrie d'un disque. A : plateaux - B : bras - C : tête - D : cylindre - E : piste - F : secteur

(comme vu précédemment). Cela permet également de disposer de plusieurs systèmes de gestion de fichiers. C'est indispensable si l'on veut héberger plusieurs systèmes d'exploitation différents sur la même machine.

Les plus répandus sur les PC sont :

- FAT pour File Allocation Table utilisé par Windows
- NTFS pour New Technology File System utilisé par Windows à partir de 1993
- ext pour Extended File System Linux
- swap zone d'échange de la mémoire vive sous Unix.

Nous détaillerons le système de fichiers d'Unix par la suite.

34 Table de partition

Un système d'exploitation ne peut pas deviner combien, ni de quelle nature sont les partitions sur un disque. Il faut une structure qui organise le disque.

Définition 44 (partitionnement). Le partitionnement est une structure qui permet de retrouver toutes les partitions d'un disque : soit par une table regroupant tous les descripteurs, soit par chaînage des descripteurs en tête des partitions.

Les plus répandus sont :

- MBR pour Master Boot Record (système propriétaire)
- GPT pour GUID Partition Table (récent et qui se généralise)

Pourquoi faire évoluer les structures anciennes ? Le MBR a pour défaut d'être limité à 4 partitions enregistrées dans une table. Une évolution a permis d'étendre cette table avec une partition appelée "étendue". On parle alors de 3 partitions primaires. Les autres étant des partitions logiques définies dans la partition étendue.

GPT lève cette contrainte. Il n'y a plus de différences entre les partitions qui se trouvent chaînées.

35 Structure d'un système de gestion de fichiers

À partir de maintenant, on considère un support comme une suite linéaire de blocs d'octets.

Un système de gestion de fichiers va devoir gérer des :

- données contenues dans les fichiers. Elles seront découpées pour être enregistrées dans des unités d'allocation
- métadonnées qui décriront l'organisation du système de fichiers.
 - la table d'implantation des blocs du fichier
 - le descripteur de fichiers
 - les catalogues (répertoires) représentant l'arborescence de fichiers
 - la table d'allocation des blocs disponibles sur le disque.

35.1 Unité d'allocation

Un fichier ne peut être écrit directement sur le disque. Il faut le découper en blocs qui ne correspondent pas nécessairement au format du disque. Certains blocs sont utilisés pour enregistrer des informations propres au système de gestion, d'autres pour les fichiers.

Définition 45 (unité d'allocation). Une unité d'allocation est un bloc d'enregistrement élémentaire du point de vue du système de gestion de fichiers qui contient une partie d'un fichier.

Les blocs sur les disques font en général 1/2 Ko (512 octets). Un bloc du système de gestion de fichiers peut faire 1 Ko. Cela signifie que si un fichier fait 2050 octets, les 2048 premiers octets seront sur deux blocs, et les deux derniers sur un troisième. Cela fait 1022 octets de perdus. Mettre en place une gestion des octets perdus ferait perdre bien plus de place que la somme des bouts de fichiers perdus. Ce serait également une perte d'efficacité en temps. La taille choisie par le système est un compromis qui se vérifie d'expérience en fonction de l'usage du système. De nouveau, c'est donc plus facile à gérer pour les informaticiens. C'est en revanche une perte considérable de support et donc une dépense inutile pour le propriétaire des données.

35.2 Descripteur de fichiers

Rappelons les informations essentielles à conserver pour garantir toutes les propriétés désirées :

- Le nom du fichier
- Son type
- Son propriétaire et son groupe
- Les droits
- Les dates (de création, modification, accès)
- Une table d'implantation sur le disque
- ...

Ces informations diffèrent d'un système à l'autre.

Définition 46 (descripteur de fichier). Un descripteur de fichier ne contient pas le fichier, mais ses métadonnées, c'est-à-dire l'ensemble des informations nécessaires à sa gestion comme le propriétaire et les droits, mais aussi son occupation sur le disque.

35.3 Table d'implantation

Définition 47 (table d'implantation). La table d'implantation est la liste des unités d'allocation utilisées pour enregistrer un fichier.

Sa place occupée par la table se calcule en fonction de la taille :

- du fichier,
- des unités d'allocation,
- des tailles des adresses nécessaires pour désigner les unités d'allocation.

Pour donner un exemple en simplifiant les calculs, supposons qu'un fichier fait 1 Go, que nos blocs fassent 1 Ko, il devrait être découpé en 1024×1024 blocs de 1 Ko. Ce nombre nécessite un entier sur 20 bits. Partons sur des entiers de 24 bits (3 octets pouvant adresser jusqu'à 16 Go).

Si l'on souhaite adresser un fichier de 1Go, il faut $1024 \times 1024 \times 3$ octets (donc 3 Mo = 0,3%).

Vous comprenez d'une part la démesure entre la taille du fichier et la taille de la table d'implantation. Plus le bloc sera grand, moins on perdra de place en table d'implantation.

Vous comprenez également que si l'on fixe une taille constante moyenne pour tous les descripteurs, on aura rempli le disque avec uniquement des descripteurs.

De nouveau, un compromis est nécessaire entre pouvoir désigner les plus gros fichiers possibles et ne pas perdre trop de place. Nous verrons qu'Unix a une solution élégante de répondre à ce problème.

35.4 Catalogue

Définition 48 (catalogue). Un catalogue (répertoire) va enregistrer la liste des fichiers dans un nœud de l'arbre. Ce sont les références entre catalogue qui forment la structure arborescente du système de fichier.

Le système de gestion de fichiers met en place un mécanisme pour différencier les catalogues des fichiers.

35.5 Table d'allocation

Définition 49 (table d'allocation). Une table d'allocation est un résumé compact de l'occupation des unités d'allocation sur le disque.

Lorsqu'on demande la création d'un fichier, il serait fastidieux de parcourir l'ensemble du disque pour chercher quelle unité est disponible. On utilise alors des bits qui représentent l'occupation d'un ensemble d'unité d'allocation. Le système gère également un pointeur sur le premier élément de la table qui indique une zone libre. À chaque création ou suppression de fichier le pointeur et la table sont mis à jour.

36 Le SGF Unix

Nous allons illustrer la description générale qui vient d'être faite avec la solution proposée par Unix.

La taille des différents éléments est déterminée au moment de la création du système de fichier. La commande `mkfs.ext4` choisira automatiquement les paramètres les plus appropriés en fonction de la taille de la partition (compromis exposé plus haut).

Pour connaître les caractéristiques de votre système vous pouvez utiliser la commande

tune2fs -l. Les valeurs peuvent être :

- Bloc size : 4096 (4 Kb)
- Inode size : 256 (1/4 Kb)

36.1 Unité d'allocation

Une unité d'allocation correspond à plusieurs blocs sur le disque dur (8 blocs de 512 dans notre exemple).

36.2 Inode

Le descripteur de fichier se nomme inode. La particularité d'Unix est qu'il ne contient pas le nom du fichier. L'avantage est qu'un fichier peut posséder plusieurs noms dans plusieurs répertoires. Il suffit de faire référence au même inode.

Un inode contient :

- la taille du fichier en octets
- l'UID : identifiant du propriétaire du fichier
- le GID : identifiant du groupe auquel appartient le fichier
- le mode du fichier qui détermine quel utilisateur peut lire, écrire et exécuter le fichier
- des chronomarques (timestamp)
 - ctime : date de dernière modification de l'inode
 - mtime : date de dernière modification du fichier
 - atime : date de dernier accès à l'inode
- le nombre de liens physiques sur cet inode
- la table d'implantation des unités d'allocation

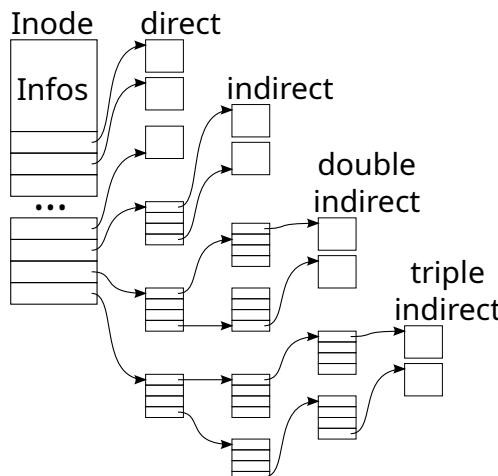


FIGURE 28: Table d'implantation

Il existe également des informations supplémentaires comme pour les fichiers spéciaux (connectés à un périphérique avec le numéro majeur et mineur).

À chaque nouveau nom, le compteur de lien est augmenté. À chaque suppression de nom le compteur de lien est décrémenté. Le fichier est supprimé quand le compteur tombe à zéro.

Le compteur prend également en compte l'utilisation par la mémoire. Un programme qui s'exécute verra son compteur passer à 2. Il peut se supprimer de son répertoire (compteur de nouveau à 1) et continuer son exécution. À sa terminaison, il est délié de la mémoire (son compteur passe à 0) et réellement supprimé.

Remarquez qu'un fichier supprimé et simplement dé-référencé, son contenu n'est pas rempli de zéro. On peut donc éventuellement retrouver des informations, mais l'on ne sait pas où les chercher.

36.3 Table d'implantation

La taille de la table d'implantation dépend de la taille totale de l'inode. Si nous avons :

- N la taille de la table d'allocation
- TI la taille des inodes
- TD la taille des informations dans l'inode (autre que la table d'allocation)
- TA taille des adresses pour référencer les unités d'allocation
- TUA taille des unités d'allocation

Nous obtenons : $N = (TI - TD)/TA$

La taille maximum des fichiers devrait donc être : $N * TUA$

Pour dépasser cette limite, la table fonctionne avec des niveaux d'indirection de références :

- Les $N - 3$ premiers éléments sont des références directes à des unités d'allocation.
- le suivant désigne une unité d'allocation qui contient des références vers des unités d'allocation (référence indirecte)
- l'avant-dernier désigne une unité d'allocation de référence indirecte (double indirection)
- le dernier désigne une unité d'allocation de référence doublement indirecte (triple indirection)

Si R est le nombre de références d'unité d'allocation que l'on peut mettre dans une unité d'allocation, le nombre d'unités maximum d'un fichier est : $(N - 3) + R + R^2 + R^3$. La taille maximum du fichier est ce nombre multiplié par la taille d'une unité d'allocation.

Une unité de 4 Ko peut contenir 512 pointeurs de 64 bits. Dans ce cas $R = 512$. Imaginez la taille des fichiers que l'on peut adresser.

L'avantage de cette approche est que pour les petits fichiers un inode suffit pour désigner tous les blocs. Pour les gros fichiers, il y a au maximum 3 indirections pour accéder à 4 Ko de données.

36.4 Répertoire

Les catalogues sous Unix se nomment répertoires. Ils sont stockés par le système comme des fichiers. En revanche on y accède par des commandes spécifiques (`mkdir`, `rmdir`...) C'est un exemple concret d'une donnée qui appartient à un utilisateur, mais que ce dernier ne peut modifier directement pour garantir son intégrité.

Ils ne contiennent que des couples

- inode
- nom de fichier

Un répertoire contient toujours au minimum 2 entrées :

- “.” le répertoire lui-même
- “..” le répertoire précédent dans l'arbre.

Puisqu'un répertoire possède un nom et qu'il se désigne lui-même, son compteur de liens est au moins de 2. Plus s'il possède d'autres noms dans d'autres répertoires (par exemple, dans ses fils il se nomme “..”).

36.5 Volume Unix

Une partition Unix contient :

- un super-bloc qui décrit la structure de la partition, la taille des éléments et des informations de montage
- des informations de journalisation
- une table d'allocation
- une table des inodes
- des unités d'allocation

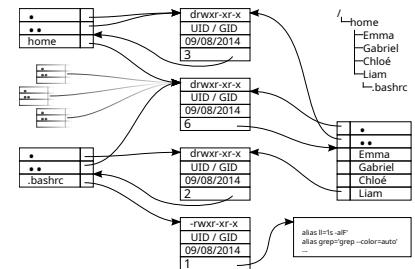


FIGURE 29: Exemple d'inodes

Définition 50 (superbloc). Un superbloc est un enregistrement des caractéristiques d'un système de fichiers, dont sa taille, celle des blocs, les blocs libres ou occupés, leur nombre...

En TD et TP, nous traiterons celui de Minix en exemple.

37 Montage

Certains systèmes gèrent des forêts de noms (une racine par partition). Unix unifie l'ensemble des partitions en les raccordant sur un arbre principal.

Définition 51 (montage). Le montage (commande “`mount`”) permet de superposer la racine d'un système de fichiers sur un nœud de l'arbre du système en cours d'exécution.

Il faut donc nécessairement créer un répertoire qui représentera la racine de la partition montée. Le répertoire de montage peut ne pas être vide. Durant la période de montage, les informations de ce répertoire sont masquées, la consultation du répertoire étant automatiquement redirigée vers la partition montée. Après le démontage de la partition, les informations réapparaissent.

Unix est capable de gérer plusieurs systèmes de fichiers. On peut donc construire un arbre en faisant succéder au long d'une branche des systèmes de natures différentes.

Les paramètres d'un montage comprennent :

- le type de montage

- la partition, désignée par le périphérique ou mieux par l'identifiant de la partition (indépendant de l'ordre de connexion des disques sur la carte mère).
- le répertoire de montage

Unix prévoit également des options importantes.

- L'option “bind” permet de monter une partie de l'arbre de nom à un autre endroit de l'arbre. Existant dès les premières versions d'Unix. L'option permet la mise au point du développement du système en lançant d'autres versions sur lui-même.
- L'option “loop” permet également de remonter des données ailleurs mais sous un autre format (par exemple un fichier ISO monté comme un CD). Le système est ainsi averti quand des tentatives d'écriture sur un même octet par deux moyens différents ont lieu (via le fichier ISO ou via le répertoire de montage).

Unix dispose également de mécanismes de montage via le réseau.

Il est possible de spécifier des auto-montages qui font apparaître le répertoire monté au moment où vous y accédez. Cela peut être pratique mais difficile à manier en cas de montage croisé, car il devient difficile de démonter l'ensemble.

38 Type de fichier

Tout dans Unix est fichier, y compris les périphériques. On les distingue par leur type et l'inode contient les informations nécessaires à leurs utilisations. Avec la commande `ls -l` le premier caractère désigne le type.

Caractère	Mode	Type de fichier
p	0x1	file (ou “named pipe”)
c	0x2	spécial en mode caractère
d	0x4	répertoire (directory)
b	0x6	spécial en mode bloc
-	0x8	fichiers ordinaires
l	0xa	lien symbolique
s	0xc	socket

TABLE 22: Les types des fichiers sous Unix

39 Les commandes shell

Voici maintenant les commandes utiles en rapport avec ce que nous venons de décrire dans ce chapitre.

Définition 52 (`fdisk`). `fdisk (format disk)` est un outil en mode texte pour gérer la géométrie d'un disque et créer des partitions

Exemple :

```
/sbin/fdisk /dev/sda
```

Définition 53 (`gparted`). `gparted (GNU Parted donc interface graphique Gnome)` est un outil graphique pour faire la même chose que `fdisk` (les deux sont complémentaires).

Exemple :

```
gparted /dev/sda
```

Définition 54 (`mkfs`). `mkfs (make file system)` formate une partition (ou un fichier) suivant le système de fichier indiqué. `mkfs` fera appel à d'autres programmes adaptés en fonction du type de système de fichiers.

Exemple :

```
mkfs.ext4 /dev/sdaX
```

Définition 55 (`fsck`). `fsck (file system check)` vérifie l'état d'un système de fichiers et sa fragmentation. Il peut réaliser des opérations de réparation automatique.

Exemple :

```
fsck /dev/sdaX
```

Définition 56 (`dd`). `dd (disk to disk copy copie bit à bit de disques)` permet la recopie d'une partition sur une autre ou d'un fichier vers un autre. On l'utilise également pour allouer un fichier continu qui peut servir de travail sur un système de fichiers en TD.

Exemple :

```
dd if=/dev/zero of=monFichier64Ko bs=1024 count=64
```

Définition 57 (mount). mount (*monte*) associe une partition dans l'arbre de fichier existant. En premier on indique le périphérique, en second le point de montage (un répertoire qui doit exister).

Exemple :

```
sudo mount -t iso9660 -o ro,loop /backup/kubuntu.iso /mnt/
```

Définition 58 (mkdir). mkdir (*make directory*) crée des répertoires. L'option -p crée en une fois une arborescence.

Exemple :

```
mkdir -p r1/r2/r3
```

Définition 59 (touch). touch permet de créer un fichier ou de changer sa date s'il existe déjà.

Exemple :

```
touch nouveauFichier
```

Définition 60 (ln). ln (*link*) crée un lien “physique” (partage du même inode) entre deux fichiers. Attention : il fonctionne comme “cp” : en premier le fichier qui existe, ensuite le fichier à créer (ou le répertoire dans lequel faire le lien).

L'option -s crée un lien symbolique.

Exemple :

```
ln -s fichierExistant ouPas monRépertoireDeLiens/
```

Définition 61 (rm). rm (*remove*) supprime des fichiers. L'option -r supprime en une fois une arborescence.

Exemple :

Avez-vous vraiment besoin d'un exemple ?

Définition 62 (tar). tar (*tape archiver*) outil conçu à l'origine pour l'archivage de bande magnétique (comme son nom l'indique). Il conserve les droits, dates et liens.

Exemple local :

```
(cd r1 ; tar -cf - .)|(cd r2; tar -xf -)
```

Exemple distant :

```
tar -cJf - monRépertoire|ssh moi@autreMachine: "cd r2; tar -xjf -"
```

Définition 63 (rsync). rsync (*remote synchronization*) est un outil extrêmement performant pour la copie de fichier ou la sauvegarde. Il possède des options pour réaliser des sauvegardes incrémentales. Il copie en un temps record en occupant toute la bande passante. Il conserve les droits, les dates et la correspondance entre inodes (par exemple pour les liens physiques).

Exemple :

```
rsync --progress -rlogpt -e ssh /home/chezMoi moi@autreMachine:/home/
```

Exemple 1^{re} archive :

```
TODAY=$(date -I)
```

```
YESTERDAY=$(date -I -d "1 day ago")  
rsync -a /home/src --delete --link-dest=/svg/$YESTERDAY /svg/$TODAY
```

Exemple archives suivantes :

```
TODAY=$(date -I)
```

```
YESTERDAY=$(date -I -d "1 day ago")
```

```
rsync -a /home/src --delete --link-dest=/svg/$YESTERDAY /svg/$TODAY
```

Définition 64 (kompare). kompare est un outil de comparaison graphique de fichiers. Il visualise les portions de texte ajoutées/modifiées/supprimées et permet de valider ou refuser des changements.