Utilisation des Systèmes Informatiques (HAI103I)

Michel Meynard

UM

Univ. Montpellier

Table des matières

Plan

Organisation de l'enseignement I

- 8 Cours par Pierre pompidor et Michel Meynard
- n TDs par groupes de 40
- n TPs avec 2 enseignants dans la même salle
- Objectif : savoir utiliser une machine sous Unix **en comprenant ce que l'on fait** (non au double-click ;))
- évaluation par un contrôle écrit terminal en fin de semestre

Introduction I

Définition d'un SE (*Operating System*)

Couche logicielle offrant une interface entre la machine matérielle et les utilisateurs.

Objectifs

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

Introduction II

Services

- multiprogrammé (ou multi-tâches) préemptif (isolement des processus)
- multi-utilisateurs (authentification)
- pilotage des périphériques toujours plus nombreux
- fonctionnalités réseaux (partage de ressources distantes)
- communications réseaux (protocoles Internet)
- personnalisable selon l'utilisation (développeur, multimédia, SGBD, applications de bureau, . . .)

Introduction III

Pricipaux OS

Microsoft Windows pricipal système sur ordi. personnels

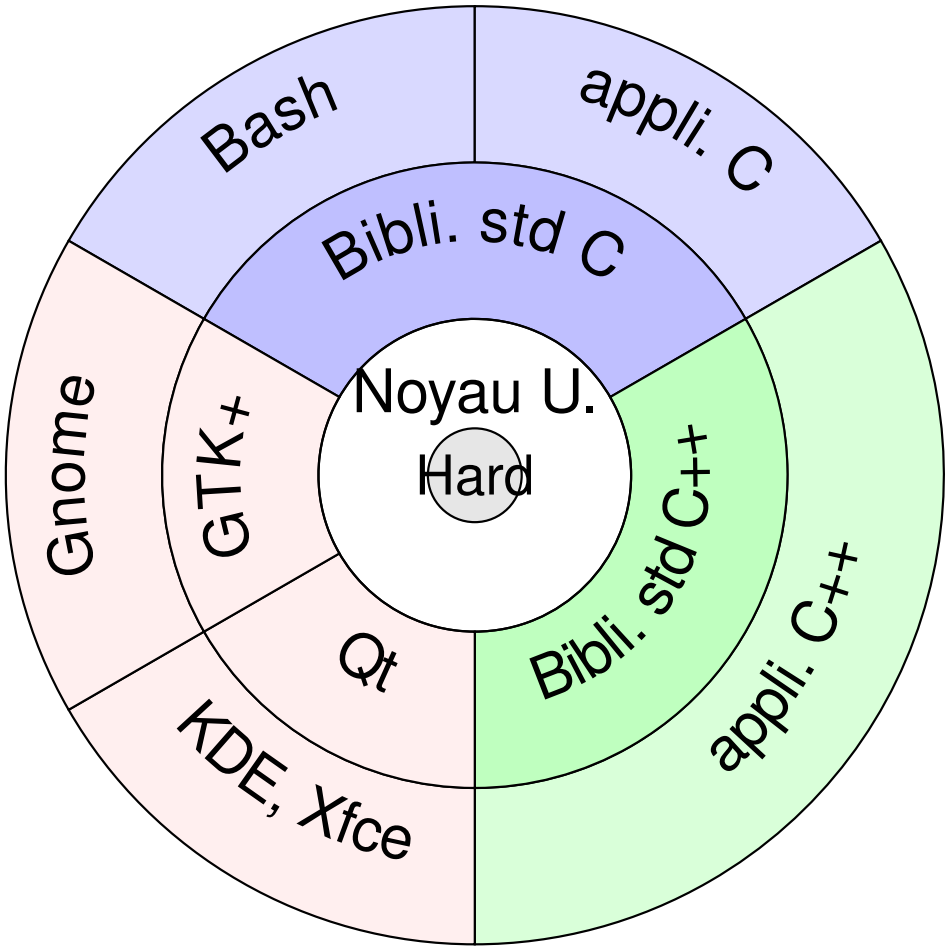
- **les Unix** GNU/LInux, BSD, Unix propriétaires (AIX d'IBM, Solaris de Sun, HP-UX, ...)

Mac OS X, iOS des dérivés d'Unix sur les produits Apple

Android, Google Chrome OS des dérivés basés sur un noyau Linux

Mainframes VM, MVS, OS/400 d'IBM, GCOS de Bull

Architecture en couche d'Unix



- une appli. C utilise la bibliothèque standard C (`printf, stat`, ...) (man 3)
- une appli C++ utilise la bibli std C++ (insertion dans un flot `<<`, les classes `vector, thread`, ...)
- d'autres bibliothèques existent (GUI)
- toute appli peut utiliser les appels noyau (man 2) : `fork, pipe,` ...

Les langages I

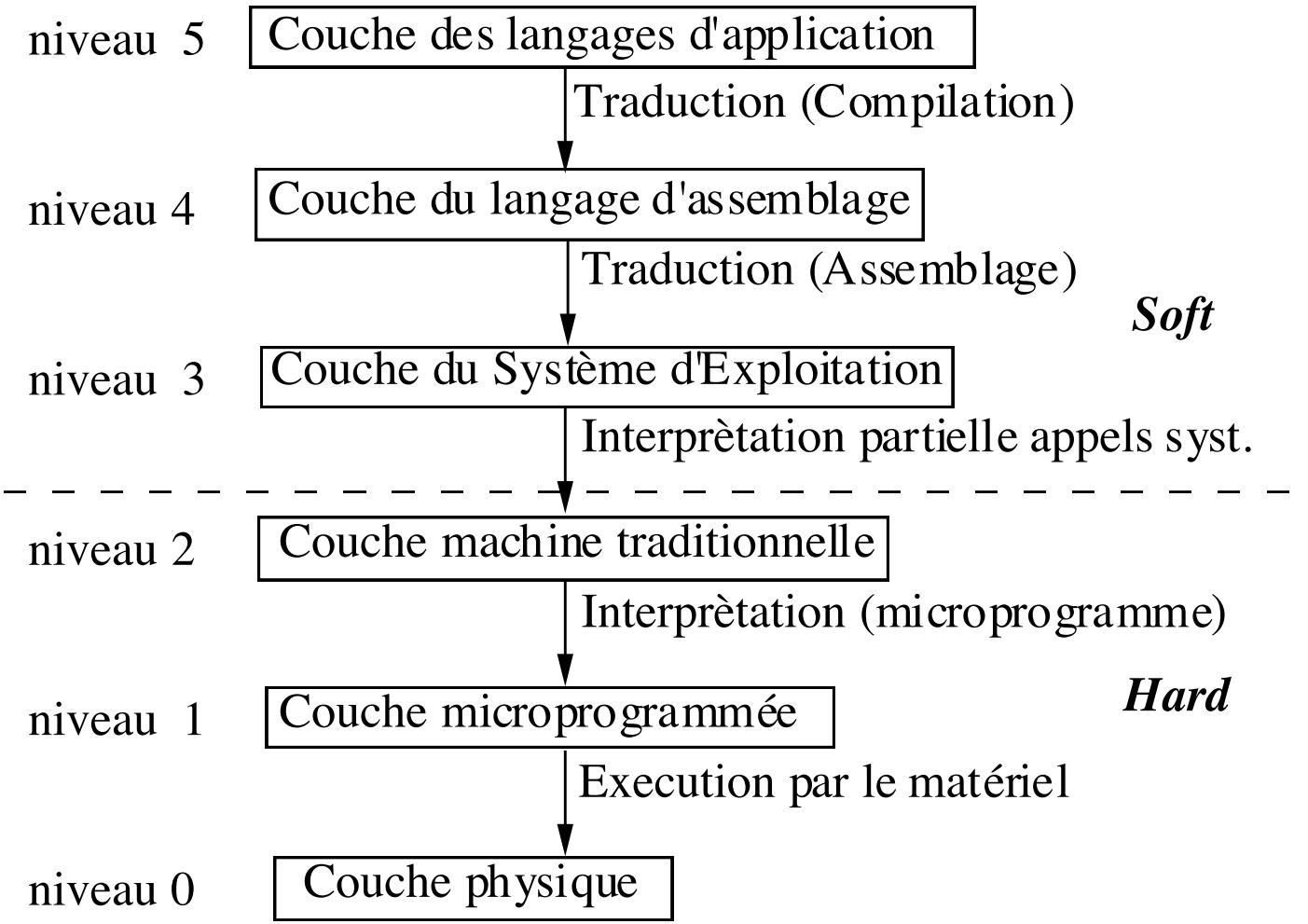
Le jeu d'instructions du processeur est limité et primitif. On construit donc au-dessus, une série de couches logicielles permettant à l'homme un dialogue plus aisé.

Interprétation Vs compilation

Les programmes en Li sont :

- soit traduits (compilés) en L_{i-1} ou L_{i-2} ou $\ldots L_1$,
- soit interprétés par un interpréteur tournant en L_{i-1} ou L_{i-2} ou $\ldots L_1$

Couches et langages I



niveaux composant (électronique) puis atomique (physique solide)

Couches et langages II

Description des couches

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

Matériel et Logiciel I

Hardware

Le matériel est l'ensemble des composants mécaniques et électroniques de la machine : processeur(s), mémoires, périphériques, bus de liaison, alimentation...

Software

Le logiciel est l'ensemble des programmes, de quelque niveau que ce soit, exécutables par une couche de l'ordinateur. Un programme est un mot d'un langage. Le logiciel est immatériel même s'il peut être stocké physiquement sur des supports mémoires.

Matériel et Logiciel II

Matériel et Logiciel sont conceptuellement équivalents

- Toute opération effectuée par logiciel peut l'être directement par matériel et toute instruction exécutée par matériel peut être simulée par logiciel
- Le choix est facteur du coût de réalisation, de la vitesse d'exécution, de la fiabilité requise, de l'évolution prévue (maintenance), du délai de réalisation du matériel
- Dans un langage donné, le programmeur communique avec une machine virtuelle sans se soucier des niveaux inférieurs.

Matériel et Logiciel III

Exemples de répartition matériel/logiciel

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

Objectifs du cours

- Comprendre l'arborescence Unix et savoir la manipuler
- Comprendre le processus de compilation des programmes (C sous Unix) et l'interprétation
- posséder les bases indispensables de la représentation des données en machines afin de comprendre l'utilité de structure de données efficaces
- appréhender les Entrées/Sorties généralisées et leur lien avec un Système de Gestion de Fichier
- maîtriser la gestion des fichiers et des flots C sous Unix

Plan

L'Arborescence Unix et les fichiers I

Mathématiquement :

- une arborescence est un graphe orienté ou :
  - chaque sommet possède un et un seul parent mis à part **un unique sommet appelé racine** qui n'en possède pas
  - chaque sommet est atteignable par un chemin (succession d'arcs et de sommets) **unique** depuis la racine
- les sommets d'une arborescence sont appelés **noeuds** et les noeuds n'ayant pas d'enfant sont appelés **feuilles**
- les arborescences sont utilisées dans plein de domaines (généalogie, organigramme, hiérarchies ...)

Sous Unix (Linux) :

- La racine de l'arborescence est nommée / (slash à ne pas confonter avec anti-slash \)

L'Arborescence Unix et les fichiers II

- les noeuds internes sont des répertoires (ou dossiers) (directory or folder)
- les feuilles sont des fichiers (files)
- **Attention**, sous Unix tout est fichier : les répertoires, les fichiers textes, les binaires exécutables, les périphériques sont des fichiers !
- Pour différencier, les sortes de fichiers, on parle de fichier régulier (regular file) pour tout ce qui n'est pas spécial (répertoire, périphérique, ...)
- dans les fichiers réguliers, on distingue les fichiers **textes qui sont lisibles par l'être humain**) des fichiers **binaires** (que l'humain ne peut comprendre)
- Par exemple, un fichier XML, JSON, C, Python, Java, HTML sont des fichiers textes
- Mais, un fichier mp3, jpg, zip, ou /bin/ls sont des fichiers binaires

Interaction avec le système I

il y a deux modes d'Interaction avec Linux

- Graphical User Interface (GUI) : environnement à fenêtres, menus, boutons cliquables à la souris
- Command Line Interface (CLI) : la ligne de commande d'un terminal graphique (XTerm ou gnome-terminal ou ...) est une application permettant de émuler une **console** 80 colonnes
- **Tout informaticien se doit d'apprendre à travailler en mode CLI !**
- le terminal exécute l'interpréteur de commandes du système d'exploitation (bash sous Linux par défaut)
- Bourne Again SHell est l'un des shell (coquille) d'Unix les plus utilisés
- il permet de faire tout ce qu'on peut réaliser en mode graphique !

bash et l'arborescence Unix I

- une commande bash est constituée d'une suite de mots séparés par des espaces (ou tabulation) : `macmd param1 param2 param3 ...`
- une commande s'exécute dans un **processus** qui vit dans un environnement dont le `working directory` (répertoire de travail) est un élément essentiel
- au début d'une session, votre répertoire de travail est votre répertoire d'accueil (racine de votre arborescence personnelle (vos fichiers))
- il existe un certain nombre de commandes prédéfinies dont certaines sont **internes** au `bash` et d'autres **externes** (`/bin`, ...)
- un `path` (chemin) représente une succession de répertoires séparés par des slash vous permettant de naviguer dans l'arborescence

bash et l'arborescence Unix II

- un chemin absolu débute par / : `/usr/bin/`
- le chemin `../../MonProjet/src` indique qu'il faut remonter 2 fois sur le parent du répertoire courant puis descendre dans le répertoire src qui est lui même dans le répertoire MonProjet
- `.` et `..` dénote du répertoire courant (wd) et de son parent
- 2 commandes internes utiles sur les chemins :
  - `dirname un/deux.txt` qui extrait le nom du répertoire (`un`)
  - `basename un/deux.txt` qui extrait le nom du fichier (`deux.txt`)

Les commandes indispensables I

man cmd	MANuel de la commande cmd (guide d'utilisation)
pwd	Print Working Directory
ls	LiSt directory
cd quelquepart	Change (working) Directory
mkdir nouvRep	crée un nouveau répertoire (MaKe DIRectory)
rmdir monRep	(ReMove) supprime un répertoire s'il est vide ...
mv ancien nouveau	(MoVe) déplace ou renomme un fichier
rm fic	(ReMove)
cp ancien nouveau	CoPy ancien fichier vers nouveau

Plan

Environnement salles machine I

Bien que ce cours soit destiné à un environnement Unix-like tels Linux (distrib. x, y, z), SunOS, FreeBSD, Raspbian ..., Nous allons quand même parler de l'environnement en salle de TP

- le bureau d'accueil posède :
  - un panneau haut contenant un bouton (Menu **Whisker** (cercle rouge)) pour lancer des applications, suivis d'onglets correspondants aux processus en cours
  - un panneau central, le bureau correspondant au répertoire `~/Desktop`
  - un panneau bas personnalisable où vous pourrez ranger des raccourcis vers vos applications favories
- lancer un terminal Xfce en cliquant sur `Whisker/Système/Terminal Xfce`
- Distribution Linux : Ubuntu

Environnement salles machine II

```
$ uname
Linux
$ lsb_release -a
No LSB modules are available.
Distributor ID:         Ubuntu
Description:          Ubuntu 20.04.2 LTS
Release:        20.04
Codename:         focal
```

- Environnement de bureau Xfce (surcouche graphique à X11)

```
$ env|grep SESSION
...
XSESSION_EXEC=xfce4-session
```

- répertoire d'accueil

```
$ pwd
/home/p00000010501
```

Créer un fichier texte I

plusieurs solutions :

- commande echo et redirection sortie standard (>)

```
$ echo "Maître Renard sur un arbre perché" >
 ↪  fable.txt
$ ls
... fable.txt ...
$ cat fa<TAB> # pour auto-complétion
Maître Renard sur un arbre perché
```

- avec un éditeur de texte VSCodium :

```
$ vscodium ~
... File/new File/ ... File/Save
```

- avec le filtre neutre `cat` et une redirection :

```
$ cat > fable.txt
Maître Renard sur un arbre perché<CTRL-d>
```

Créer un script bash I

- un script est un programme en langage interprété stocké dans un fichier texte
- un script bash contient une suite de commandes
- pour l'exécuter, il faut lui donner le droit d'eXécution (par défaut les fichiers créés n'ont pas ce droit (voir fable ;)))

```
# dans VSCodium, taper et sauver la suite
echo "Hello World !"
pwd
ls
# sauver dans mon1erscript et revenir au terminal
$ chmod u+x mon1er<TAB>
$ bash mon1erscript
Hello World !
/home/p00000010501
 angularTournoi IntroSyst sketchbook autosave ...
```

Créer un script bash II

- Remarquons que l'on peut simplifier le lancement de la commande en indiquant l'interpréteur dans la première ligne en commentaire

```
#!/bin/bash
echo "hello"; pwd; ls

$ ./mon1erscript
```

- Bien entendu le langage Bash est un langage évolué qui permet de faire bien plus qu'une séquence (alternative, répétitive, fonctions ...)

Principaux répertoires d'un système Unix I

- `/bin, /usr/bin, /usr/local/bin` commandes du système, compilateurs (gcc), applications (firefox). Ce sont des **binaires exécutables**
- `/home` racine des répertoires d'accueil des utilisateurs
- `/dev` périphériques blocs et caractères du système
- `/etc` fichiers de configuration (`password, bahrc, apache2/, ...`)
- `/usr/include/` entêtes standard du C (`stdio.h, ctype.h, ...`)
- `/sbin` commandes réservées à l'administrateur (`root`)
- `/lib, /usr/lib` bibliothèques (libraries)
- `/proc` contient des infos sur les processus en cours d'exécution

Principaux répertoires d'un système Unix II

- `/mnt, /automnt` point de **montage** de systèmes de fichiers (clé USB)
- `/var/log, /var/spool` données fréquemment réécrites par le système

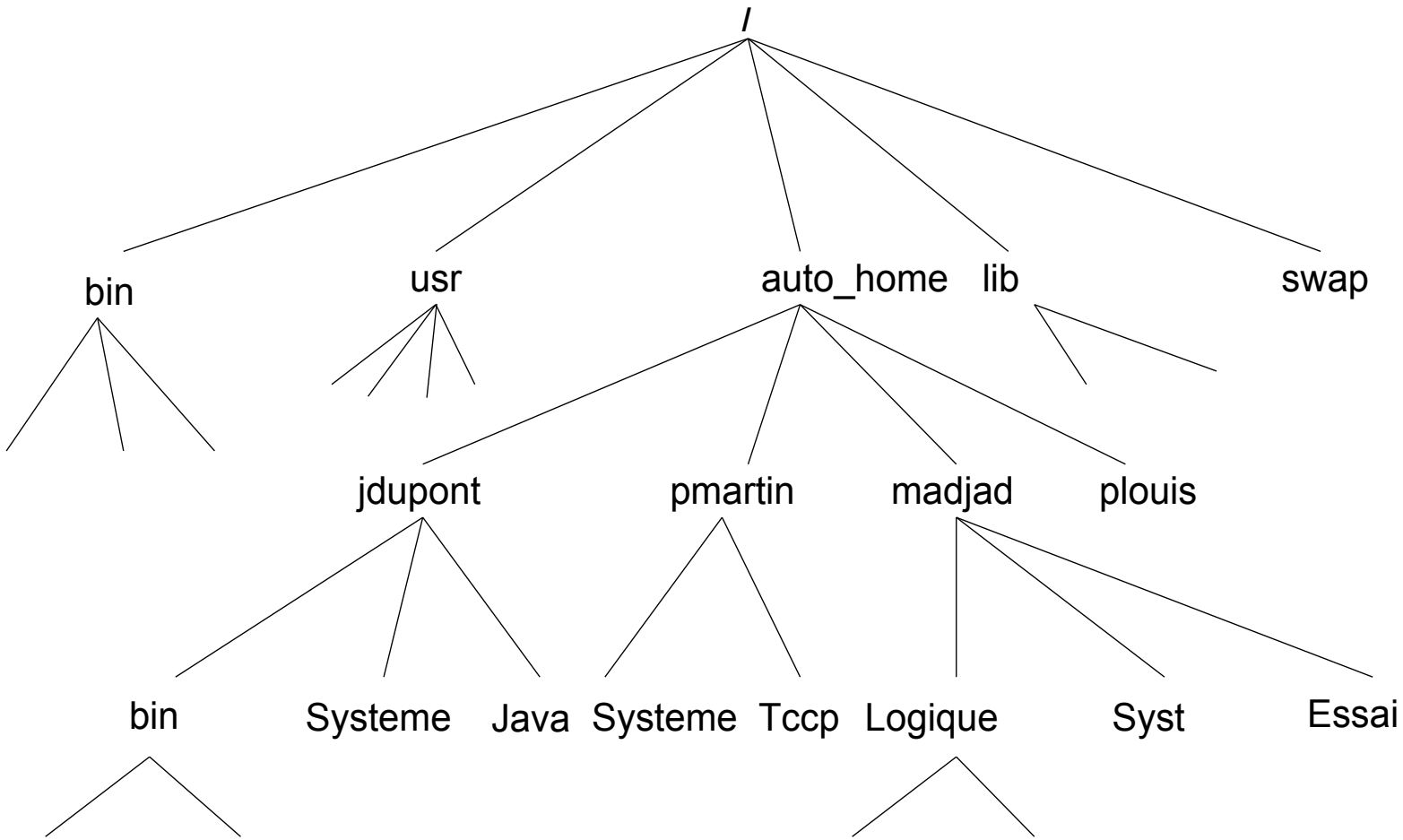
Plan

Plan

Objectifs

- Gérer l'espace disque, répondre aux demandes d'allocations et de libération d'espace ;
- Donner à l'utilisateur une vision **arborescente** simple ;
- Utiliser efficacement l'espace disque en place et en temps ;
- Assurer la sécurité des données grâce à un système de droits ;
- Gérer efficacement petits et gros fichiers ;
- Permettre l'utilisation de différents systèmes de fichiers (FAT32, NTFS, E4FS, NFS ...) greffés dans l'arborescence.

Vision utilisateur



L'utilisateur face au système

L'utilisateur veut des fichiers :

- accessibles grâce à un nom (au moins 1),
- organisés de sorte à les retrouver "facilement",
- sur lesquels il a le droit de propriété absolu et le droit de laisser faire certaines actions aux autres utilisateurs,
- copiables, déplaçables, renommables ...

Problèmes et réponses du système Unix

Le système doit gérer un certains nombres de problèmes :

- concurrence des utilisateurs sur un même espace disque
- allocation, désallocation des blocs et gestion du chaînage des blocs constituant un fichier
- minimisation de l'espace d'administration des fichiers (table des **inodes**, table des blocs dispo., ...)
- solution générale des OS : une ou plusieurs arborescences de répertoires et de fichiers
- identification interne des fichiers par un numéro d'inode
- gestion des droits simple rwx (Read Write eXecute)
- gestion de systèmes de fichiers non Unix

Plan

Utilisateur Unix I

- Un utilisateur Unix est identifié par un `uid` et possède un nom `uname`
- La commande `whoami` permet d'afficher le `uname`

```
$ whoami
michel.meynard@umontpellier.fr
```

- Un utilisateur est membre de plusieurs groupes d'utilisateurs
- A un instant donné, un utilisateur appartient à 1 groupe **primaire** qui est par défaut le premier groupe de sa liste de groupes
- Lorsqu'un utilisateur crée un fichier, celui-ci appartient à son groupe primaire
- Un utilisateur peut changer de groupe primaire grâce à la commande : `newgrp monAutreGroupe`
- Chaque groupe possède un identifiant `gid` et un nom `gname`

Utilisateur Unix II

- La commande `id` permet de connaitre l'identité complète de l'utilisateur courant :

```
$ id
uid=77007(michel.meynard@umontpellier.fr)
 ↪  gid=77007(p00000010501)
 ↪  groupes=77007(p00000010501),1029(audio),1050(ENS_To
```

- Sur Mac OSX (autre Unix BSD), on obtiendra des informations similaires :

```
$ id
uid=501(michel) gid=20(staff)
 ↪  groups=20(staff),12(everyone),61(localaccounts),79(
```

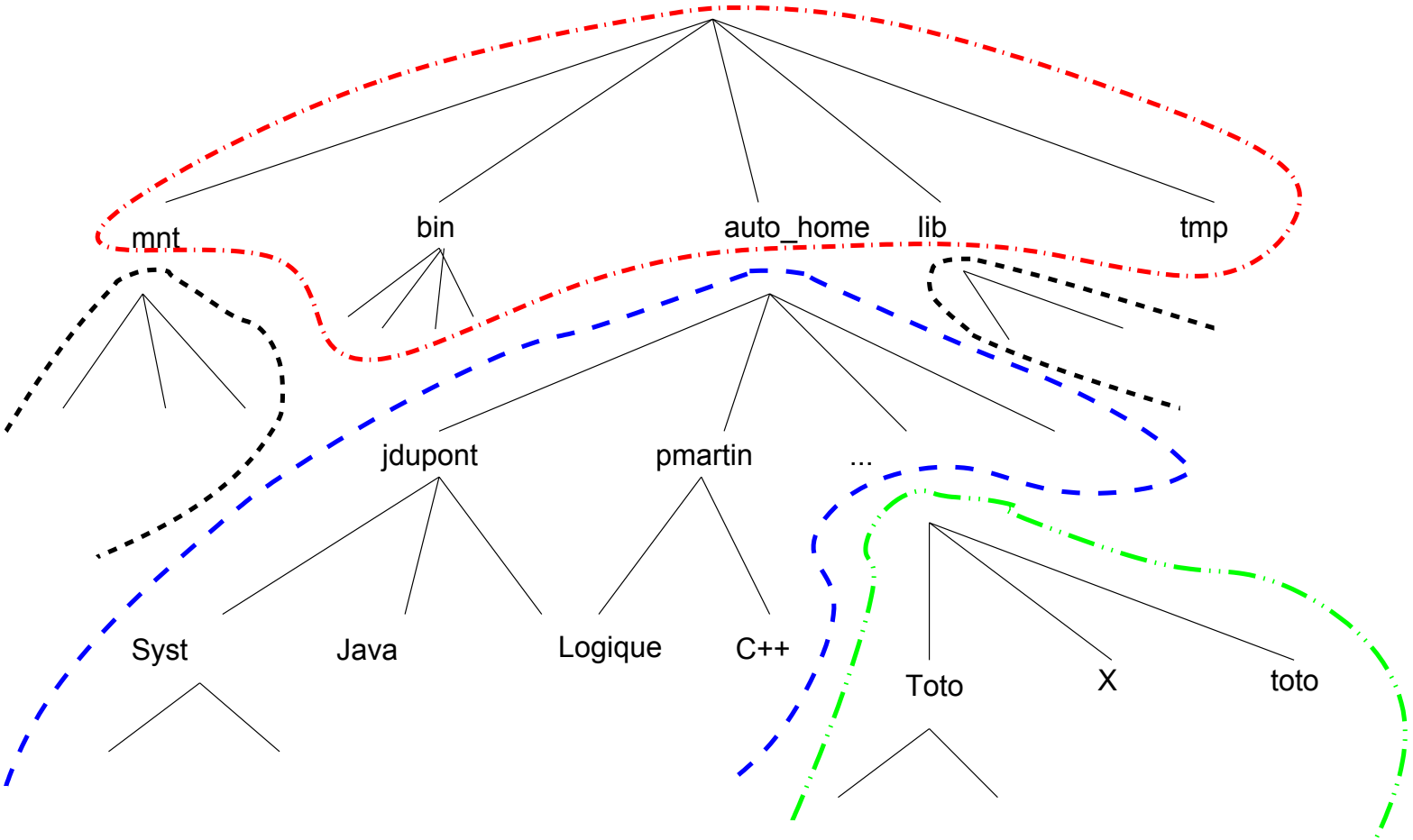
- Sous Linux, le groupe primaire d'un utilisateur est un groupe privé ne contenant que lui-même !
- Ces informations sont disponibles dans les fichiers de configuration `/etc/passwd`, `/etc/group`, `/etc/sudoers`

Utilisateur Unix III

- L'utilisateur `root (id=0)` est l'administrateur de la machine et à tous les droits
- Même quand on administre sa propre machine, il ne faut pas se logger en tant que root mais plutôt utiliser la commande `sudo` pour administrer sa machine ...
- Dans les salles machines de l'UM, les utilisateurs ne sont pas locaux à chaque machine (pas plus que leur espace disque) mais sont recensés dans un répertoire `LDAP` ...

Plan

Partitions et sous-arbres



Partition et Système de fichier

- un disque dur est généralement partitionné en plusieurs partitions
- au début du disque dur, résident le *Master Boot Record* et la table des partitions
- chaque partition est formattée selon un type de système de fichier
- différents *File System* existent : e4fs (Linux), NTFS (Windows), FAT32 (DOS Windows), APFS (Apple), ...
- chaque partition peut posséder un secteur de boot (secondaire) dans le cas ou la partition est bootable
- l'unité d'allocation dans un système de fichier est le **bloc**

Structure d'un système de fichier Unix

Un système de fichier linux (e3fs ou e4fs) se compose de plusieurs parties :

<table>
<tr><td rowspan="2">gestion
3%</td><td>Table des inodes</td></tr>
<tr><td>Table d'allocation</td></tr>
<tr><td>97%
données</td><td>Blocs de fichiers</td></tr>
</table>

Espace de gestion
- éventuellement secteur de lancement (*bootstrap*).
- une table des *inodes* (ou *i-nœuds*) : métadonnées des fichiers et localisation des blocs
- une table d'allocation permettant de connaître les blocs libres

Espace des données contient les contenus des fichiers et quelques blocs d'indirection pointés par des inoeuds

Plan

Table des inodes

num.	type	droits	liens	prop.	grp	taille	dates	pointeurs

num. numéro d'inode

type type de l'inode : fichier régulier (-), répertoire (d), ...

droits droits habituels rwx pour utilisateur, groupe, autres

liens compteur de liens durs ou nb de sous-répertoires

prop. identité (numéro) du propriétaire

grp identité du groupe

taille taille du fichier en nombre d'octets

pointeurs numéros des blocs dans l'espace des données

dates création, modification, accès

Exemple de représentation des Fichiers

num.	type	droits	liens	prop. ; grp	taille	dates	pointeurs
1450	–	`rwxr-xr-x`	1	470 ; 47001	125	*	...

Répertoires

/home

num.	nom
1450	hello.c

Exemple de représentation des Fichiers et Répertoires

num.	type	droits	liens	prop. ; grp	taille	dates	pointeurs
1450	-	rwxr-xr-x	1	470 ; 47001	125	*	...
795	d	rwxr-x--	2	470 ; 47001	2048	*	...

Répertoires

/home

num.	nom
795	•
1450	hello.c

Exemple de représentation des Fichiers et Répertoires

num.	type	droits	liens	prop. ; grp	taille	dates	pointeurs
1450	-	rwxr-xr-x	1	470 ; 47001	125	*	...
795	d	rwxr-x--	2	470 ; 47001	2048	*	...
2	d	rwxr-xr-x	2	0 ; 0	2048	*	...

Répertoires

/home

num.	nom
2	••
795	•
1450	hello.c

/ (racine)

num.	nom
2	•

Exemple de représentation des Fichiers et Répertoires

num.	type	droits	liens	prop. ; grp	taille	dates	pointeurs
1450	-	rwxr-xr-x	1	470 ; 47001	125	*	...
795	d	rwxr-x--	2	470 ; 47001	2048	*	...
2	d	rwxr-xr-x	2	0 ; 0	2048	*	...

Répertoires

/home

num.	nom
2	••
795	•
1450	hello.c

/ (racine)

num.	nom
2	•
795	home
7654	usr

Exemple de représentation des Fichiers et Répertoires

num.	type	droits	liens	prop. ; grp	taille	dates	pointeurs
1450	-	rwxr-xr-x	1	470 ; 47001	125	*	...
795	d	rwxr-x--	2	470 ; 47001	2048	*	...
2	d	rwxr-xr-x	2	0 ; 0	2048	*	...

Répertoires

/home

num.	nom
2	••
795	•
1450	hello.c

/ (racine)

num.	nom
2	••
2	•
795	home
7654	usr

Plan

Un système simple de droits

- Axiome : tout accès à un fichier doit être réalisé après ouverture (*open*) d'un **chemin** (path) absolu ou relatif
- l'ouverture vérifie l'accessibilité de l'utilisateur du processus à chaque répertoire traversé et au fichier final
- 3 types de droits : r(ead), w(rite), x(eXecute)
- 3 catégories d'utilisateur : u(owner), g(roup), o(ther)
- tout répertoire étant un fichier, les droits signifient :
  - r lecture du contenu du répertoire (ls)
  - w écriture dans le répertoire c-à-d création (*creat*), suppression (*rm*), renommage (*mv*) de fichier
  - x traversée du répertoire : un fichier lisible par tous, situé dans un répertoire non traversable, ne pourra être lu
- rappel : à tout processus est associé un répertoire courant (*getcwd()*), un utilisateur (*getuid()*) et un groupe (*getgid()*)

Plan

Structure d'un Répertoire

- Un répertoire contient une liste de couples (entrées) n^o *inode* $\Leftrightarrow$ *nom* où chaque nom est unique
- la **longueur des noms des fichiers** est variable, avec une limite administrable, fixée par le système, par exemple 256 octets
- les éléments autres que le *numéro d'inode* et le *nom*, inclus dans un répertoire, permettent la **gestion de la liste**, par exemple la longueur de l'élément courant
- tout répertoire contient au moins deux entrées • et • • afin de permettre la navigation relative au répertoire courant

Structure d'un Répertoire

- Un répertoire contient une liste de couples (entrées) n^o *inode* $\Leftrightarrow$ *nom* où chaque nom est unique
- la **longueur des noms des fichiers** est variable, avec une limite administrable, fixée par le système, par exemple 256 octets
- les éléments autres que le *numéro d'inode* et le *nom*, inclus dans un répertoire, permettent la **gestion de la liste**, par exemple la longueur de l'élément courant
- tout répertoire contient au moins deux entrées • et • • afin de permettre la navigation relative au répertoire courant
- la taille d'un répertoire est gérée autrement que celle d'un fichier (multiple de la taille d'un bloc)

Structure d'un Répertoire

- Un répertoire contient une liste de couples (entrées) n^o *inode* $\Leftrightarrow$ *nom* où chaque nom est unique
- la **longueur des noms des fichiers** est variable, avec une limite administrable, fixée par le système, par exemple 256 octets
- les éléments autres que le *numéro d'inode* et le *nom*, inclus dans un répertoire, permettent la **gestion de la liste**, par exemple la longueur de l'élément courant
- tout répertoire contient au moins deux entrées • et • • afin de permettre la navigation relative au répertoire courant
- la taille d'un répertoire est gérée autrement que celle d'un fichier (multiple de la taille d'un bloc)
- les opérations sur les répertoires sont soit des appels systèmes (`mkdir()`, `rmdir()`) soit des fonctions de bibliothèque (`opendir()`, `readdir()`, `closedir()`)
- on ne peut admettre qu'un utilisateur puisse modifier directement (`open`, `write`) un répertoire ; le SF pourrait être corrompu.

Plan

umask le masque de création par défaut I

Lors de la création de fichiers ou de répertoires par le propriétaire, des droits par défaut sont affectés grâce au `umask` :

- la commande `umask`, permet de voir et de modifier ce masque

```
$ umask
0022
$ umask 02
$ umask
0002
```

- le masque indique (en octal) les droits qui ne seront pas affectés au fichiers créés
- il utilise le complément à 1 pour les répertoires et le complément à 2 pour les fichiers réguliers !
- Avec un umask à 022 :
  - les fichiers n'auront ni le droit d'écriture, ni le droit d'exécution pour le groupe et les autres

umask le masque de création par défaut II

- les répertoires n'auront pas le droit d'écriture pour le groupe et les autres

- bien entendu, `chmod` permet ensuite de faire ce que l'on veut !

Plan

Notion de lien dur (*hard link*)

- chaque entrée dans un répertoire référence un élément de la table des inodes, c'est à dire un fichier
- plusieurs entrées du même répertoire ou dans différents répertoires peuvent référencer le même inode : `ln hello.c hello2.c`
- les deux noms de fichiers sont des liens durs équivalents qui pointent sur le même contenu et les mêmes métadonnées (inode)
- l'accès en écriture sur cet unique fichier peut être réalisé via l'un ou l'autre
- l'usage historique des liens était le partage de fichiers communs à un groupe de développeurs :
  `ln commun.h ~jdupont/commun.h`

Exemple de liens durs (*hard link*)

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

Répertoires

tp5

num.	nom
1450	commun.h

ProjetX

num.	nom
1450	commun.h

On mémorise dans l'inode le **nombre** de liens !

Exemple de liens durs (*hard link*)

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

Répertoires

tp5

num.	nom
1450	commun.h

ProjetX

num.	nom
1450	commun.h

On mémorise dans l'inode le **nombre** de liens !

Lien Dur

Un *lien dur* ou *physique* est différent d'un *lien symbolique* ou *raccourci* qu'on étudiera plus tard.

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

Caractéristiques :

- Un lien dur est forcément dans une **même partition** (SF) ; on dit qu'un lien dur ne peut *traverser* les SFs.
- On ne peut créer un lien dur sur un **répertoire** : cela supprimerait la structure arborescente (graphe avec circuits)
- Noter qu'il y a un seul inode, donc un seul propriétaire, un seul contenu, un seul ensemble de droits, . . . **MAIS** des utilisateurs différents devant appartenir au groupe du fichier afin de pouvoir lire/modifier le contenu

Problèmes Induits par les Liens Durs

Une correction à faire : La suppression d'une référence (nom de fichier, lien dur) ne supprimera pas forcément l'inode et tout le contenu du fichier.

Principe de la solution :

- à chaque création d'un lien incrémenter le nombre de liens dans la table des inodes
- le décrémenter à chaque suppression
- ne supprimer l'inode et le contenu (désallocation des blocs) que lorsque le nombre de liens est nul

Remarque : la commande de suppression de fichier (`rm`) est implémentée par l'appel système `unlink()` qui supprime une entrée de répertoire et peut avoir un effet de bord

Algorithme de Suppression

Algorithme 2 : supprimer(fichier)

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

sinon
- afficher suppression impossible

Exemple de Suppression

Situation de départ

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

Répertoires

tp5

num.	nom
1450	commun.h

tp8

num.	nom
1450	commun.h

Exemple de Suppression

`rm tp5/commun.h`

num.	type	droits	liens	prop.	taille	dates	pointeurs
1450	–	`rwxr-xr-x`	1	470 ; 47001	125	*	8003475

Répertoires

tp5

num.	nom

tp8

num.	nom
1450	commun.h

Exemple de Suppression

`rm tp8/commun.h`

num.	type	droits	liens	prop.	taille	dates	pointeurs

Répertoires

tp5

num.	nom

tp8

num.	nom

Avantages et Manques

Avantages

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

Manques

- la limite au SF est très réductrice
- pas de référence à un répertoire
- les éditeurs de texte tels emacs enregistrent en renommant l'ancienne version en `commun.h~` et crèent un nouveau fichier (inode) `commun.h`

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

Plan

Le Problème de la Taille

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

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

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

Blocs Directs et Indirects

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

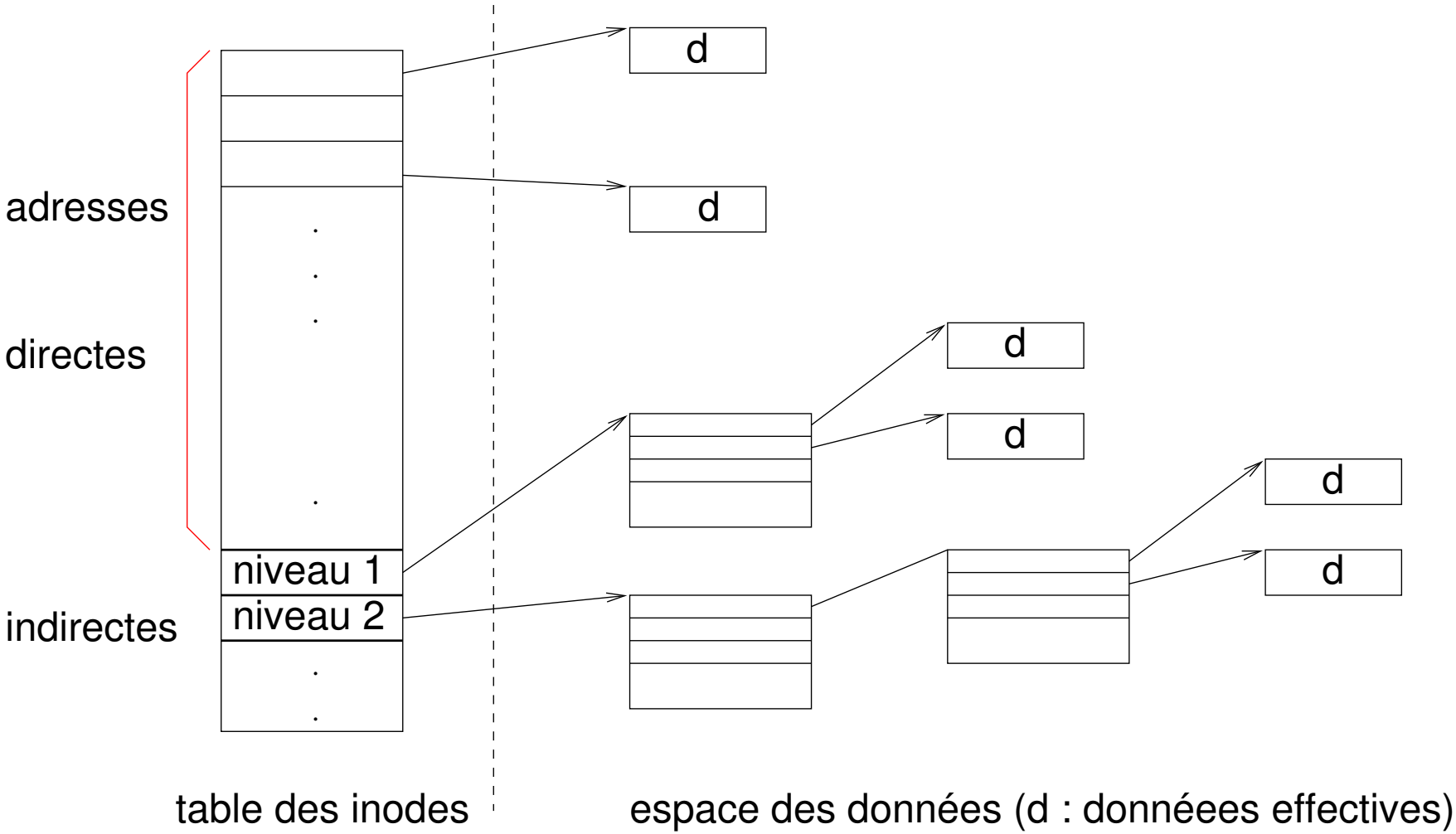
Méthode :
Adressage de tout fichier : un arbre dont les feuilles sont les blocs de données et les nœuds internes des blocs d'adresses.

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

Méthode en Détails

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

Arbre d'Allocation



Exemple - Taille Maximale d'un Fichier

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

Données de base : on suppose que

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

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

Taille Maximale d'un Fichier - suite

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

La taille maximale d'un fichier possible par **l'adressage des blocs** est de : $40Kio + 4Mio + 4Gio + 4Tio$.
On peut donc approximer cette taille à $4Tio$ ce qui représente $1/4$ des adresses de blocs disponibles avec 32 bits d'adresse !

Exercices

1. Si la taille des blocs de données est doublée (à $8Kio$) calculer le facteur de multiplication de la taille maximale d'un fichier (approximation) : $*16$ donc $64Tio$
2. Avec des blocs de $4Kio$, sur quelle longueur faudrait-il coder la taille du fichier dans la table des inodes, afin de satisfaire la taille atteinte par l'adressage des blocs ? 42 bits
3. On considère que les blocs non terminaux (ceux contenant des adresses de blocs) sont "volés" à l'espace des données. Combien de blocs sont ainsi subtilisés avec les blocs de $4Kio$ et un fichier de taille $4Gio$? $4 * 2^{30}/4 * 2^{10} = 2^{20} adrs$ de 4 octets, soit $2^{22} oct = 4Mo$, soit $2^{10} blocs$.

Plus difficile :

4. Situation de départ : blocs de $4Kio$, taille du fichier codée sur 32 bits. On veut agrandir la capacité maximale d'un fichier et passer à $32Gio$. Que peut-on proposer ? taille sur 35 bits donc 40 bits=5 oct

En résumé : des calculs à optimiser

Lors du formattage du SF, il faut prendre en compte :

- le codage de la taille du fichier dans la table des inodes
- l'espace physique réel disponible sur la partition disque
- la taille de chaque adresse de bloc
- la taille de chaque bloc
- la taille de la table des inodes (nb de fichiers)

Exemple :

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

Plan

Lien Symbolique ou raccourci

Objectif : dépasser les limites des liens durs :

- limitation à une même partition (SF)
- impossibilité de pointer sur un répertoire
- basé sur numéro de inode et pas sur un chemin symbolique

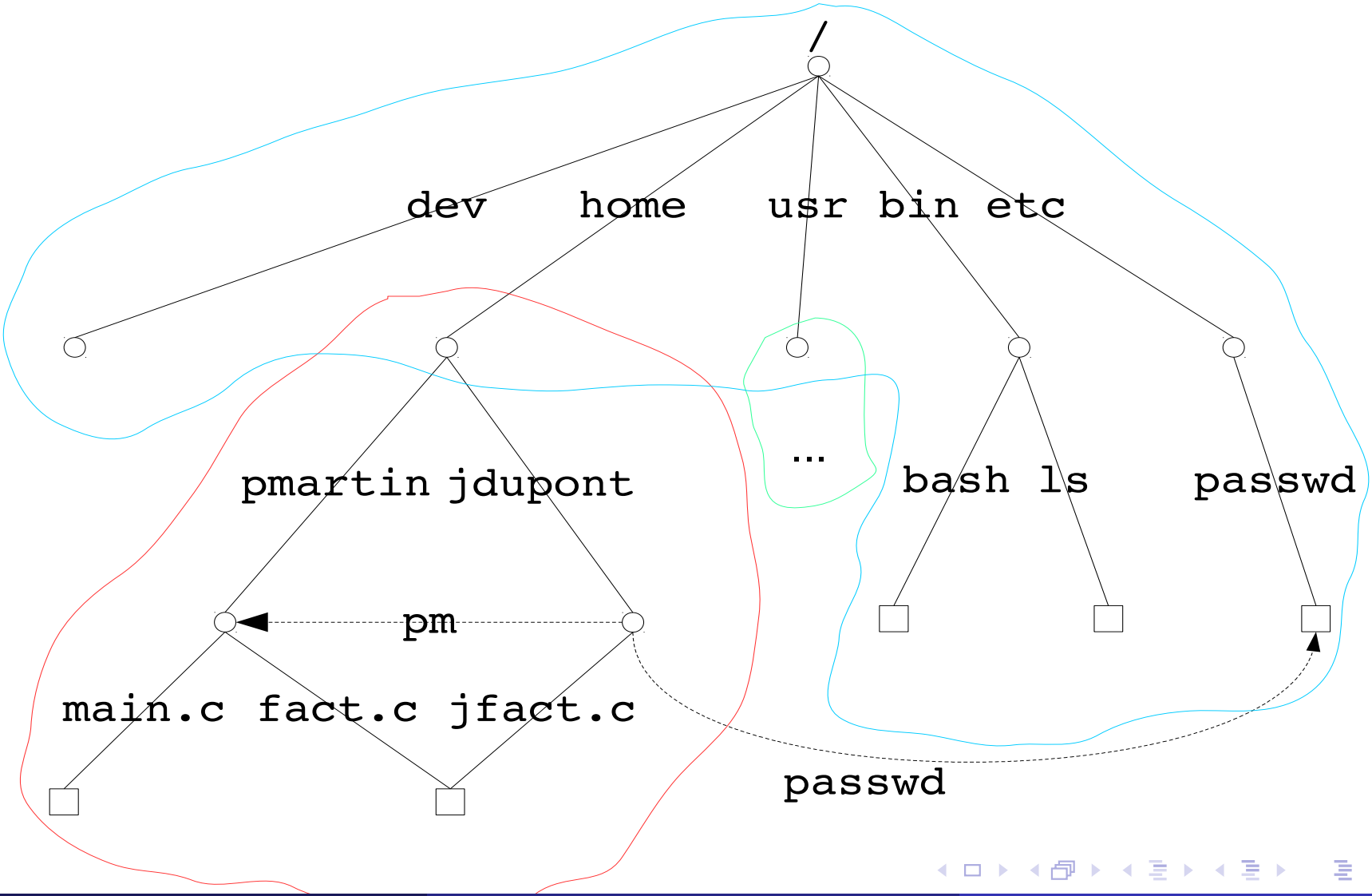
Méthode : un fichier de type nouveau (l) permettant de désigner (référencer, pointer sur) un *chemin* déjà existant dans l'arborescence

Exemples :

- `ln -s /etc/passwd ./passwd`
- `ln -s /home/pmartin/ pm`
- `ln -s /home/pmartin/main.c ./monmain.c`

Point de vue du système : toute action demandée sur le raccourci agira sur le fichier lié, sauf quelques exceptions (lstat)

Exemples de Lien Symbolique



Inode lien symbolique

Table des inodes

num.	type	droits	liens	prop.	taille	dates	pointeurs
3232	`l`	`rwxr-xr-x`			11		99999

Attention :

- noter le type `l` : ni fichier régulier (-), ni répertoire (d), mais lien symbolique (l)
- noter la taille 11 : exactement celle de la chaîne de caractères `/etc/passwd` (contenu dans le bloc 99999)

Caractéristiques des Liens Symboliques

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

Problèmes

- un lien symbolique vers un répertoire ascendant crèe un circuit qu'il ne faut pas prendre dans un parcours récursif (`find`) : `ln -s .. c`
- circuit : `touch a; ln -s a b; rm a;ln -s b a; cd a` provoque une erreur après un certain nombre de résolution de raccourcis !
- lien mort : un raccourci peut référencer un chemin qui n'existe plus ; une vérification d'existence est parfois réalisée à la création (linux)

Commandes et fonctions concernant les Liens Symboliques

- Commande : `ln -s [ancien] [raccourci]`
- Appel système : *symlink()*.
- **lien dur** commande : `ln [ancien] [nouveau]`, appel système : *link()*
- `stat("raccourci", ...` accèdera au fichier référencé tandis que `lstat` accèdera au raccourci
- `open("raccourci", ...` ouvrira le fichier référencé donc `cat raccourci` ou `emacs raccourci` aussi
- `rm raccourci` supprimera le raccourci,
- `cd raccourci` concaténera le mot "/raccourci" au répertoire courant, ce qui peut être bizarre : `ln -s .. c; cd c; pwd; cd ..` a comme effet de revenir dans le répertoire de départ !
- les droits et le propriétaire sont ceux relatifs au raccourci (`chmod, chown, chgrp`)

Plan

Plan

schéma général système

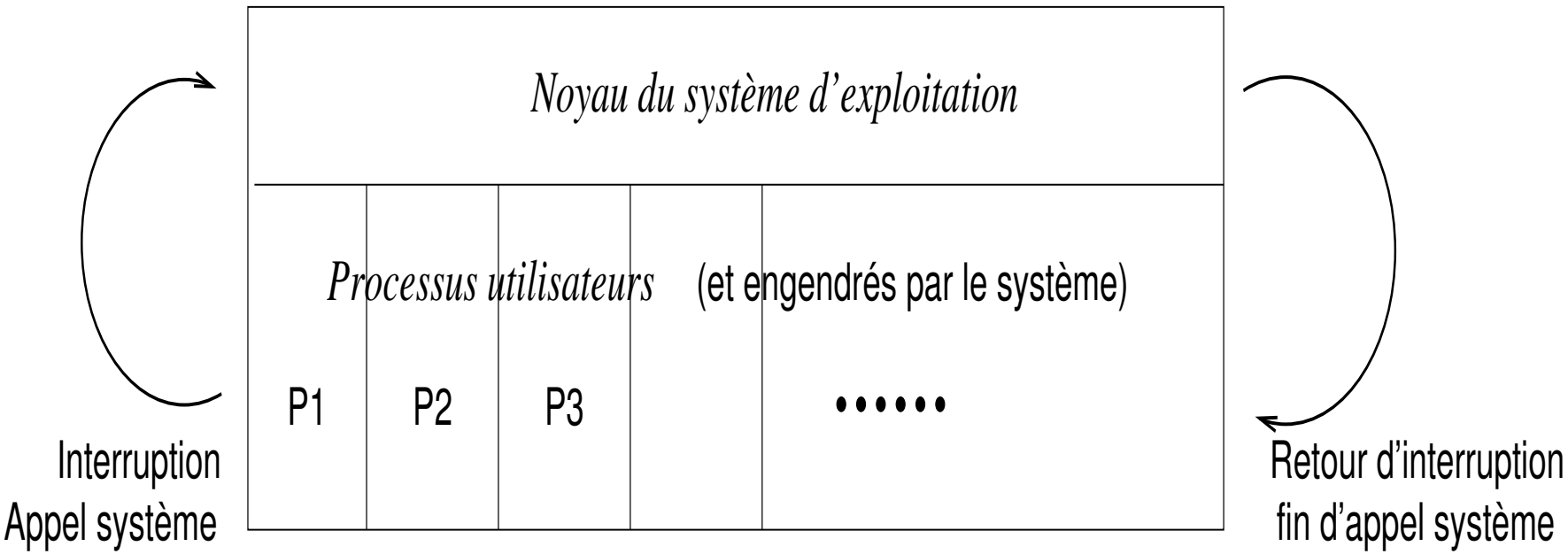


Table des processus

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

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

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

Plan

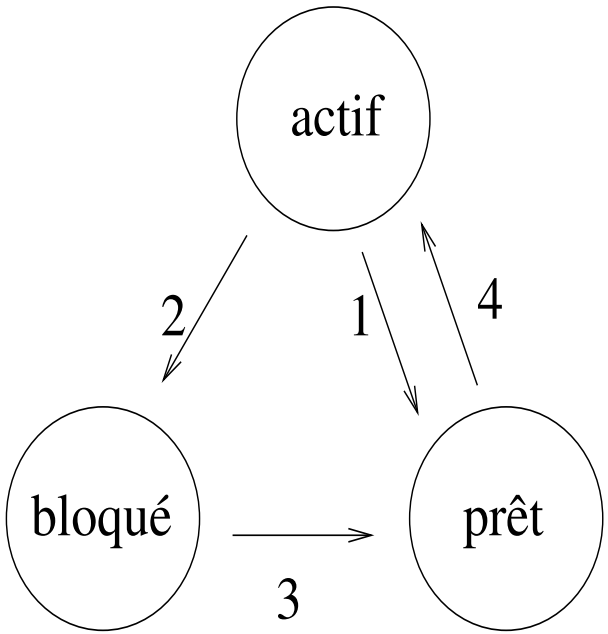
Entrées/Sorties (I/O) d'un processus I

- Un processus peut manipuler des fichiers réguliers selon plusieurs modes :
  - lecture (read)
  - écriture (write)
  - lecture et écriture, ajout (append), ...
- Pour cela, il possède une table des fichiers ouverts
- Un processus lancé par un shell hérite des fichiers précédemment ouverts par son parent
- Par défaut, un bash possède 3 fichiers ouverts :
  - **stdin** : l'entrée standard (clavier en lecture)
  - **stdout** : la sortie standard (l'écran ou la fenêtre en écriture)
  - **stderr** : la sortie d'erreur standard (l'écran ou la fenêtre en écriture)
- Ainsi, la commande `ls` lancée depuis un terminal affichera la liste des fichiers dans le même terminal.

Plan

États d'un processus

On commence par les états de base :



actif tient la ressource UC

prêt seule la ressource UC lui manque

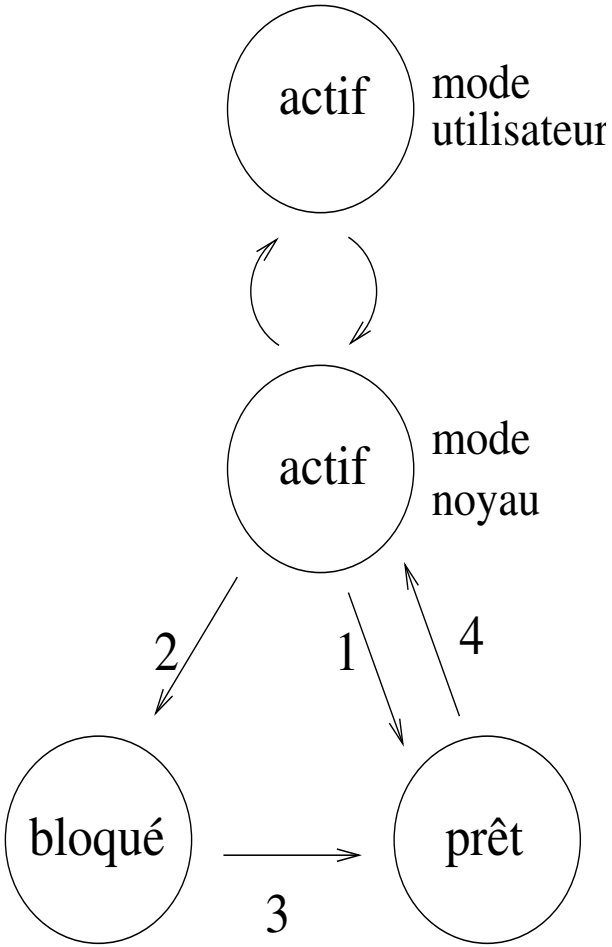
bloqué manque au moins une autre ressource

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

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

Un peu plus

On complète (un peu) ces états de base :



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

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

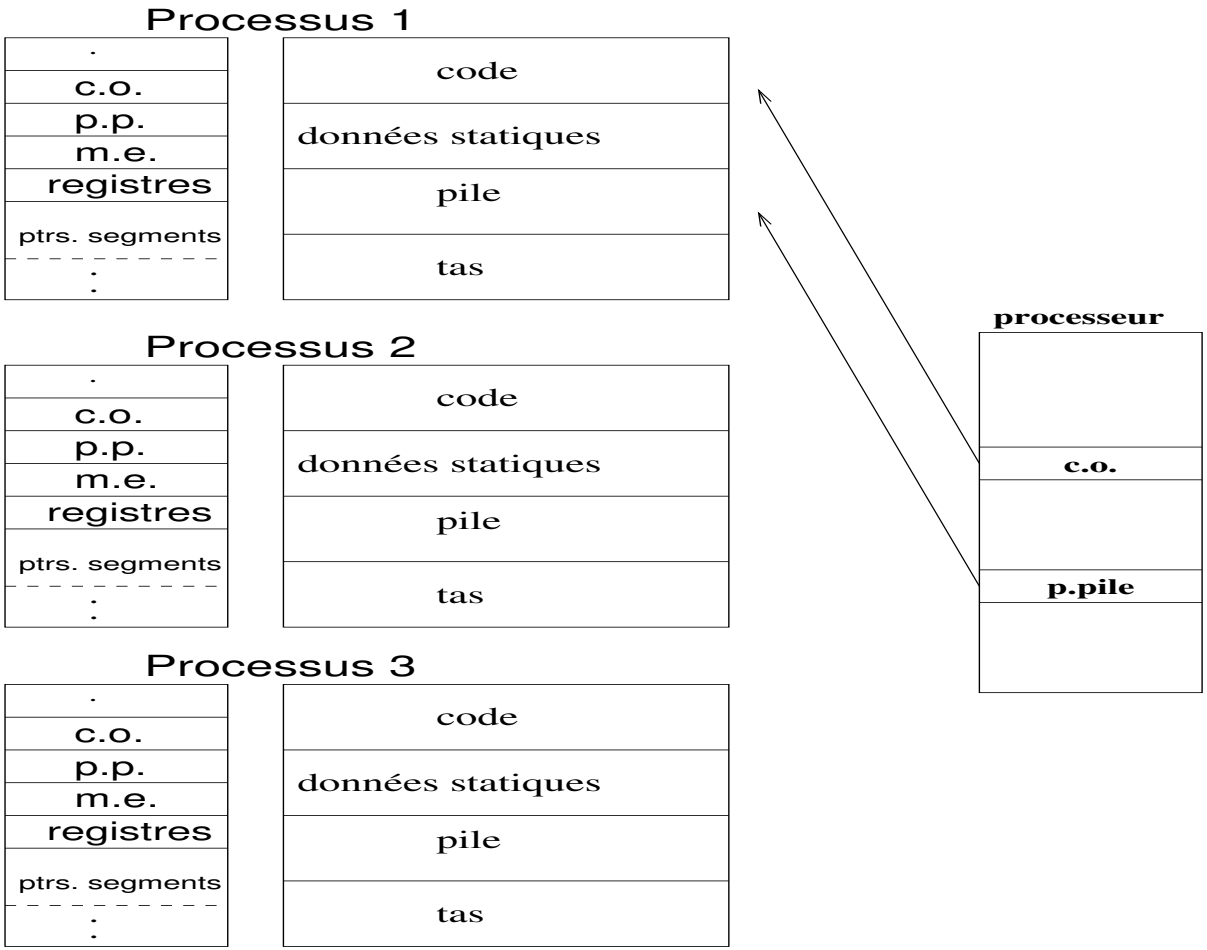
Plan

Principe du changement de contexte

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

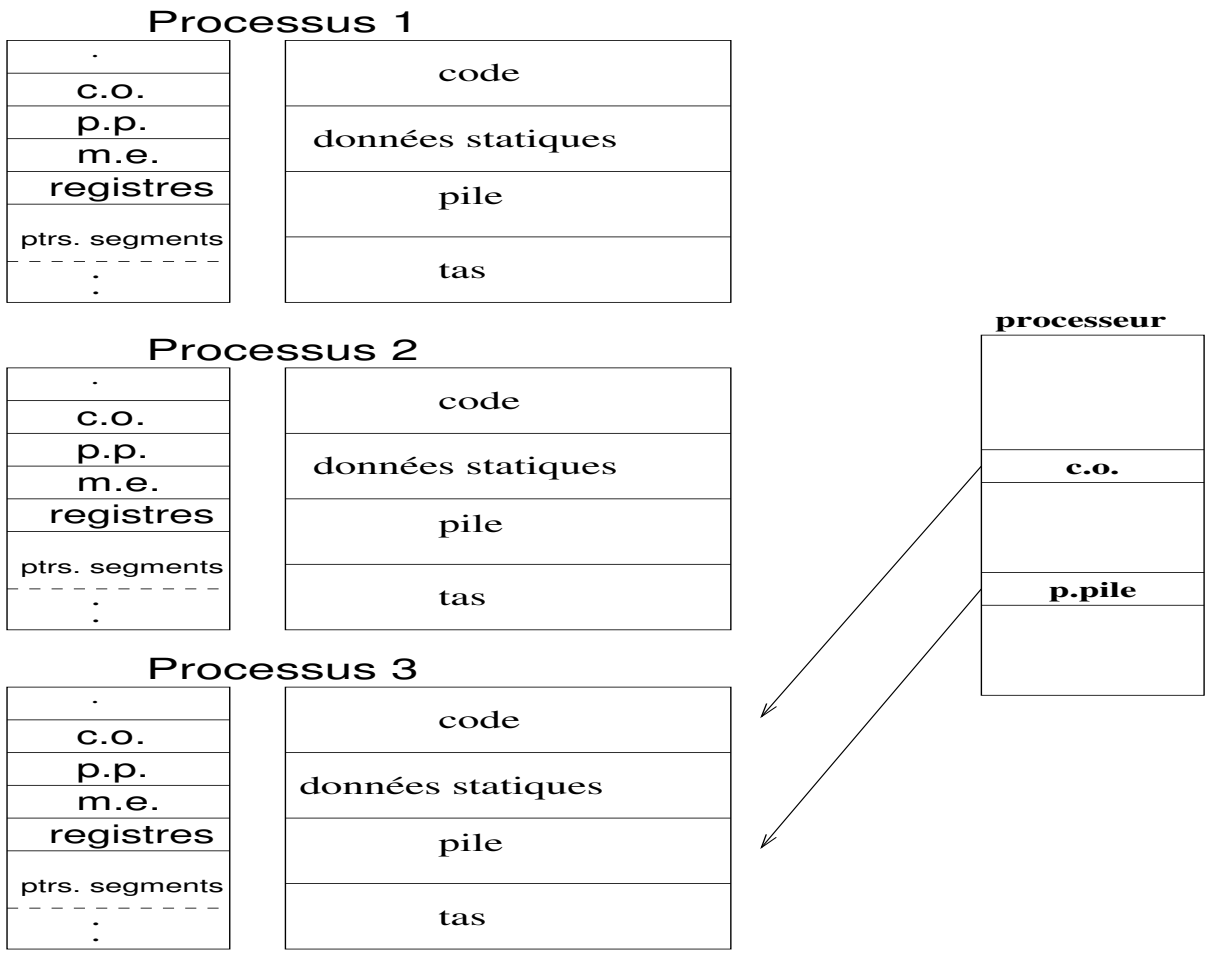
On dit que le système effectue un *changement de contexte*, ou un *basculement de contexte* (*context switch*).
Ce changement se produit lorsque le processus actif passe à l'état bloqué ou prêt et qu'un autre processus devient actif.

Un processus actif



Un autre processus actif

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



Réalisation des changements de contextes

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

Déroulement d'un appel noyau

On peut décrire le déroulement d'un appel noyau :

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

Interrogation orale

Questions :

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

Rôle de l'ordonnanceur

Règles à respecter :

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

Schéma algorithmique ordonnanceur

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

Plan

Scénario - Étape Initiale

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

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

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

Scénario - Étape 2

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

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

Suite Étape 2

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

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

Scénario - Étape 3

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

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

Remarques

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

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

Scénario - Étape 4

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

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

Suite Étape 4

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

D'autres soucis ?

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

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

ou des questions :

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

Plan

commande ps I

Cette commande liste les processus selon différentes syntaxes d'options (BSD, standard).

```
$ ps f
  PID TTY      STAT   TIME COMMAND
19701 pts/1    Ss     0:00 bash
20173 pts/1    S+     0:00  \_ man ps
20183 pts/1    S+     0:00      \_ less
17752 pts/0    Ss     0:00 bash
20245 pts/0    R+     0:00  \_ ps f
```

- f (forest) représente la relation parent/enfant entre processus
- PID est l'identifiant de processus

commande ps II

- TTY est le terminal d'attachement du pus : ici deux onglets d'une même application Terminal présents dans `/dev/pts/` (pseudo terminaux)
- STATe état du pus :
  - S sleep interruptible (attente d'un événement)
  - s leader de Session
  - + groupe des pus d'avant-plan
  - Run désigne les pus éligibles (prêts) ou l'élu.
- TIME indique la durée passée sur le processeur en mode noyau et (+) en mode utilisateur

D'autres options permettent de visualiser d'autres colonnes :

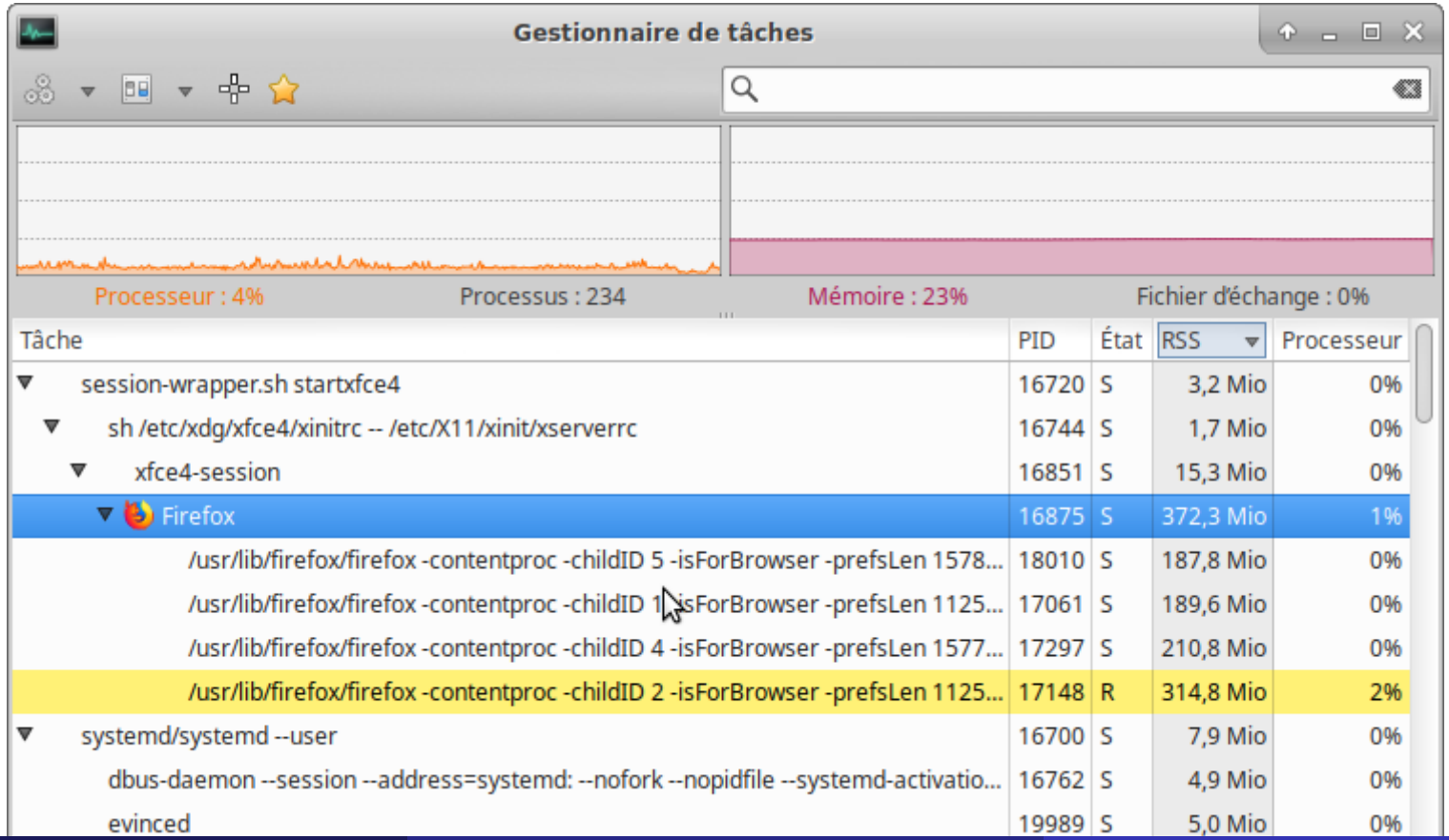
commande ps III

```
$ ps fux
USER        PID %CPU %MEM    VSZ   RSS TTY      STAT START TIME COMMAND
mmeynard  19449  0.0  0.0  14120  3176 ?        S    10:25 0:00  |   \_ /bin/b
mmeynard  19450  0.8  3.5 2144256 289652 ?      Sl   10:25 1:04  |       \_ ./
mmeynard  16942  0.0  0.1 173336 14784 ?        S    09:52 0:00  \_ /usr/lib/x
mmeynard  16947  0.0  0.2 331292 18844 ?        Sl   09:52 0:00  \_ /usr/lib/x
mmeynard  16985  0.0  0.3 343600 25772 ?        Sl   09:52 0:00  \_ /usr/lib/x
mmeynard  16989  0.0  0.4 692892 34008 ?        Sl   09:52 0:08  \_ /usr/lib/x
mmeynard  17011  0.0  0.3 329712 24844 ?        Sl   09:52 0:00  \_ /usr/lib/x
mmeynard  17748  0.1  0.5 646612 41264 ?        Sl   09:53 0:10  \_ /usr/bin/x
mmeynard  17752  0.0  0.0  14596  3824 pts/0    Ss   09:53 0:00      \_ bash
mmeynard  20437  0.0  0.9 404980 79280 pts/0    Sl   11:09 0:03      |   \_ em
mmeynard  23313  0.0  0.0  30596  3296 pts/0    R+   12:29 0:00      |   \_ ps
mmeynard  20856  0.0  0.0  14596  3812 pts/2    Ss+  11:09 0:00      \_ bash
```

- RSS Resident Set Size mémoire physique utilisée en Kio
- VSZ Virtual memory SiZe (Kio)
- Sl muLti-thread

Autres commandes

- top affiche la liste ordonnée des pus selon le pourcentage d'utilisation du processeur
- des gestionnaires de tâches existent pour chaque distribution qui permettent de visualiser les processus



Plan

Génération de processus

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

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

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

Sous Unix

Sous Unix, deux phases distinctes :

- mise en place d'un clône, par une copie de l'ensemble des segments du processus demandeur (fork)
- **optionnellement**, mise en place de nouveaux segments de code et données statiques, réinitialisation de la pile et du tas (exec)

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

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

Principe de fonctionnement de *fork()*

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

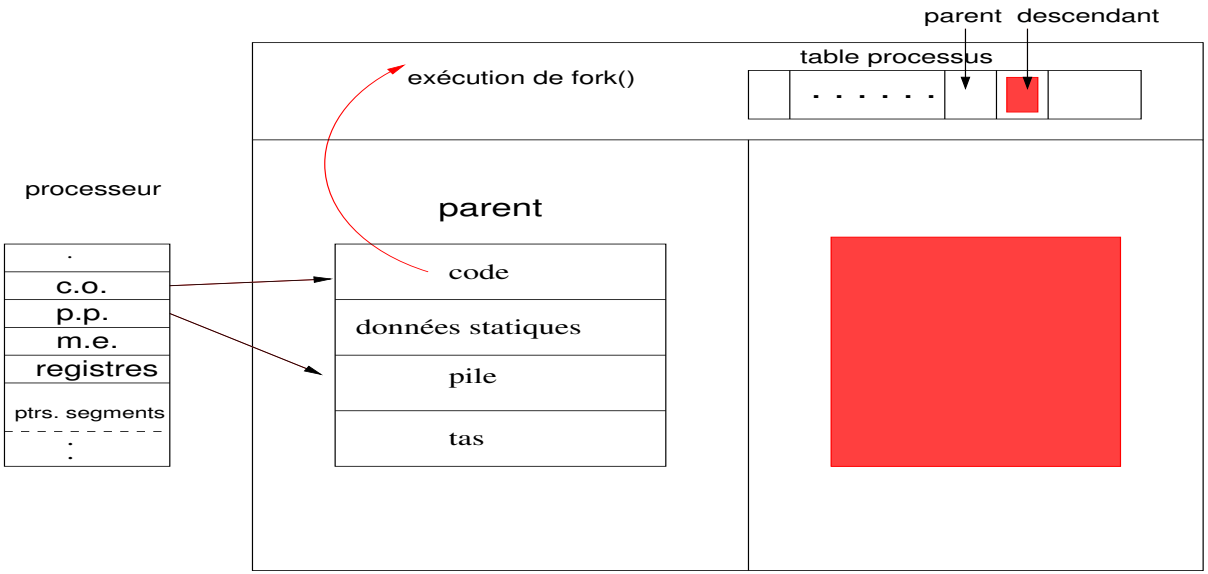
On peut noter que l'enfant aura un moyen de reconnaître son générateur ; en effet, si un parent peut avoir plusieurs enfants, un enfant ne peut avoir qu'un seul parent (ppid)

Schéma algorithmique fork()

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

Déroulement

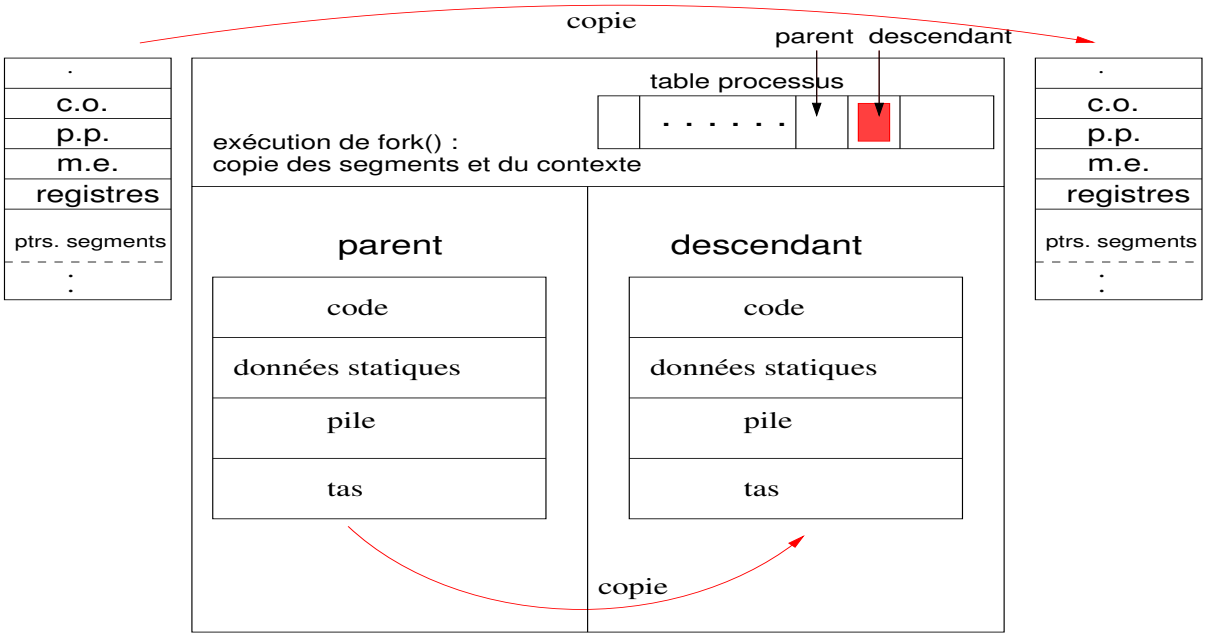
Le processus exécute ce code ; il y a appel noyau : *fork()*.



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

Suite et fin Déroulement

Cet appel fait une copie :



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

Plan

Unix : Recouvrement de Processus

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

Principe :

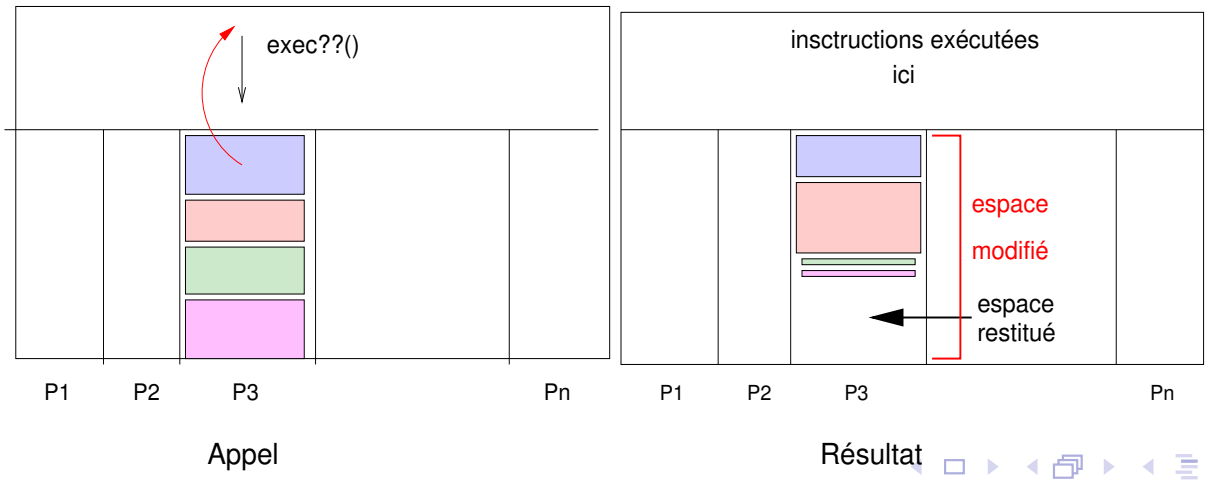
- vérifier l'existence et l'accessibilité (droits) du fichier exécutable ;
- écraser son propre segment de code par le nouvel exécutable ;
- générer un nouveau segment de données statiques ;
- vider le segment de pile ;
- vider le tas ;
- faire quelques modifications dans la table des processus (espace mémoire alloué, compteur ordinal, etc).
- placer le compteur ordinal sur le point d'entrée du code en lui passant les paramètres d'éxécution et l'environnement

Déroulement

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

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

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



Exemples I

Exemple 1 : L'interprète bash exécute la commande `ls` :

- le processus bash se duplique (fork)
- le processus parent (bash) se met en attente de la fin de son pus enfant (wait)
- le pus enfant se recouvre par la commande externe `/bin/ls`. Comme il a hérité du même terminal que son parent, il affiche la liste des fichiers dans le terminal. Lorsqu'il a fini (exit), un signal est envoyé au processus parent pour le réveiller
- le parent se réveille et affiche l'invite de commande $

Exemples II

Exemple 2 : L'interprète bash exécute la commande `vscodium &` :

- le processus bash se duplique (fork)
- le processus parent (bash) n'attend pas la fin de son pus enfant et affiche l'invite de commande $
- le pus enfant se recouvre par la commande externe `/net/apps/bin/vscodium`. Comme il a hérité du même terminal que son parent, il affiche les messages d'erreur de vscodium sur le terminal de bash

Question : Que se passe-t-il si l'interprète bash exécute la commande `echo Hello` ?

Plan

Utiliser les processus en Bash I

- tout processus possède un **environnement** constitué d'une liste de variables d'environnement qui sont copiées depuis le processus parent
- En Bash, pour voir les variables d'environnement : `env`
- Pour créer une nouvelle variable d'environnement : `export EDITOR=vscodium`
- Pour l'utiliser : `echo $EDITOR; $EDITOR &`
- Tout processus possède une entrée standard, une sortie standard, une sortie d'erreur standard qui sont généralement associées au terminal
- Redirection de sortie vers un fichier : `cmd1 > monfic`
- Redirection de sortie en concaténation vers un fichier : `cmd1 >> monfic`

Utiliser les processus en Bash II

- Redirection en entrée depuis un fichier : `cmd1 < monfic`
- Redirection de sortie d'erreur vers un fichier :
  `cmd1 2> anomalies.txt`
- toute commande renvoie un code d'état indiquant la bonne exécution de la commande (0) ou une erreur (différent de 0). En Bash, ce code est visible dans la variable `$?`
- `cmd1 ; cmd2` séquence de commandes
- `cmd1 && cmd2` enchaînement si réussite de cmd1
- `cmd1 || cmd2` enchaînement si échec de cmd1
- `cmd1 | cmd2` tube (pipe) entre la sortie standard de cmd1 et l'entrée standard de cmd2
- tuer le processus en avant-plan : Ctrl-C
- tuer un pus de pid 12345 : `kill 12345`

Utiliser les processus en Bash III

- générer une fin de fichier (EOF) depuis le clavier : Ctrl-D
- détacher un processus de son terminal en le lançant en arrière-plan : `vscodium &`

Plan

Le mécanisme de "tube" (pipe) et les filtres Unix I

- Tubes et filtres doivent être compris car ils permettent des manipulations complexes en utilisant des commandes existantes
- Un tube est un espace mémoire géré comme une file d'attente (FIFO) dans lequel un pus écrit des octets à la queue et un pus lit les mêmes octets dans le même ordre en tête
- l'écrivain et le lecteur sont souvent deux processus distincts
- Bash permet avec la syntaxe `cmd1 | cmd2` :
  - `cmd1` est l'écrivain qui enfile les octets produits dans sa sortie standard dans le tube (plus d'affichage dans le terminal car redirection dans le tube)
  - `cmd2` est le lecteur qui lit les octets depuis le tube et les utilise comme entrée standard (clavier redirigé)

Le mécanisme de "tube" (pipe) et les filtres Unix II

- les deux commandes sont lancées en parallèle et se synchronisent via le tube : si le tube est vide, le lecteur se bloque et sera réveillé dès que l'écrivain aura travaillé (respectivement si tube plein)
- Un filtre est une commande qui reçoit un flot de données sur son entrée standard, les transforme et écrit un flot de données sur sa sortie standard
- On compose les filtres par le mécanisme de tube pour obtenir de nouveaux filtres
- Il suffit de connaître les filtres de base proposés avec le système Unix, pour résoudre une bonne part de problèmes de transformation ou de recherche de fichier

Quelques filtres courants I

- `grep expreg` permet d'extraire de son entrée standard chaque ligne de texte contenant un mot correspondant à l'**expression régulière** `expreg` :

```
$ ls | grep '.txt$'
toto.txt
vide.txt
$ cat spair.c | grep include
#include <stdio.h>
#include <stdlib.h>
#include "pair.h"
```

- `cat` est le filtre nul : il transmet les caractères de l'entrée standard sur la sortie standard

Quelques filtres courants II

```
$ cat
je tape
je tape
au clavier
au clavier
```

- les filtres peuvent aussi être utilisés comme des commandes classiques en fournissant un paramètre fichier. `cat` permet ainsi de conCATéner tous les fichiers passés en paramètre :

```
$ cat *.c | grep "int i"
for (int i=0;i<argc;i++){
int i=0;
$ grep ash /etc/shells
/bin/bash
```

- `head -n` sélectionne les n premières lignes (par défaut n=10)
- `tail -n` sélectionne les n dernières lignes

Quelques filtres courants III

- `wc` compte les mots (Word Count), les octets et les lignes d'un fichiers

```
$ tail -4 pair.c | grep return | wc
2       4      40 # 2 lignes, 4 mots, 40 octets
```

- `tee fic` recopie l'entrée standard dans la sortie standard tout en dupliquant l'entrée standard dans `fic` (permet une sauvegarde en cas d'enchaînement de nombreux pipes)
- `cut` permet de ne conserver que certaines colonnes de fichiers `.csv` (Character Separated Values) correspondant à des feuilles de calcul ou des fichiers de configuration
- `sed` (Stream EDitor) transforme le texte entré :

```
$ echo "il court" | sed "s/il/elle/"
elle court
```

Plan

Plan

Interprétation vs Compilation I

- L'interprétation permet un cycle de développement plus rapide : édition du code, test par interprétation, édition du code, ...
- L'exécution du code est plus lente qu'avec du code machine compilé
- Diverses solutions intermédiaires existent :
  - Java : compilation `javac` puis interprétation (`java, Python`)
  - Just In Time Compiler : compilation juste avant l'interprétation (JavaScript)

Bash, un langage et un interpréteur de commandes I

Bourne Again SHell est un langage assez difficile à appréhender pour diverses raisons :

- syntaxe complexe (espaces, retour ligne) et messages d'erreur peu explicatifs
- destiné à taper des commandes ou à écrire de petits scripts utilitaires
- des types de données très limité : chaînes de caractères 90% et tableaux (il existe des dictionnaires et des entier POSIX ...)
- certaines chaînes de caractères peuvent être traitées comme des expressions arithmétiques

Bash et les variables I

- une variable a un nom et une valeur qui sont tous deux des chaînes de caractères !
- l'affectation d'une variable est réalisée par l'opérateur = **sans espace autour** : `mavar='hello world !'` ou bien `v=toto`
- une chaîne de caractères ne contenant pas d'espaces n'a pas besoin de guillemets ou d'apostrophes
- l'évaluation d'une variable est déclenchée par l'opérateur $:

```
$ echo $mavar $v
hello world ! toto
```

- Cette substitution (et d'autres) est possible à l'intérieur d'un littéral chaîne entouré de guillemets mais pas d'apostrophe :

```
$ nom=Michel
$ echo "hello world $nom \!"
hello world Michel \!
```

Bash et les variables II

- les **variables locales** créées par affectation ne sont visibles que dans le processus bash courant
- les **variables d'environnement** créées par `export NOM=Michel` sont visibles dans tous processus enfant du bash courant
- On peut voir ces variables (par convention en majuscules) avec la commande `env` (TERM, SHELL, HOME, USER, PATH, DISPLAY, ...)

Script Bash I

- un script bash est un fichier texte contenant une séquence de commandes Bash et auquel on a donné le droit d'exécution
- on le lance soit en préfixant son nom par l'interpréteur `bash` ou bien en indiquant en première ligne la localisation de ce dernier
- un script a accès à ses arguments (argv du C) grâce à une notation positionnelle `$1 $2 $3 ...`
- un script a accès à son environnement puisqu'il hérite des variables d'env. de son processus parent

Exemple : fichier `hello.sh`

```
echo "hello"
```

Exécution

```
$ bash hello.sh
hello
```

Script Bash exécutable I

On souhaite lancer notre script comme une autre commande sans préfixer par bash :

- rendre le script exécutable : `chmod u+x hello.sh`
- ajouter une **première** ligne indiquant l'emplacement de l'interpréteur :

```
#!/bin/bash
echo "hello $1"
```

- lancer la commande :

```
$ ./hello.sh Dupont
hello Dupont
$ ./hello.sh
hello
```

- Enfin, si l'on veut lancer notre exécutable sans le préfixer par `./`, il faut ajouter le répertoire de travail (.) dans la variable d'environnement `PATH`

Script Bash exécutable II

```
$ export PATH=${PATH}:.
$ hello.sh titi
hello titi
```

- Afin de généraliser l'exécution de programmes situés dans le répertoire courant, il suffit de recopier la ligne d'exportation de `PATH` dans le fichier de configuration de bash dénommé `~/.bashrc` qui est exécuté à chaque lancement d'un nouveau processus bash

Plan

Compilation des programmes (C sous Unix) I

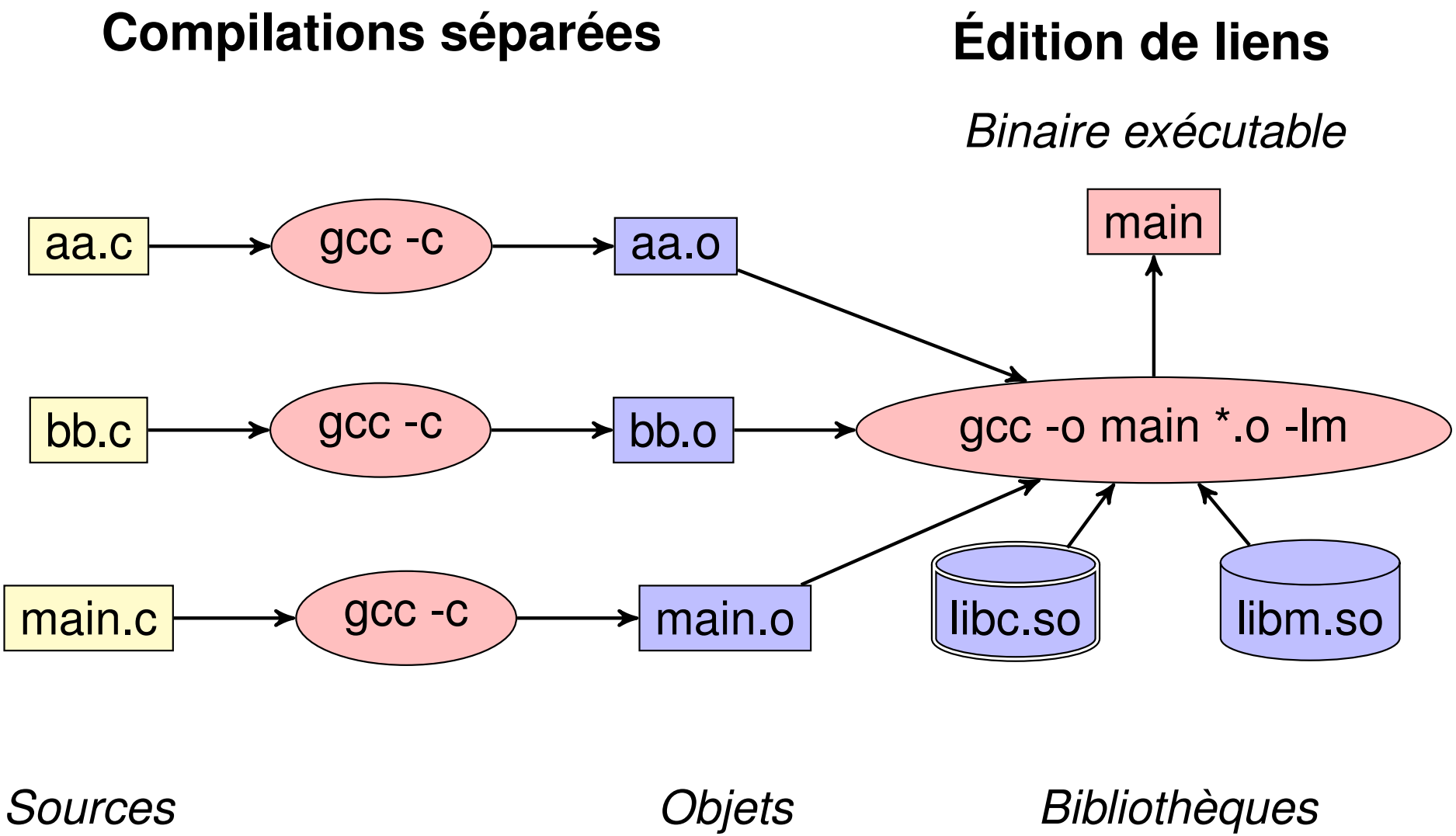
- Unix développé en C donc interface naturelle avec le SE
- Compilation vs interprétation : maîtriser les phases
  - Prétraitement des directives de compilation (`#include, #define, #ifdef, ...`) de chaque fichier
  - Analyse lexicale et syntaxique (parse error ou syntax error)
  - Analyse sémantique (correspondance de type, déclaration préalable des objets, . . .)
  - Compilation proprement dite du source C en source écrit en langage d'assemblage
  - Assemblage en fichier objet .o
  - Edition des liens des objets entre eux et avec la ou les bibliothèques pour réaliser le **fichier binaire exécutable**
- Cette succession est souvent réalisée à l'aide d'une unique commande : `gcc monprog.c -o monprog`

Compilation des programmes (C sous Unix) II

- la commande `gcc` supporte ces principales options :

-c	Compiler et assembler seulement (compile)
-o xxx	Renommage du fichier de sortie (output)
-lm	Utilisation de la librairie mathématique `libm.a` ou `.so`
-Wall	Voir tous les avertissements (Warning all)
-g	Ajoute les informations de débogage nécessaires à `gdb`
-E	Que le prétraitement
-S	Compiler sans assembler
-std=c99	Permet les déf de var dans les for, les //, (c11 pour le standard C 2011)
-static	pour l'édition de lien statique

Processus de compilation



Structure d'un programme C I

```
Cours$ cat argv.c

#include <stdio.h>

int main(int argc, char *argv[], char *env[]) {
  printf("Nombre d'arguments : %i\n\nListe des arguments
   ↪  :\n",argc);
  for (int i=0;i<argc;i++){
    printf("%s\n",argv[i]);
  }
  return 0;
}
```

Structure d'un programme C II

```
Cours$ gcc -o argv argv.c -Wall
Cours$ argv un 2 34.5
Nombre d'arguments : 4

Liste des arguments :
argv
un
2
34.5
Cours$ echo $?
0
```

Les fonctions I

Déclaration d'une fonction

```
char* itoa(int, char *);
```

- le type de retour de la fonction (`char*`)
- le nom de la fonction (`itoa`)
- la liste des types des paramètres (éventuellement accompagnés des noms de paramètres formels)
- un `;` indispensable
- plusieurs déclarations identiques de la même fonction sont possibles (inclusions multiples du même fichier d'en-tête)
- le type `void` permet de déclarer une fonction sans résultat ou sans paramètre

Les fonctions II

Définition d'une fonction

```
char* itoa(int i, char *s){ ...}
```

- le ; est remplacé par un **bloc** d'instructions
- les noms des paramètres formels sont indispensables
- pas de surcharge : une unique définition dans tout le programme
- la fonction `main()` est l'unique point d'entrée du programme. C'est une fonction comme les autres (elle peut être appelée récursivement)
- la déclaration d'une fonction doit **précéder** son appel, mais sa définition peut être absente (dans un autre fichier objet ou dans une bibliothèque)

Un exemple complet : fact.c I

```c
#include <stdio.h>
#include <stdlib.h>

unsigned int fact(unsigned int);

int main(int argc, char* argv[], char* env[]){
  if(argc!=2){
    fprintf(stderr, "Syntaxe incorrecte : %s <entier>\n",
    ↪  argv[0]);
    return 1;
  }
  int n=atoi(argv[1]);
  if (n<0){
    fprintf(stderr, "L'argument doit être un entier
    ↪  positif !\n");
    return 2;
```

Un exemple complet : fact.c II

```
  }
  printf("%d!=%d\n",n,fact(n));
  exit(0);                                  /* ou return  0 */
}
unsigned int fact(unsigned int i){
  if (i<=1)
    return 1;
  else
    return i*fact(i-1);
}
```

Un exemple complet : fact.c III

Quelques exécutions

```
Cours$ gcc -o fact -Wall fact.c; fact
Syntaxe incorrecte : fact <entier>
Cours$ fact -65
L'argument doit être un entier positif !
Cours$ echo $?
2
Cours$ fact 12
12!=479001600
Cours$ fact toto
0!=1
Cours$ echo $?
0
```

Passage des paramètres I

- En C, le seul mode de passage des paramètres à une fonction est le passage par **copie** (aussi appelé par valeur) : une copie du paramètre réel (d'appel) est placé sur la pile et c'est cette copie qui est ensuite utilisée par la fonction appelée
- il ne peut donc pas y avoir de modification par l'appelée sur le paramètre réel
- le passage d'un paramètre de type pointeur permet à l'appelée de modifier la zone pointée mais pas le pointeur lui-même
- le passage d'un tableau à une fonction est similaire au passage du pointeur sur la première case de ce tableau : par conséquent, le contenu du tableau pourra être modifié

Passage des paramètres II

Exemple passageparam.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void modifieur(int i, char* s, float t[2]){
  i=i+1;
  s[0]='M'; s=NULL;
  t[0]=0.0; t++;
  return;
}
int main(int argc, char* argv[], char* env[]){
  int n=5;
  char* ch=malloc(strlen("bonjour")+1);
  // les chaînes littérales sont const!
  strcpy(ch,"bonjour");
  float compl[2]={1.1,2.2};
```

Passage des paramètres III

```
  printf("AVANT : n=%d ; ch=%s ; compl={%f,%f} ; &ch=%p ;
   ↪  &compl=%p\n", n, ch, compl[0], compl[1], &ch,
   ↪  &compl);
  modifieur(n,ch,compl);
  printf("APRES : n=%d ; ch=%s ; compl={%f,%f} ; &ch=%p ;
   ↪  &compl=%p\n", n, ch, compl[0], compl[1], &ch,
   ↪  &compl);
  return 0;
}
```

Exécution

```
AVANT : n=5 ; ch=bonjour ; compl={1.100000,2.200000} ;
&ch=0x7fff53fd7a68 ; &compl=0x7fff53fd7a90
APRES : n=5 ; ch=Monjour ; compl={0.000000,2.200000} ;
&ch=0x7fff53fd7a68 ; &compl=0x7fff53fd7a90
```

Les variables C : propriétés I

- en C, toute variable est **typée**, ce qui lui donne une taille (sizeof) et un codage (voir représentation des données)
- une variable est **située** dans un des deux segments suivant : la pile, le segment de données statique. Un objet dynamique est situé dans le tas, il n'est accessible que par un pointeur
- la **durée de vie** d'une variable ou d'un objet dyn. est liée à sa localisation :
  - pile : durée de vie de la fonction dans laquelle elle a été définie
  - tas : depuis le `malloc()` jusqu'au `free()`
  - statique : durée de vie du processus
- la **portée** d'une variable est la zone du programme où elle peut être utilisée. Une variable définie en dehors de toute fonction a une **portée globale** à toute l'application (sauf si `static` qui limite au fichier). Une variable définie dans une fonction ou dans un bloc a une **portée locale** au bloc.

Les variables C : propriétés II

- la **résolution** de portée d'un nom de variable consiste à remonter les blocs englobants pour retrouver la définition de variable la plus proche
- une variable globale peut être déclarée en la faisant précéder du mot-clé `extern` : `extern int g;`. Toute variable ne peut être définie qu'une **unique** fois

Exemple portee.c

```
#include <stdio.h>
float g=10.2;
int main(int argc, char* argv[], char* env[]){
  {
    char g='A';
    for(int g=1;g<5;g++){
      printf("g=%d;",g);
    }
```

Les variables C : propriétés III

```
    printf("\ng=%c\n",g);
  }
  printf("g=%f\n",g);
  return 0;
}
```

Exécution

```
Cours$ portee
g=1;g=2;g=3;g=4;
g=A
g=10.200000
```

Les types C I

Un type de données utilise un système de codage et a une taille (`sizeof()`) qui est un multiple de l'octet. Le codage des types est fixé dans la norme du langage tandis que leur taille dépendent parfois des architectures de machines (32 bits, 64 bits, ...).

char entier signé en complément à 2 sur 1 octet. Il permet de représenter les caractères ASCII (7 bits), les octets lus dans des fichiers, les cases des chaînes de caractères. Il existe aussi `signed char` et `unsigned char`

int entier signé en complément à 2 de taille dépendant de la machine (souvent 4 octets). Le type `unsigned int` est de même taille mais codé en RBNS

Les types C II

short, long entier court (long) codé en complément à 2 dont la taille dépend de l'architecture. Par exemple (Mac OS X i5) : int(4), short(2), long(8), long long(8). Les types non signés correspondants sont possibles.

C99 cette norme définit les types de taille fixée : `int8_t`, `uint8_t`, `int16_t`, `uint32_t`. Elle définit également des types rapides de taille minimale comme : `uint_fast64_t` (en-tête `stdint.h`)

float nombre flottant IEEE-754 avec des tailles non fixées : `float(4)`, `double(8)`, `long double(16)`

Les types C III

pointeur un pointeur est une adresse mémoire (entier non signé) sur un objet d'un certain type. Le type `char *` n'est pas le même que `int *` même s'ils ont la même taille. Le type `void *` est un pointeur générique (sur n'importe quoi). Le pointeur `NULL` vaut 0 et pointe sur une adrs mémoire interdite ! L'arithmétique des pointeurs est basée sur une unité égale à la taille du type pointé : incrémenter un pointeur sur `char` avance de 1 alors que sur un `int*` l'incrémentation avance de `sizeof(int)` (4)

Les types C IV

tableau séquence d'objets de même type e.g. `int t[4]`. La taille d'un tableau n'est pas définie dans le tableau : il faut soit la conserver dans une autre variable (`argc` est la taille d'`argv`), soit positionner un objet terminateur à la fin de la séquence (`'\0'` en fin de chaîne, `NULL` en fin d'`env`). Depuis c99, la taille d'un tableau local peut être initialisé à l'exécution. La taille d'un tableau(`sizeof()`) est la taille d'un objet multiplié par le nombre de cases. Le nom du tableau peut être vu comme un pointeur constant adressant la première case. L'opérateur d'indexation ([exp]) peut être appliqué à un nom de tableau comme à un pointeur pour référencer une case.

Les types C V

struct séquence hétérogène d'objets e.g. :
`struct cell{int val;struct cell *suiv;} tete;`
Les champs de la `struct` sont référencés grâce à la notation pointée sur la variable `tete` : `(tete.suiv)->val` est la seconde valeur de la liste. **Il faut allouer (malloc) de la mémoire aux structures de données dynamiques.**

union un champ parmi plusieurs possibles :
`union dyn{int i;float f;char *s;} x;` x est une variable qui peut contenir un entier, un flottant ou une chaine. La taille d'une union est la taille de son plus grand composant.

typedef permet de définir un nouveau type : `typedef exptype nom;`

Un exemple de type liste I

liste.h

```
/** @file liste.h
* @brief en-tête des fonctions de manipulation de liste
* @author Michel Meynard*/
#ifndef _LISTE
#define _LISTE
/** @typedef liste
* @brief le type liste est un pointeur sur cell. */
typedef struct cell* liste;
/** @typedef cell
* @brief une cellule composée d'un entier et d'une liste.
 ↪  */
typedef struct cell {
  int val;      /**< élément proprement dit */
  liste suiv;   /**< pointeur sur cellule suivante */
} cell;
```

Un exemple de type liste II

```
/** Crèe une liste d'entier vide.
* @return une liste vide (NULL)
*/
liste creerListe();
/** Teste si une liste est vide.
* @param l la liste à tester
* @return 0 si non vide, 1 sinon
*/
int vide(liste l);
/** Retourne le premier entier de la liste sans le
 ↪  retirer.
* @param l la liste
* @return le premier entier de l
* @warning non défini si liste vide
*/
int premier(liste l);
/** Retourne la liste l sans son premier élément (sans le
 ↪  désallouer et
```

Un exemple de type liste III

```
* sans modifier l).
* @param l la liste
* @return la suite de la liste
* @warning non défini si liste vide
*/
liste suite(liste l);
/** Retourne la liste l à laquelle on a ajouté un nouveau
* premier élément.
* (sans modifier l)
* @param i l'entier à ajouter en premier
* @param l la liste
* @return la nouvelle liste*/
liste ajDeb(int i, liste l);
/** Teste si un entier fait partie d'une liste.
* @param i l'entier recherché
* @param l la liste à tester
* @return 1 si i est dans l, 0 sinon
```

Un exemple de type liste IV

```
*/
int dansListe(int i, liste l);
/** Vide une liste en désallouant toutes ses cellules.
* @param pl un pointeur sur la liste à vider
* @warning effets de bord sur des listes qui
 ↪  partageraient des cellules
*/
void vider(liste *pl);
#endif /* _LISTE */
```

Un exemple de type liste V

liste.c

```
#include<stdlib.h>
#include "liste.h"
liste creerListe(){return NULL;}        // creer une liste
 ↪  vide
int vide(liste l){return (l==NULL);}    // teste si vide
int premier(liste l){return l->val;}    // non defini si
 ↪  liste vide
liste suite(liste l){return l->suiv;} // non defini si
 ↪  liste vide
liste ajDeb(int i, liste l){                  // ajoute i au
 ↪  début de l
  liste nouv=(liste) malloc(sizeof(cell));
  nouv->val=i;
  nouv->suiv=l;
  return nouv;
```

Un exemple de type liste VI

```
}
int dansListe(int i, liste l){          // vrai si i dans l
  return !vide(l) && (
          i==premier(l) ||
          dansListe(i,suite(l))
          );
}
void vider(liste *pl){                  // pl est un
 ↪  pointeur sur la liste
// vide recursivement une liste (attention aux listes qui
 ↪  pointaient dessus !)
  if (vide(*pl)) return;    // vidage d'une liste par
   ↪  desalloc de ses cellules
  else {
    vider(&((*pl)->suiv));          // appel recursif
    free((liste)*pl);                  // desalloue, ne
     ↪  modifie pas
```

Un exemple de type liste VII

```
        (*pl)=creerListe();
        return;
    }
}
```

Un exemple de type liste VIII

main.c

```
#include <stdio.h>
#include <stdlib.h>
#include "liste.h"
int main(int argc, char *argv[]){
  if(argc<2){
    fprintf(stderr, "Un argument entier S.V.P. !\n");
    return 1;
  }
  liste prems=ajDeb(2,ajDeb(3,ajDeb(5,ajDeb(7,ajDeb(11,
   ↪  ajDeb(13,creerListe()))))));
  if(dansListe(atoi(argv[1]),prems)){
    printf("%d est un nombre premier < 17 !\n",
     ↪  atoi(argv[1]));
  }else{
    printf("%d n'est pas un nombre premier < 17 !\n",
     ↪  atoi(argv[1]));
```

Un exemple de type liste IX

```
    }
    vider(&prems);
    return 0;
}
```

Compilation puis exécution

```
gcc -o main liste.c main.c -std=c99 -Wall
$ main 13
13 est un nombre premier < 17 !
$ main 6
6 n'est pas un nombre premier < 17 !
```

Plan

Conclusion

Il est absolument indispensable de comprendre les concepts de base des machines informatiques et des systèmes d'exploitation afin :

- de manipuler différents systèmes d'exploitation en comprenant leurs différences et leurs ressemblances
- d'évaluer correctement les résultats numériques fournis par les machines (précision)
- de programmer intelligemment les algorithmes dont on a minimisé la complexité (des E/S fréquentes peuvent ruiner un algorithme d'une complexité inférieure à un autre)
- de pouvoir optimiser les parties de programme les plus utilisées en les réécrivant en langage de bas niveau (C)
- de se préparer à la programmation concurrente
- d'oser utiliser des systèmes d'exploitation dont le code source est connu !