Au menu aujourd'hui

Généralités

- Déroulement du cours
- Architecture générale d'un système
- Choix du cours : Unix et OCaml
- OCaml et le système d'exploitation

Le système de fichiers

- Système de fichiers
- Opérations sur le système de fichiers

Déroulement du cours

Principes et Programmation des Systèmes d'exploitation

Didier Rémy

Chercheur à l'INRIA-Rocquencourt

Email : Didier.Remy@inria.fr

URLs du cours :

http ://pauillac.inria.fr/~remy/poly/system/

http ://www.enseignement.polytechnique.fr/profs/informatique/Didier.Remy/system/

Travaux dirigés :

Fabrice Le Fessant et Maxence Guesdon

Objectifs du cours

Comprendre le fonctionnement du système.

- Comment sont stockées les données sur le disque ?

  Comment fait-on pour agrandir un fichier ? de plusieurs gigas ? Pour déplacer un fichier dans la hiérarchie ?

- Que se passe-t-il quand vous exécutez un programme ?

  Qui lance votre programme ? Comment votre programme fait-il pour lire un fichier ?

- Comment préserver l'intégrité du système quand votre programme plante ? quand la machine plante ?

  Si votre programme prend feu, jusqu'où se propage l'incendie ?

- Que se passe-t-il quand vous imprimez un fichier ? ouvrez une fenêtre ? ou broutez la toile ?
- Comment vous redonner la main même lorsqu'un programme boucle ? équitablement ?

Objectifs du cours

Comprendre le fonctionnement du système.

Apprendre la programmation système.

- Comment interagir avec le système d'exploitation ? (depuis un programme utilisateur)
- avec d'autres programmes tournant sur la même machine ?
- ... sur une machine distante ?

Objectifs du cours

Comprendre le fonctionnement du système.

Apprendre la programmation système.

Illustrer quelques principes.

- Modularisation
  Interfaces bien définies entre des composants, indépendemment de leur implémentation.
- Architecture multi-couches
  Statification des opérations complexes.
- Compromis entre efficacité, simplicité, et robustesse.

Objectifs du cours

Comprendre le fonctionnement du système.

Apprendre la programmation système.

Illustrer quelques principes.

Simplifier

- La programmation système est réputée difficile.
- Nous tenterons de la rendre simple.

- En simplifiant souvant la réalité.
- Mais il est toujours possible de détailler *selon les besoins*.
  - On expliquera les principes dans le cours.
  - Pour utiliser un appel système, la documentation on-line (`man` et `OCaml`) suffisent.
  - Pour comprendre tous les détails d'une partie du système, il faut un manuel Unix ou Linux.

Fonctionnement du cours

9 Cours + 9 TDs, calendrier sur la page du cours.

Présence nécessaire en TD comme en cours

sauf exception,
mais l'exception ne doit pas être la règle

(même si ce n'est pas la ligne politique de l'École...)

- Ce qui est dit en cours est supposé connu de tous (tenez-vous informés si vous êtes absents).
- Sans surprise, mais expérimentalement, ceux qui décrochent sont aussi ceux qui sont absents en cours ou en TD.
- On ne refait pas le cours en TD pour les absents.

Fonctionnement du cours

9 Cours + 9 TDs, calendrier sur la page du cours.

Présence nécessaire en TD comme en cours

sauf exception,
mais l'exception ne doit pas être la règle
(même si ce n'est pas la ligne politique de l'École...)

Les questions sont les bienvenues

- N'hésitez pas, pendant le cours.
- Si vous décrochez pendant le cours, posez des questions à la fin du cours ou pendant les TDs.

Commentaires/Suggestions sur le cours

(format, rapidité, etc.)

- N'hésitez pas, après le cours.
- Directement, ou en passant par vos délégués.
- N'attendez pas la fin du trimestre.

Évaluation

Un examen écrit, portant sur l'ensemble du cours théorique (principes) et pratique (programmation).

Les notes de cours pourront être interdites pour tout ou une partie de l'examen.

Pour ceux qui le choisissent en système

C'est une autre évaluation, indépendante.

Généralités

Rôles du système d'exploitation

Intéragir avec le monde extérieur

- Un programme calcule une fonction mathématique.
- Au minimum, il vous faut lui donner des entrées (au clavier, par le réseau), lire le résultat (à l'écran, sur papier, etc.)

Rôles du système d'exploitation

Intéragir avec le monde extérieur

Effectuer les tâches de bas niveau

- Intéraction avec les périphériques
- Pas besoin de programmer le protocole d'intéraction pour imprimer un fichier ou se connecter par téléphone.

Rôles du système d'exploitation

Intéragir avec le monde extérieur

Effectuer les tâches de bas niveau

Protéger la machine

- Contre des ordres corrects non voulus.
  (effacer tout le disque par exemple)
- Contre des ordres incorrects envoyés aux périphériques.
- Au pire, votre programme plante, pas la machine.

Rôles du système d'exploitation

Intéragir avec le monde extérieur

Effectuer les tâches de bas niveau

Protéger la machine

Cacher les détails de votre machine

L'interface système

- donne l'illusion que toutes les machines sont identiques.
- rend les programmes (relativement) portables.

Rôles du système d'exploitation

Intéragir avec le monde extérieur

Effectuer les tâches de bas niveau

Protéger la machine

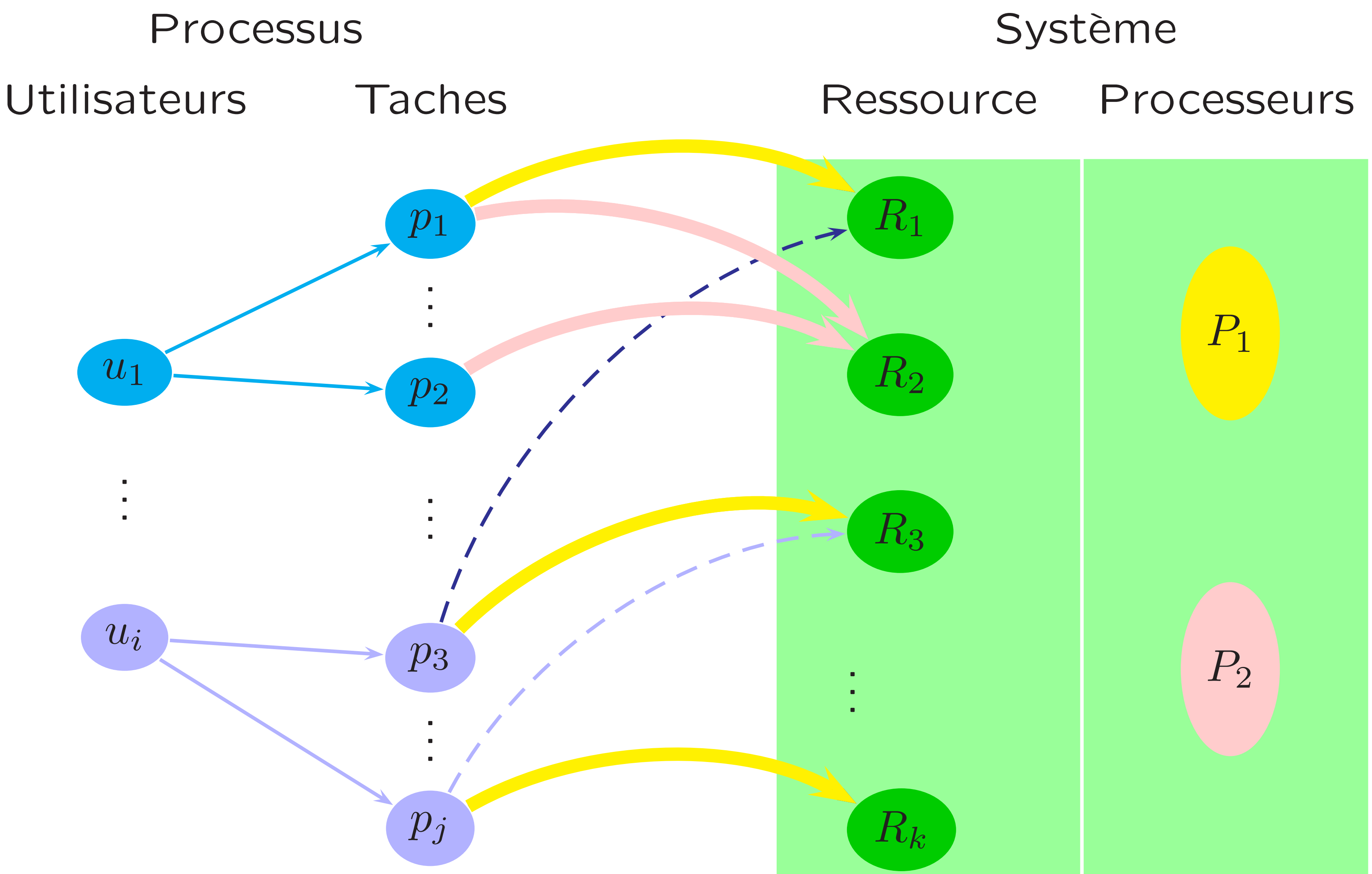
Cacher les détails de votre machine

Partager la machine

Le système d'exploitation donne l'illusion que

- le programme tourne seul sur la machine.
- l'utilisateur est seul sur la machine.

Résumé (partiel)



1234 21314342

À quoi ça ressemble ?

Une baguette magique

- C'est votre vision jusqu'aujourd'hui :
- Vous faites, sans le savoir, appel au système, qui vous répond comme par magie, pour :

  lire un caractère, créer un fichier, lire un fichier, etc.

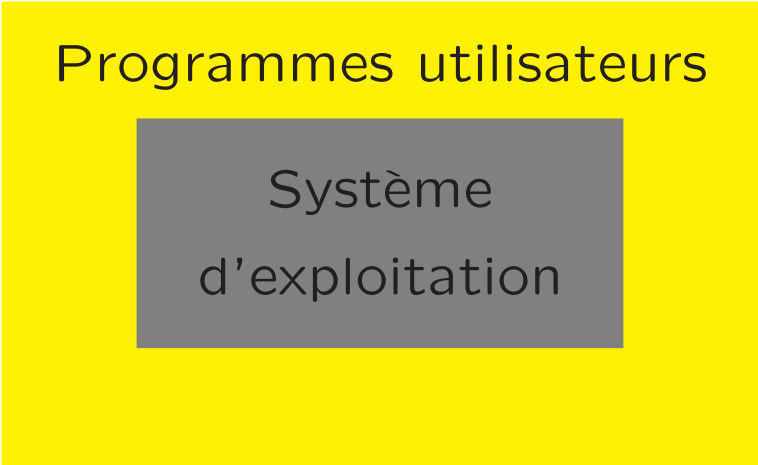
- Mais aussi, sans le demander vous êtes sous le contrôle du système :

  c'est le système qui lance votre proramme, alloue la mémoire pour le faire tourner, l'interrompt pour entrelacer l'exécution d'autres programmes.

À quoi ça ressemble ?

Une baguette magique

Une boîte noire



- Vue de l'extérieure = programmation *avec le* système

À quoi ça ressemble ?

Une baguette magique

Une boîte noire

- Vue de l'intérieur = programmation *du* système

À quoi ça ressemble ?

Une baguette magique

Une boîte noire

Programmes Utilisateurs

Système d'exploitation

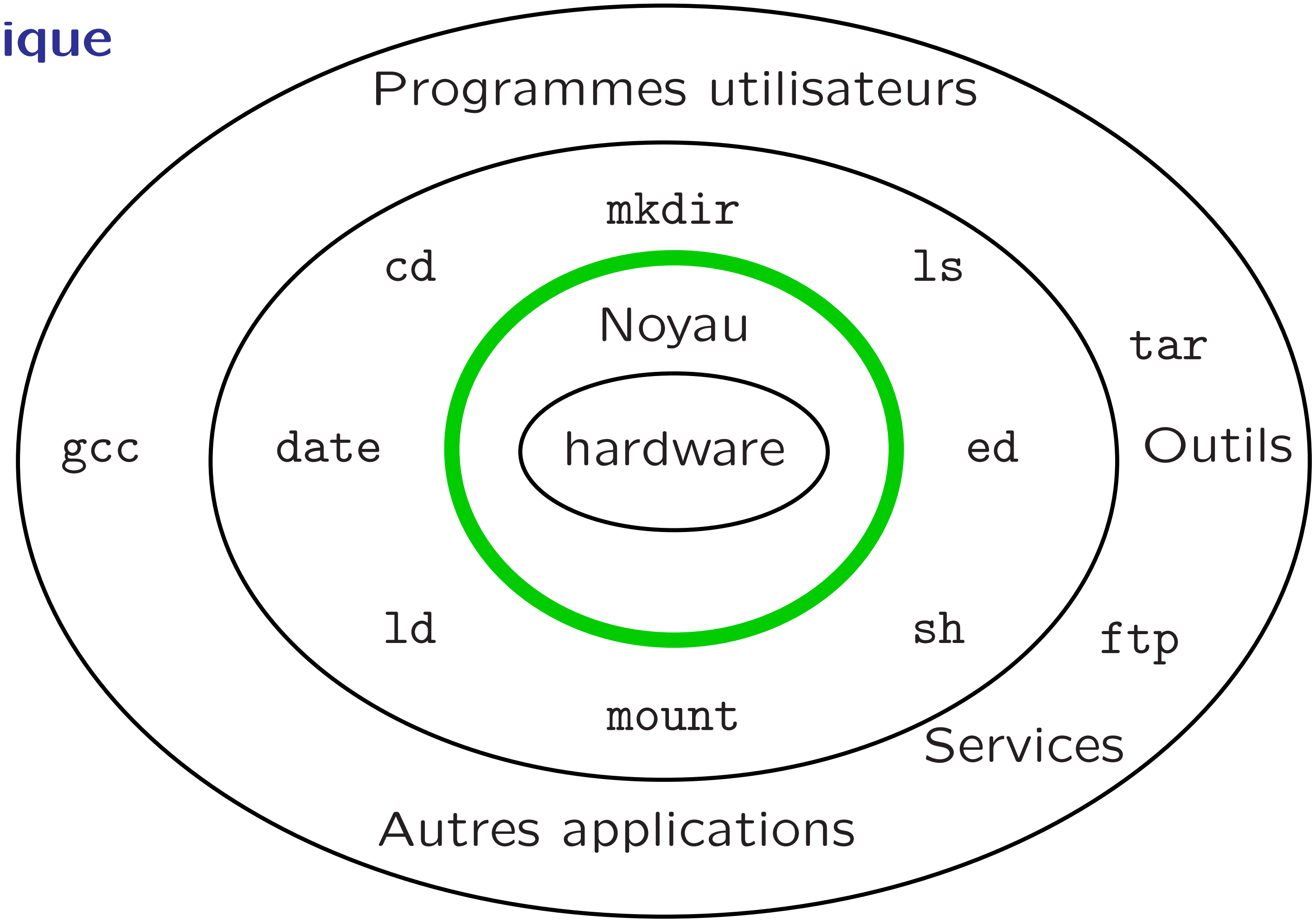
- Entre les deux, une interface rigoureusement définie :
  - c'est le jeu d'appels systèmes
  - décrits avec leur sémantique précise.
    Comportement de chaque appel système *y compris* en cas d'anomalie, avec la liste complète des erreurs qui peuvent être retournées.

À quoi ça ressemble ?

Une baguette magique

Une boîte noire

Un oignon



À quoi ça ressemble ?

Une baguette magique

Une boîte noire

Un oignon

- Beaucoup de couches empilées ou juxtaposées.
- Seules les couches profondes dépendent de la machine.
- Le plus petit possible : les couches qui peuvent être mises à l'extérieure ne font pas partie du système (diminue le noyau critique donc augmente la robustesse)
- Ne restent que celles qui sont essentielles pour la sûreté ou l'efficacité.

À quoi ça ressemble ?

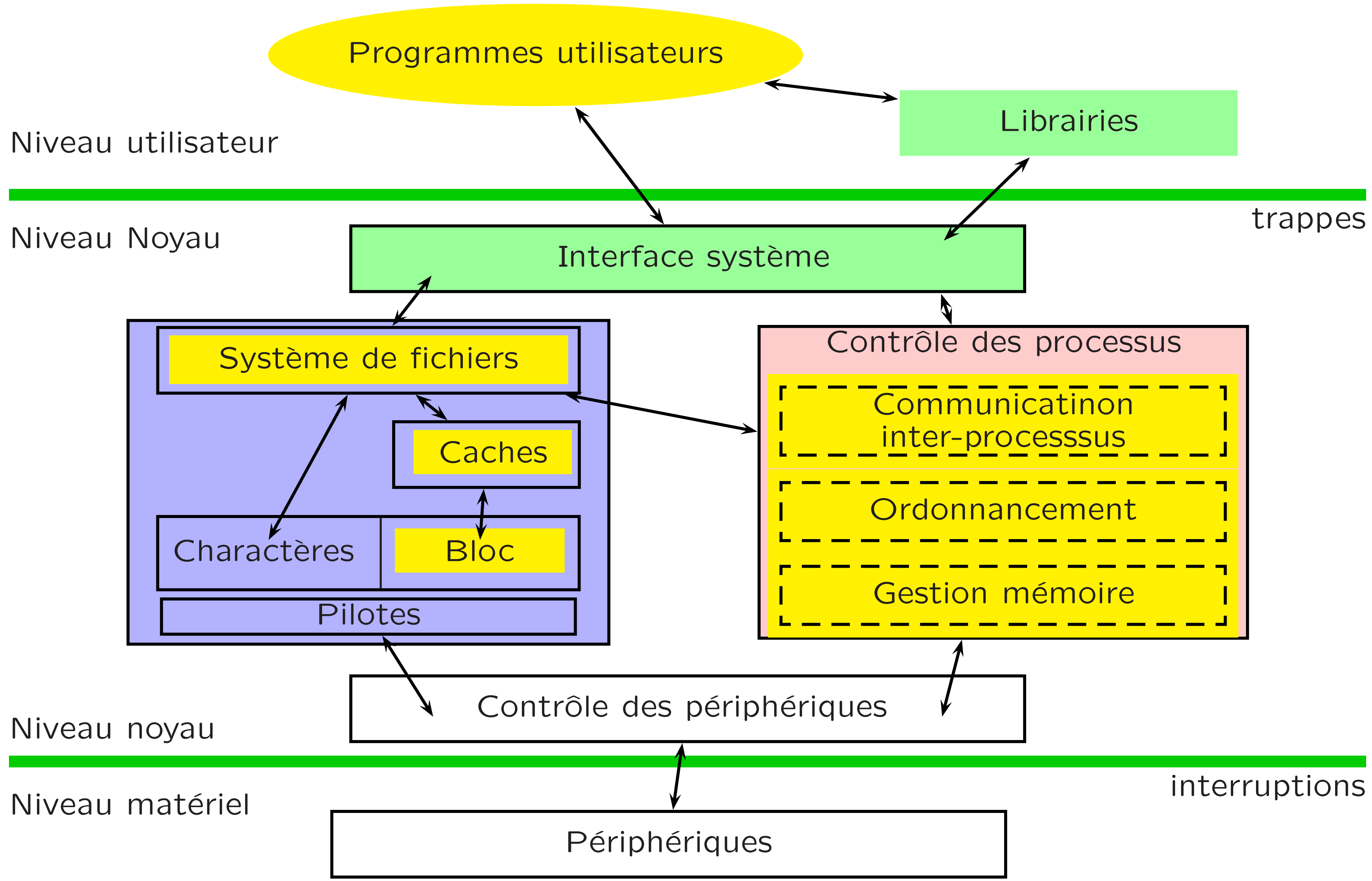
Une baguette magique

Une boîte noire

Un oignon

En fait

- c'est une architecture complexe,
- qui doit être robuste, portable, *etc.*
- Un bon exemple de programmation *de* systèmes.



Programmes utilisateurs
Librairies
Niveau utilisateur
trappes
Niveau Noyau
Interface système
Système de fichiers
Caches
Charactères
Bloc
Pilotes
Contrôle des processus
Communicatinon inter-processsus
Ordonnancement
Gestion mémoire
Contrôle des périphériques
Niveau noyau
interruptions
Niveau matériel
Périphériques

Plan du cours

Programmation du système

Les fichiers

Les processus

Création, interruptions, ordonnancement des taches.

Communication inter-processus

Signaux, tuyaux (pipes), prises (sockets)

Communication inter-machines

Prises (sockets)

Les coprocessus (threads)

Concurrence, verrous, conditions

Plan du cours

Programmation du système

Implémentation du système

(quelques coups de projecteurs sur...)

Fichiers, Tampons

Mémoire

Gestion mémoire, mémoire virtuelle, caches.

Processus

Ordonnancement, *etc.*

Quel système d'exploitation ?

pour le cours.

Unix, Linux, Mach... Windows ?

pour le cours.

Unix, Linux, Mach... Windows ?

En principe, le choix importe peu

Les grands principes des systèmes d'exploitation transcendent les particularités de tel ou tel système.

Les besoins sont les mêmes système de fichiers, gestion mémoire, ordonnancement des taches, *etc.*

Les progrès sont (lentement) transférés d'un système à un autre.

Au sens large le système d'exploitation ne se réduit pas au noyau. Il inclut les bibliothèques qui l'entourent et sont essentielles pour son utilisation : compilateur, chargeur, éditeur, *etc.*

Linux *ou* Gnu-Linux comme revendiqué par Richard Stallman ?

Quel système d'exploitation ?

pour le cours.

Unix, Linux, Mach... Windows ?

En principe, le choix importe peu

En pratique, Windows est exclu (sans sources)

pour le cours.

Unix, Linux, Mach... Windows ?

En principe, le choix importe peu

En pratique, Windows est exclu (sans sources)

De plus, Unix/Linux

- partagent les mêmes concepts avec des différences minimes.
- sont les plus répandus (après Windows),
- offrent un libre accès aux sources,
- présentent une architecture relativement simple,
- Unix garde une importance historique dans le domaine des OS.

- Linux $\approx$ Unix (différents noyaux mais bibliothèques similaires)
- Normes POSIX : rend les différents Unix compatibles
  (mêmes appels systèmes)

Pour programmer le système

Le langage d'implémentation d'un système est presque exclusivement C (même si des expériences ont été menées avec d'autres langages), car il combine :

- Une sémantique assez claire (comparé à Java, C++),
- Une très bonne efficacité,
- Un langage qui permet l'accès à (presque) toutes les capacités de la machine (opérations sur les bits, etc.)
- Allocation explicite, qui est (encore) la règle pour les systèmes d'exploitation.

Quel langage de programmation ?

Pour programmer le système

- C est le langage de choix pour programmer un système d'exploitation.
- Il s'impose pour étendre un système existant écrit en C.
- Mais ce ne sera pas forcément toujours le cas...
  Besoin de prouver la correction des programmes (pour des systèmes embarqués)

Quel langage de programmation ?

Pour programmer le système

- C est le langage de choix pour programmer un système d'exploitation.
- Il s'impose pour étendre un système existant écrit en C.
- Mais ce ne sera pas forcément toujours le cas...
  Besoin de prouver la correction des programmes (pour des systèmes embarqués)

Pour interagir avec le système

- L'interface *primitive* avec le système est une librairie C.
- Les langages généraux raisonnables s'interfacent avec C et relèvent les appels systèmes essentiels.
- Un langage de haut niveau permet de redéfinir une interface plus claire et plus sûre, voir plus générique.
- Une application *ne fait pas qu'*interagir avec le système : un langage de haut niveau bénéficie à tout le programme.

Quel langage de programmation ?

Pour programmer le système

- C est le langage de choix pour programmer un système d'exploitation.
- Il s'impose pour étendre un système existant écrit en C.
- Mais ce ne sera pas forcément toujours le cas...
  Besoin de prouver la correction des programmes (pour des systèmes embarqués)

Pour interagir avec le système

- OCaml relève les appels systèmes essentiels, avec une interface de plus haut niveau que C.
- Ce qui assure plus de sécurité à la fois pour la partie système et le reste du programme.

(Description approximative, minimale)

Un programme qui tourne possède :

- Un *environnement d'exécution*, fixé par le système au lancement du programme, consultable et modifiable par le programme.
- Des *variables-système* modifiables seulement par le système (afin de préserver des invariants), à sa propre initiative ou sur demande du programme.
  Par exemple : `umask`, `uid`, `gid`, `cwd`, *etc.*
- Des *descripteurs* (fichiers, connexions) également gérés par le système.

OCaml et le système d'exploitation

La librairie `Pervasives` fournit les fonctions d'entrée-sortie tamporisées ainsi que quelques appels système. Par exemple :

```
exit : int -> 'a
```

Les librairies `Sys` (préchargée) et `Unix` relèvent la plupart appels système.

Pour les utiliser

1. Commencer le fichier source `foo.ml` par :

   ```
   open Sys
   open Unix
   ```

2. Pour compiler avec la librairie `Unix` :

   ```
   ocamlc -o foo unix.cma foo.ml
   ```

   ou bien

   ```
   ocamlc -o foo -c foo.ml
   ocamlc -o foo unix.cma foo.cmo
   ```

Interface minimale avec le programme appelant

Les arguments de la ligne de commande

```
Sys.argv : string array
```

L'environnement d'exécution

```
Unix.environment : unit -> string array
Unix.getenv : string -> string array
```

Le code de retour

```
Pervasives.exit : int -> 'a
```

Un retour normal exécute `exit 0`

Une exception levée non-rattrapée appelle `exit 2`.

Pour installer un postlude à un programme :

```
at_exit : (unit -> unit) -> unit
```

Le reste (cours) : entrées-sorties, signaux, tuyaux, prises, . . .

Origine

Une erreur pendant un appel système (une opération avec le système) interrompt celui-ci et est indiquée au programme appelant avec la source de l'erreur.

Signalement

En C, une erreur correspond à un code de retour de −1 et le type de l'erreur est codée dans une variable globale `errno`. Il faut donc tester systématiquement le code de retour d'un appel système.

En OCaml, une erreur lors d'un appel système lève l'exception :

```
exception Unix_error of error * string * string
```

L'erreur est décrite par un type énuméré :

```
type error = E2BIG | EACCESS | ... EUNKNOWNERR of int
```

Récupération des erreurs

Avec la construction

```
try ... with Unix_error (EACCESS, _, _) | ...
```

Le filtrage peut être sélectif en fonction du type de l'erreur.

Erreurs fatales

```
handle_unix_error : ('a -> 'b) -> 'a -> 'b
```

`handle_unix_error` f a évalue f a et si une erreur est levée affiche un rapport d'erreur et termine le programme avec le code 2.

Utilisé typiquement sur la fonction principale.

Erreur non fatales

C'est toujours le cas dans les fonctions de librairie.

Code de finalisation

Certains appels systèmes ouvrent des structures système qu'il faudra refermer ensuite, que le calcul se termine normalement ou de façon erronée. Le code de fermeture est appelé finalisation.

La construction `try ... finalize` est programmable.

```
let try_finalize f x finally y =
  let res = try f x with exn -> finally y; raise exn in
  finally y;  res
```

Utilisation typique

```
let desc = openfile () in
let treat arg = (* code de traitement utilisant desc *) in
try_finalize treat arg close desc
```

Le système de fichiers

Le système de fichiers

Il permet de stocker les données de façon persistante.

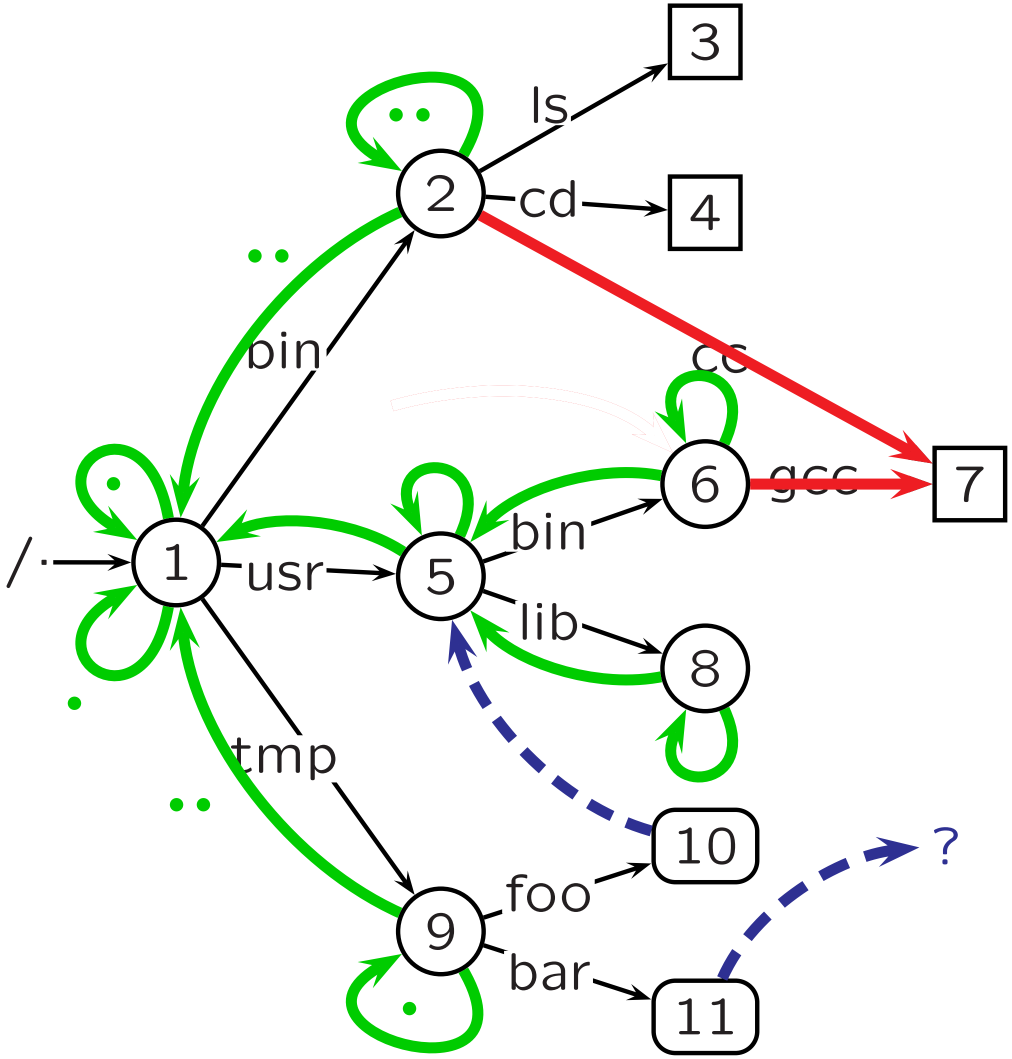
Un programme ne voit pas directement les informations telles qu'elles sont stockées sur le disque, mais par l'intermédiaire du système de fichiers qui lui en donne une vue abstraite :

- les détails d'implémentation sont cachés,
- la présentation à l'utilisateur est indépendante de leur représentation sur machine,
- le système peut préserver des invariants (car l'utilisateur n'a pas accès aux détails de la représentation).

Plan

- Vue abstraite de la hiérarchie
- Exploration et modification de la hiérarchie.
- Lecture-écriture sur les fichiers réguliers (prochain cours...)

Un ~~arbre~~ graphe acyclique + pointeurs arrières
liens symboliques



liens durs

interdit

liens inverses

liens symboliques

Lien dur sur un répertoire

Chemins de 9 à 8 ?

- ../usr/lib
- ./../usr/lib, *etc.*
- foo/lib

Système de fichiers (graphe)

Le système de fichiers est un graphe :

- Un seul point d'entrée appelé la racine, désigné par /.
- Les arcs sont étiquetés par des noms.
- Deux arcs issus d'un même nœud ont des étiquettes différentes.
- Un noeud non-orphelin est appelé répertoire et ne peut avoir qu'un seul parent. Il a toujours au moins deux fils, lui-même et son parent, qui sont étiquettés par `.` et `..` respectivement.

Chaque nœud peut donc être désigné de façon univoque au moins par un chemin à partir de la racine.

Chaque nœud a un numéro unique qui permet la comparaison (égalité physique)—en fait deux numéros `device` et `inode`.

Système de fichiers (chemins)

Un chemin relatif est une liste de noms de fichiers.

On peut le représenter par une chaîne caractères, en séparant les noms de fichiers par le caractère spécial /, qui ne peut pas apparaître dans les noms de fichiers.

Pour interpréter un chemin, il faut préciser un nœud de départ, en général, déterminé par le contexte.

Exemples : `local/bin` et `../tmp/foo`

Un chemin absolu est un chemin relatif interprété à partir de la racine. On le note en le précédent d'un / (qui désigne la racine).

Exemples : `/bin/ls` et `/home/remy/cours/system/1/`

Un / supplémentaire à la fin des chemins menant à des répertoires est toléré.

Position dans la hiérarchie

Un programme (et plus généralement un processus Unix a une position courante dans la hiérarchie à partir de laquelle sont évalués les liens relatifs).

```
getcwd : unit -> string
chdir : string -> unit
```

L'orgine de la hiérarchie

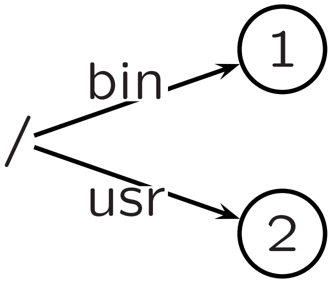
Pour information plus que pour usage...

L'origine de la hiérarchie n'est pas forcément celle que l'on pense : le système peut tromper le programme et ne lui montrer qu'une vue élaguée (faire `man 2 chroot`).

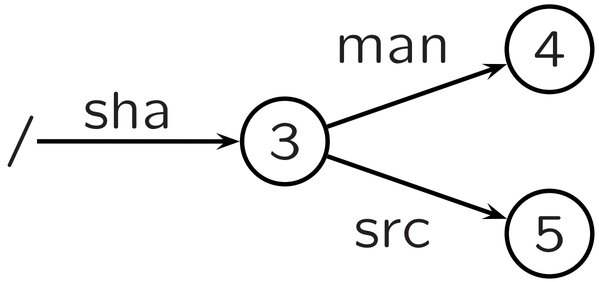
Monter une hérarchie dans une autre

Avant montage

Gauche

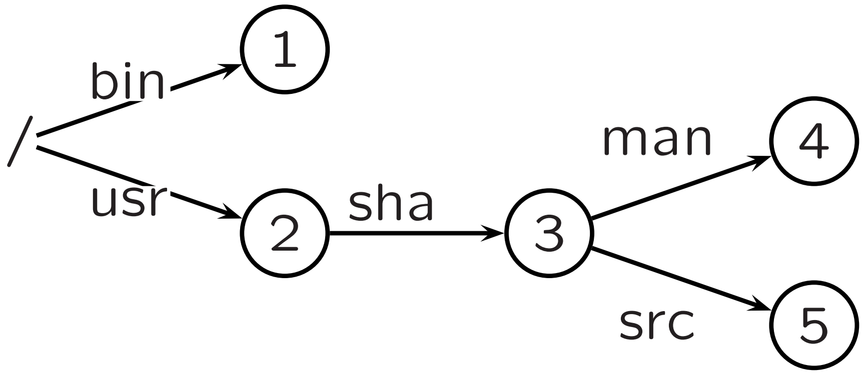


Droite



Après montage

de la hiérarchie droite à l'adresse /usr de la gauche



Les chemins en OCaml

Pour être paramétrique en fonction des conventions du système (Windows, Unix), OCaml définit les opérations sur les fichiers de façon abstraite dans le module `Filename`.

```
current_dir_name : string
parent_dir_name : string
concat : string -> string -> string
```

Pour décomposer les chemins, on peut utiliser les fonctions :

```
dirname : string -> string
basename : string -> string
```

Pour chaque chemin p, absolu ou relatif, `dirname` p et `filename` p retournent un chemin d et un nom b tel que d/n équivaut à p.

NB : Les calculs sur les chemins ne font pas d'appels systèmes.

Exercice

Remplir la table :

p	`dirname` p	`basename` p
`foo/bar`	`foo`	`bar`
`foo`	`.`	`foo`
`.`	`.`	`.`
`foo/.`	`foo`	`.`

Est-ce que `foo/.` est égale à `./foo` ?

Si et seulement si `foo` est si un répertoire.

Types de fichiers

Selon leur contenu :

- Les répertoires, seuls à avoir des fils, n'ont pas de contenu.
- Les fichiers ordinaires : leur contenu est quelconque (suite de caractères).
- Les liens symboliques : leur contenu est un chemin, relatif ou absolu, qui peut ou non désigner un autre nœud.
  Un lien symbolique est normalement intéprété comme un pointeur dans le graphe qu'il faut suivre.
- Les autres cas vus plus loin dans le cours :
  - Les tuyaux, les prises.
  - Les fichiers spéciaux de type caractère ou de type bloc.

En Unix, la communication avec des périphériques se fait par des fichiers spéciaux. Le programme n'a pas besoin, en général, de savoir qu'il s'agit d'un fichier spécial (exemple : impression).

Types de fichiers en OCaml

Le type concret énuméré `file_kind` décrit les différents types de fichiers :

`S_REG`	fichier normal
`S_DIR`	répertoire
`S_LNK`	lien symbolique
`S_CHR`	fichier spécial de type caractère
`S_BLK`	fichier spécial de type bloc
`S_FIFO`	tuyau
`S_SOCK`	prise

Liens symboliques

Les liens symboliques sont interprétés comme des pointeurs.

On les suit lors d'une recherche dans la hiérarchie des fichiers :

- Si p/f est un lien symbolique absolu $/q$ alors p/f désigne $/q$.
- Si p/f est un lien symbolique relatif q alors p/f désigne p/q.

Lorsqu'un lien symbolique f apparaît dans un chemin $p/f/q$,
la seule intéprétation possible est de remplacer p/f par le nœud qu'il désigne, qui doit être un répertoire.

Lorsqu'un lien symbolique f apparaît à la fin d'un chemin p/f,
on peut interpréter ce chemin comme le lien lui-même ou comme l'objet désigné par le lien : le contexte doit alors préciser quelle interprétation choisir.

Vue acyclique de la hiérarchie

Les répertoires “.” et “..” permettent de remonter à la racine à partir de n’importe quel nœud.

On peut donc atteindre tous les nœuds de l’arbre à partir de n’importe quel autre (si on a les droits d’accès).

Les liens “.” et “..” créent des cycles.

Les liens symboliques permettent des racourcis. Ils peuvent également créer des cycles.

Un parcours récursif

- doit toujours éviter les répertoires “.” et “..”
- va en général éviter les liens symboliques. Sinon, il faut limiter le nombre maximal de liens symboliques traversés récursivement.

Glanage de fichiers...

Le recyclage des nœuds d'une hiérarchie peut se faire par compteur de référence, car (une lecture de) la hiérarchie est acyclique.

Une destruction de fichier ou de dossier vide se fait en O(1). Il faut toujours vider les répertoires avant de pouvoir les effacer !

Les liens symboliques sont ignorés pour le comptage des références. On peut donc détruire un fichier alors qu'il existe quelque part dans la hiérarchie un lien symbolique qui le désigne. (Tout comme on peut créer un lien symbolique vers un fichier inexistant.)

Opérations sur les fichiers

Différents types d'opérations

- Consultation d'un nœud (méta-information)
- Parcours de la hiérarchie (en tout sens...)
- Modification de la hiérarchie (création, destruction de nœuds).
- Écriture/Lecture du contenu des fichiers.

Selon les différents types de fichiers

Ces opérations dépendent du type de fichiers :

- Fichiers ordinaire : lecture, écriture
- Répertoires : lecture, ajout.
- Création, destructions de nœuds (liens durs, symboliques, etc.)
- Répertoires et fichiers sont de taille variable, pouvant être grande. On les lits petits à petit en utilisant des descripteurs.

Méta-informations sur les fichiers

Identification

Le numéro du périphérique (device) sur lequel il est.

Le numéro du nœud sur ce périphérique.

Type du fichier

Taille

Dates de modifications

(a) dernier accès

(m) dernière modification du contenu

(c) dernier modification du contenu ou des méta-informations.

Droits d'accès

Propriétaires et Groupes

Les utilisateurs appartiennent à un ensemble de groupes d'utilisateurs.

Les utilisateurs et les groupes d'utilisateurs sont identifiés par un numéro, appelé user-id (`uid`) et group-id (`gid`).

La liste des utilisateurs et des groupes est dans une base de données du système (traditionnellement les fichiers `/etc/passwd` et `/etc/group`. On peut la consulter avec les fonctions :

```
getpwnam: string -> passwd_entry
getgrnam: string -> group_entry
```

```
getpwuid: int -> passwd_entry
getgrgid: int -> group_entry
```

Un programme qui s'exécute a un `uid` et un `gid`, consultables :

```
getuid : unit -> int
getgid : unit -> int
```

Droits d'accès

autorisation en lecture (r), écriture (w), ou exécution (x)
se note `rwx` qui représente un nombre octal (1 bit par lettre).

propriétaires

- l'utilisateur propriétaire du fichier,
- le groupe propriétaire du fichier ou
- tout le monde.

Codés sur un entier par l'union des permissions individuelles :

$$\underbrace{\overbrace{\_\ \_\ \_}^{user(U)}\ \overbrace{\_\ \_\ \_}^{group(G)}\ \overbrace{\_\ \_\ \_}^{other(O)}}_{\texttt{0oUGO}}$$

$+$ Les bits s et le bit t

Exemple : Que signifie `0o644` ?

Informations sur un fichier en OCaml


```
stat : string -> stats
lstat : string -> stats
```

où `stats` est un type enregistrement :

`st_dev : int`	Identificateur de la partition	identité dans la hiérarchie
`st_ino : int`	Identificateur du fichier dans la partition	
`st_kind : file_kind`	Type du fichier.	
`st_perm : int`	Droits d'accès au fichier	
`st_nlink : int`	Nombre d'entrées pour un répertoire ou nombre de liens durs pour un fichier.	
`st_uid : int`	Le numéro de l'utilisateur propriétaire.	
`st_gid : int`	Le numéro du groupe propriétaire.	
`st_rdev : int`	(pour les fichiers spéciaux).	
`st_size : int`	La taille du fichier, en octets.	
`st_atime : int`	La date du dernier accès au contenu du fichier.	
`st_mtime : int`	La date de la dernière modification du contenu	
`st_ctime : int`	La date du dernier changement de l'état.	

Droit d'accès dynamiques

Un programme qui tourne peut tester s'il a le droit d'accès à un fichier.

```
access : string -> access_permission list -> unit
```

où les permissions élémentaires sont décrite par le type :

```
type access_permission = R_OK | W_OK | X_OK | F_OK
```

L'accès, calculé dynamiquement peut être plus restreint qu'il ne paraît dans l'information statique retournée par `stat`. Pourquoi ?

Modifier les droits d'accès

Les droits

```
chmod : string -> int -> unit
```

Les propriétaires

On peut changer le propriétaire (uid) et le groupe (gid) avec `chgrp`

```
chown : string -> int -> int -> unit
```

Lecture d'un répertoire en OCaml

On lit les répertoires en deux étapes :

1. On ouvre un descripteur sur le répertoire.

```
opendir : string -> dir_handle
```

2. On lit les entrées une par une.

```
readdir : dir_handle -> string
```

3. La lecture lève l'exception `End_of_file` à la fin.

Exemple (attention à bien refermer le répertoire ouvert)

```
let iter_dir f d =
  let dir_handle = opendir d in
  try while true do f (readdir dir_handle) done with
    End_of_file -> closedir dir_handle
  | x -> closedir dir_handle; raise x;;
```

Exercice : `fold_dir`, `list_of_dir`

Exemple : lister un répertoire

```
let ls = iter_dir print_endline;;
let ls_i d =
  iter_dir
  (fun s ->
    print_int (lstat (Filename.concat d s)).st_ino;
    print_char '\n')
  d;;
```

En faire un programme :

```
let ls() =
  for i = 1 to Array.length Sys.argv - 1
  do ls Sys.argv.(i) done in
handle_unix_error ls();;
```

Un lien symbolique

```
readlink : string -> string
```

Un fichier régulier (cours suivant)

Pas à pas On ne peut créer un nœud à un chemin p/f que si le nœud p existe et est un répertoire.

Bien entendu, il faut aussi que le programme ait les droits nécessaires.

Le masque de création

Un programme a une variable système `umask` qui est utilisée lors de la création des fichiers et répertoires. Un argument `perm` de type `file_perm` est alors combiné avec le masque en prenant `perm & ~umask` pour valeur des droits d'accès à la création.

Pour lire et/ou modifier la variable système `umask` :

```
umask : file_perm -> file_perm
```

Le nouveau masque est installé et l'ancien est retourné.

Un répertoire vide

```
mkdir : string -> file_perm -> unit
```

Le répertoire ne doit pas déjà exister.

Un lien dur

```
link : string -> string -> unit
```

`link` *source* *dest* crée le chemin *dest* (qui ne doit pas existé) vers le fichier *source* (qui doit déjà exister). Auncun nœud n'est créé.

Un lien symbolique

```
symlink : string -> string -> unit
```

`symlink` *source* *dest* créer le fichier *dest* qui est un lien symbolique vers *source* (existant ou non).

Création d'un fichier

Voir le cours suivant.

Détruire

La destruction d'un nœud a deux effets :

- rendre immédiatement le nœud invisible dans la hiérarchie,
- libérer (pour recyclage) lorsque plus aucun programme ne l'utilisera (peut être longuement retardé). Application

Un répertoire vide

```
rmdir : string -> unit
```

Un fichier (régulier, lien, etc)

```
unlink : string -> unit
```

`unlink` *source* détruit le dernier arc du chemin source. Le nœud désigné par source n'est détruit que s'il n'a plus de parents.

Renommer


```
rename : string -> string -> unit
```

La commande `rename` *source* *dest* permet de déplacer le nœud (existant) *source* à la position *dest*.

- *source* peut être un fichier ou un répertoire.
- si *dest* existe, il est écrasé (sauf si c'est un répertoire non vide).

Prochain cours