Système et programmation système

Julien BERNARD

Université de Franche-Comté – UFR Sciences et Technique
Licence Informatique – 2è année

2015 – 2016

Première partie

Généralités

Plan de ce cours

1. Introduction
   - À propos de votre enseignant
   - À propos du cours Système
2. Système d'exploitation
   - Qu'est-ce qu'un système d'exploitation ?
   - Unix et GNU/Linux
3. Utilisation du système
   - Interpréteur de commande (shell)
   - Pages de manuel

Plan

Votre enseignant
Qui suis-je ?

Qui suis-je ?

Julien BERNARD, Maître de Conférence (enseignant-chercheur)
julien.bernard@univ-fcomte.fr, Bureau 426C

Enseignement

- Responsable du semestre 1 (Starter) de la licence Informatique
- Cours : Publication web et scientifique (L1), Algorithmique (L2), Système (L2), Sécurité (L3)

Recherche

Optimisation dans les réseaux de capteurs

Plan

UE Système
Organisation

Équipe pédagogique

- Julien Bernard : CM, TD (julien.bernard@univ-fcomte.fr)
- Éric Merlet : TP (eric.merlet@univ-fcomte.fr)

Volume

- Cours : 12 x 1h30, lundi 11h00
- TD : 12 x 1h30, mardi 9h30 (Gr. 2) et 11h00 (Gr. 1)
- TP : 12 x 1h30

Évaluation

- 2 devoirs surveillés
- 2 projets en TP : projet shell, projet C

UE Système
Comment ça marche ?

Mode d'emploi

1. Pas de cours en ligne
2. Prenez des notes ! Posez des questions !
3. Le TD n'est pas l'application du cours !
4. Comprendre plutôt qu'apprendre
5. Le but de cette UE n'est pas d'avoir une note !

Niveau d'importance des transparents

	trivial	pour votre culture
$\star$	intéressant	pour votre compréhension
$\star\star$	important	pour votre savoir
$\star\star\star$	vital	pour votre survie

Note : les contrôles portent sur *tous* les transparents !

UE Système
Contenu pédagogique

Objectif

Comprendre et manipuler les principes de base d'un système de type Unix

- Système d'exploitation
- Shell et ligne de commande
- Programmation C
- Fichiers
- Processus
- Mémoire virtuelle

UE Système
Bibliographie


Andrew Tanenbaum.
Systèmes d'exploitation.
3è édition, 2008, Pearson


Christophe Blaess
Développement système sous Linux.
3è édition, 2011, Eyrolles.

Plan

Système d'exploitation
Définition

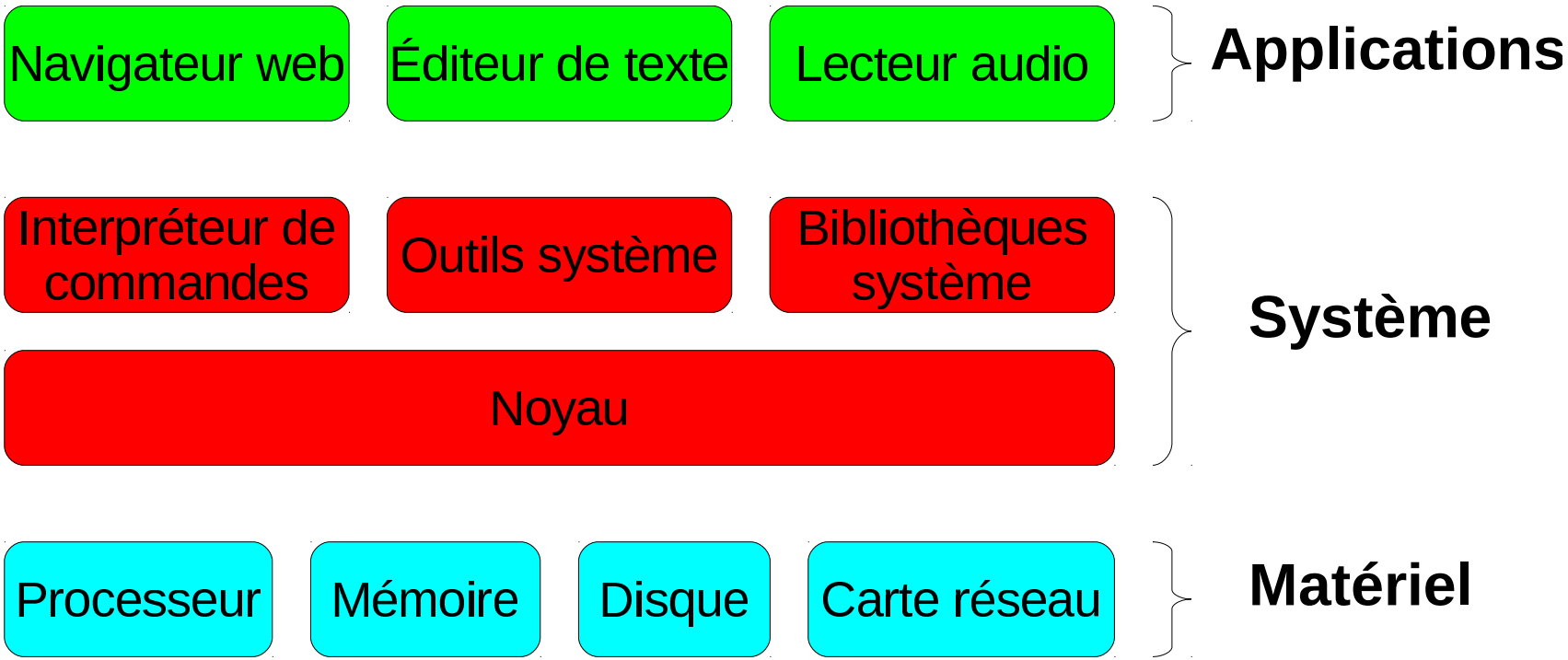
Définition (Système d'exploitation)

Le système d'exploitation, abrégé SE (en anglais operating system, abrégé OS), est l'ensemble de programmes central d'un appareil informatique qui se place à l'interface entre le matériel et les logiciels applicatifs. (Source : Wikipedia)

- permet de libérer l'utilisateur de la complexité de la programmation du matériel
- propose une gestion flexible et optimisée de ses différents composants (processeur, affichage, stockage)

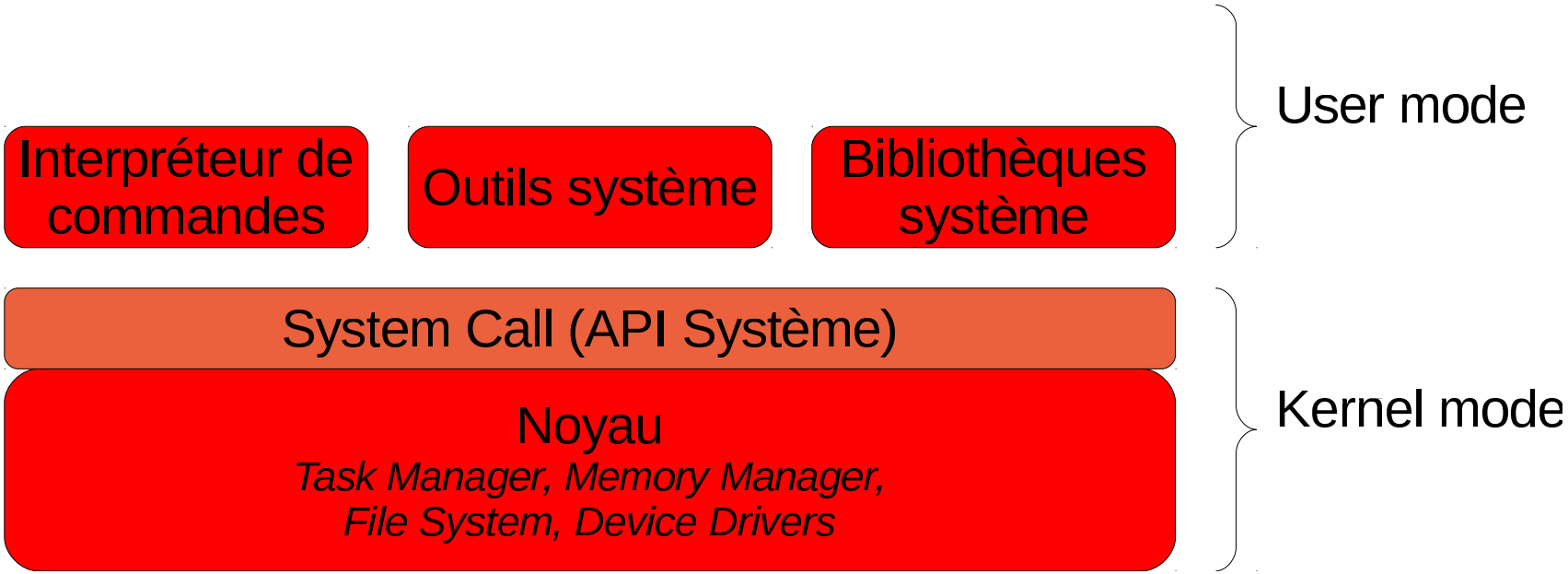
Système d'exploitation

Entre les applications et le matériel



Système d'exploitation

Noyau et bibliothèque système



Système d'exploitation
Typologie

- *multi-tâches* : exécution simultanée de plusieurs programmes
- *multi-utilisateurs* : utilisation simultanée par plusieurs usagers
- *multi-processeurs* : capable d'exploiter plusieurs processeurs
- *temps réel* : temps d'exécution des programmes garanti

Exemples

- DOS : mono-utilisateur, mono-tâche
- Windows 95, OS/2 : mono-utilisateur, multi-tâches
- NT, Linux, Solaris : multi-tâches, multi-utilisateurs, multi-processeurs
- QNX : multi-tâches, multi-utilisateurs, temps réel
- Symbian OS : multi-tâches, mono-utilisateur, temps réel

Plan

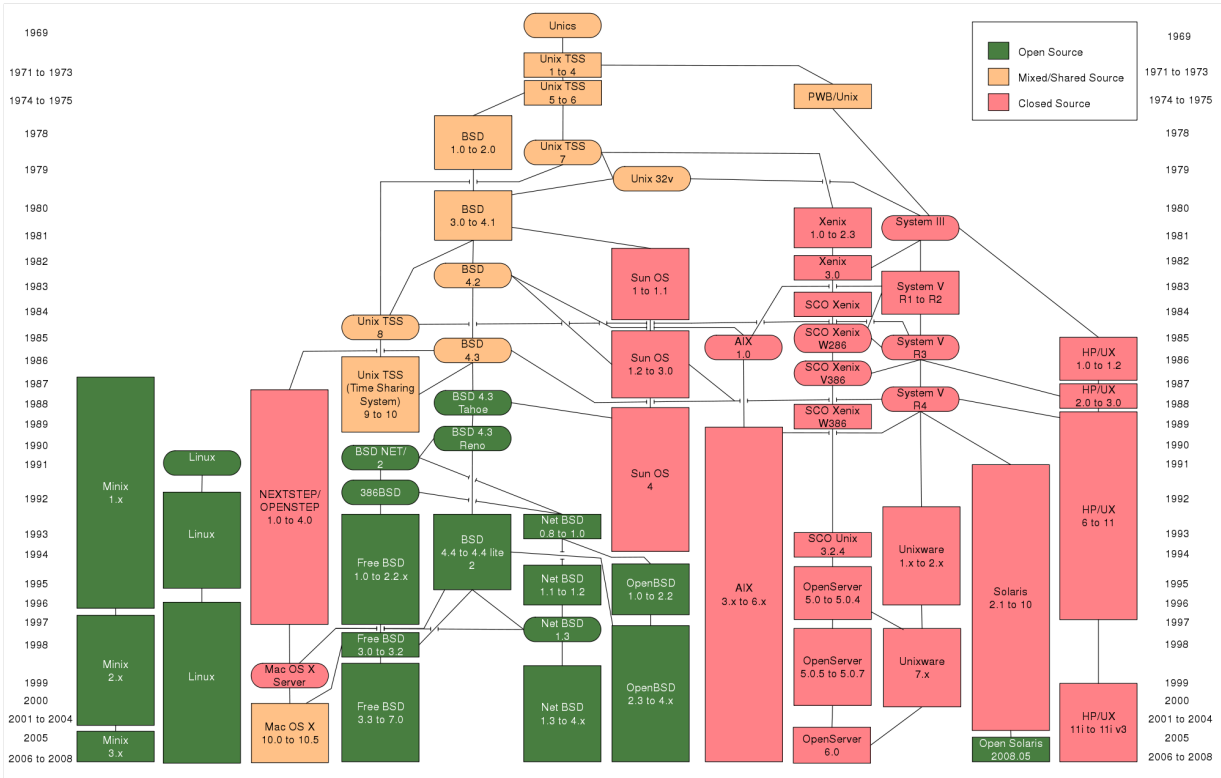
Unix
Quelques dates-clefs

- 1969 : Création d'Unix, Bell Labs (AT&T)
  Ken Thompson, Dennis Ritchie, Brian Kernighan
- 1973 : Création de C et réécriture d'Unix en C (portabilité)
- 1977 : 1BSD (Berkeley Software Distribution), Bill Joy
- 1982 : Sortie de AT&T System V
- 1982 : Création de Sun Microsystems, Bill Joy
- 1983 : Création de GNU (GNU's Not Unix), Richard Stallman
- 1984 : Spécifications X/Open
- 1988 : Spécifications POSIX
- 1991 : SunOS (BSD) devient Solaris (SVR4)
- 1991 : Création de Linux, Linus Torvalds
- 1994 : Sortie de 4.4BSD
- 1995 : Création d'OpenBSD, Theo De Raadt
- 2001 : Mac OS X, Apple, Steve Jobs

Unix
La jungle des Unix

Exemples

- Propriétaire : IBM AIX, Sun Solaris, HP/UX, Mac OS X
- Libre : Linux, FreeBSD, OpenBSD, NetBSD, Minix



Unix
Philosophie

Définition (Philosophie d'Unix, 1994, Mike Gancarz)

1. La concision est merveilleuse.
2. Écrivez des programmes qui font une seule chose mais qui le font bien.
3. Concevez un prototype dès que possible.
4. Préférez la portabilité à l'efficacité.
5. Stockez les données en ASCII.
6. Utilisez le levier du logiciel à votre avantage.
7. Utilisez les scripts shell pour améliorer l'effet de levier et la portabilité.
8. Évitez les interfaces utilisateur captives.
9. Faites de chaque programme un filtre.

GNU/Linux

Qu'est-ce que c'est ?

Dates importantes

- 1983 : Création de GNU (GNU's Not Unix), Richard Stallman
- 1984 : Création de la FSF (Free Software Foundation)
- 1991 : Création de Linux, Linus Torvalds

But

Offrir un système d'exploitation libre compatible POSIX

Composants

- GNU : outils système, bibliothèques système
- Linux : noyau

GNU/Linux

Distributions

Définition (Distribution)

Une distribution GNU/Linux est un ensemble cohérent de logiciels (libres) assemblés autour du système d'exploitation GNU/Linux. (Source : Wikipedia)

- Distributions généralistes vs Distributions spécifiques
- Distributions communautaires vs Distributions commerciales

Exemples

- Debian, Ubuntu
- Red Hat, Fedora, SUSE, OpenSUSE
- Mageia, Arch Linux, Gentoo, Slackware

GNU/Linux

Paquetage

Définition (Paquetage)

Un paquetage est une archive comprenant les fichiers informatiques, les informations et procédures nécessaires à l'installation d'un logiciel sur un système d'exploitation. (Source : Wikipedia)

Caractéristiques

- Méta-données : description, version, dépendances
- Gestionnaire de paquetage : cohérence fonctionnelle du système
- Paquet source vs Paquet binaire

Exemples (Format de paquetage)

- `deb`, dpkg, apt/aptitude
- `rpm`, RPM, yum/zypper/urpmi

Plan

Interfaces
GUI vs CLI

★★

Interface graphique (Graphical User Interface)

- Paradigme WIMP : «window, icon, menu, pointing device»
- Métaphore du bureau
- Communication par mouvement

Interpréteur de commande (Command Line Interface)

- Mode texte
- Avantages :
  - Automatisation des tâches
  - Contrôle à distance
  - Faible consommation de ressources
- Communication par langage

Shell Unix
Familles

sh et dérivés

- `sh` : Bourne SHell, 1977, Steve Bourne, Version 7 Unix
- `ksh` : Korn SHell, 1983, David Korn
- `bash` : Bourne-Again SHell, 1987, Brian Fox, GNU
- `ash` : Almquist SHell, 1989, Kenneth Almquist
- `zsh` : Z SHell, 1990, Paul Falstad
- `dash` : Debian Almquist SHell, 1997, Herbert Xu, Debian

csh et dérivés

- `csh` : C SHell, 1978, Bill Joy, BSD
- `tcsh` : TENEX C SHell, 1979, Ken Greer

Interpréteur de commandes

Principes

★★★

Fonctionnement d'un interpréteur de commandes

1. Attente de la commande avec le prompt ($ ou #)
2. Saisie de la commande
3. Analyse puis exécution de la commande avec affichage du résultat

Format des commandes

- Commande élémentaire :
  `$ commandes [options] [fichiers_ou_données]`
- Succession de commandes :
  `$ commande1; commande2`

Exemple

```
$ echo -n "Bonjour, nous sommes le "; date
```

Interpréteur de commandes

Erreurs courantes

★★

Exemples

- Commande introuvable

```
$ cag
cag: not found
```

- Aucun fichier ou dossier de ce type

```
$ cat inconnu.txt
cat: inconnu.txt: No such file or directory
```

- Option non valide

```
$ cat -w
cat: invalid option - 'w'
```

- Permission non accordée

```
$ cat /etc/shadow
cat: /etc/shadow: Permission denied
```

Interpréteur de commandes

Commandes de base

★★

Exemples

- `$ whoami`
  Afficher l'identifiant d'utilisateur
- `$ uname [-snrvma]`
  Afficher des informations sur le système
- `$ uptime`
  Indiquer depuis quand le système a été mis en route
- `$ date`
  Afficher ou configurer la date et l'heure du système
- `$ cal [[month] year]`
  Afficher un calendrier
- `$ echo [-n] [string]`
  Afficher une ligne de texte

Plan

Page de manuel

Qu'est-ce que c'est ?

★★★

Définition (Page de manuel)

Une page de manuel (manpage) est une documentation complète relative à un programme, une fonction, un fichier, etc. Les pages de manuel sont rangées dans différentes sections. Le nom d'une page de manuel est de la forme `page(section)`. RTFM !

Commande `man`

```
$ man [[section] page]
```

Afficher la page de manuel `page` de la section `section`

Exemple

```
$ man man
```

Afficher la page de manuel `man(1)` de la commande `man`

Page de manuel
Sections de pages de manuel

★★

Sections des pages de manuel

1. Commandes utilisateur
2. Appels système
3. Fonctions de bibliothèque
4. Fichiers spéciaux
5. Formats de fichier
6. Jeux
7. Divers
8. Administration système
9. Interface du noyau Linux

Page de manuel

Exemple : `$ man uptime`

```
UPTIME(1)                  Linux User's Manual                 UPTIME(1)



NAME
       uptime - Tell how long the system has been running.

SYNOPSIS
       uptime
       uptime [-V]

DESCRIPTION
       uptime gives a one line display of the following information.  The cur-
       rent time, how long the system has been running,  how  many  users  are
       currently  logged  on,  and the system load averages for the past 1, 5,
       and 15 minutes.

       This is the same information contained in the header line displayed  by
       w(1).

       System load averages is the average number of processes that are either
       in a runnable or uninterruptable state.  A process in a runnable  state
       is  either  using the CPU or waiting to use the CPU. A process in unin-
       terruptable state is waiting for some I/O access, eg waiting for  disk.
       The  averages  are  taken over the three time intervals.  Load averages
       are not normalized for the number of CPUs in a system, so a load  aver-
       age  of 1 means a single CPU system is loaded all the time while on a 4
       CPU system it means it was idle 75% of the time.

FILES
       /var/run/utmp
              information about who is currently logged on

       /proc  process information

AUTHORS
       uptime was written by Larry Greenfield <greenfie@gauss.rutgers.edu> and
       Michael K. Johnson <johnsonm@sunsite.unc.edu>.

       Please send bug reports to <albert@users.sf.net>

SEE ALSO
       ps(1), top(1), utmp(5), w(1)



Cohesive Systems              26 Jan 1993                       UPTIME(1)
```

Aide sur les commandes

`--help` et autres commandes

`--help`

Option disponible pour (presque) toutes les commandes. Exemple :

```
$ uptime --help
usage: uptime [-V]
    -V display version
```

Autres commandes d'aide

- `$ whatis [name]`
  Afficher la description des pages de manuel
- `$ apropos [keyword]`
  Chercher dans le nom et la description des pages de manuel
- `$ info [item]`
  Lire le document Info (alternative à `man`)

Deuxième partie

Système de fichiers

Plan de ce cours

4. Système de fichiers
   - Fichiers
   - Répertoires
   - Information et navigation
   - Utilisateurs et permissions
   - Inodes et structure sur le disque

5. Gestion de fichiers
   - Opérations de base
   - Liens

Plan

Qu'est-ce qu'un fichier ?

★★★

Définition (Fichier)

Un fichier est un lot d'*informations* portant un *nom*.

Tout est fichier

Sous Unix, tout est fichier :

- Les fichiers «physiques» ou fichiers réguliers
- Les fichiers «virtuels»
- Les répertoires
- Les périphériques (disques durs, souris, clavier, etc.)
- Les tubes (Voir le chapitre «Shell et scripts shell»)
- Les sockets (Voir le cours «Système et Réseau» en L3)

Opérations sur les fichiers

★★

Opérations

Sur tous les fichiers, on peut réaliser les opérations suivantes :

- Ouverture : prévient le système qu'on commence à utiliser un fichier
- Fermeture : prévient le système qu'on a terminé d'utiliser un fichier
- Lecture : lit des informations depuis un fichier
- Écriture : écrit des informations dans un fichier

Voir le chapitre «Manipulation de fichiers»

Nom de fichier

★★

Nom de fichier

Un nom de fichier est formé d'un *nom* décrivant généralement son contenu et d'une (ou plusieurs ou zéro) *extension* qui indique son type. L'ensemble du nom doit être inférieur à 255 caractères. Il y a une différence entre majuscule et minuscule !

Fichier caché

Un fichier dont le nom commence par '.' est un fichier caché.

Bonnes pratiques de nommage

- Utiliser uniquement des lettres, des chiffres et '_', '-', '.'
- Ne pas utiliser d'espace !
- Ne pas utiliser d'accent !
- Ne pas nommer deux fichiers de manière identique à la casse près !

Exemples de fichiers

Exemples

- `README`
  fichier texte (sans extension) souvent présent pour décrire le contenu du répertoire courant et des sous-répertoires
- `.bashrc`
  fichier (caché) de configuration de `bash` présent dans le répertoire utilisateur
- `systeme.pdf`
  le fichier qui me sert pour cette présentation
- `gmp-5.0.1.tar.bz2`
  archive (`.tar`) compressée à l'aide de `bzip2` (`.bz2`) des sources de la bibliothèque GMP (GNU Multiple Precision Arithmetic Library) en version 5.0.1 $\rightarrow$ il y a donc ici deux extensions et non quatre.

Plan

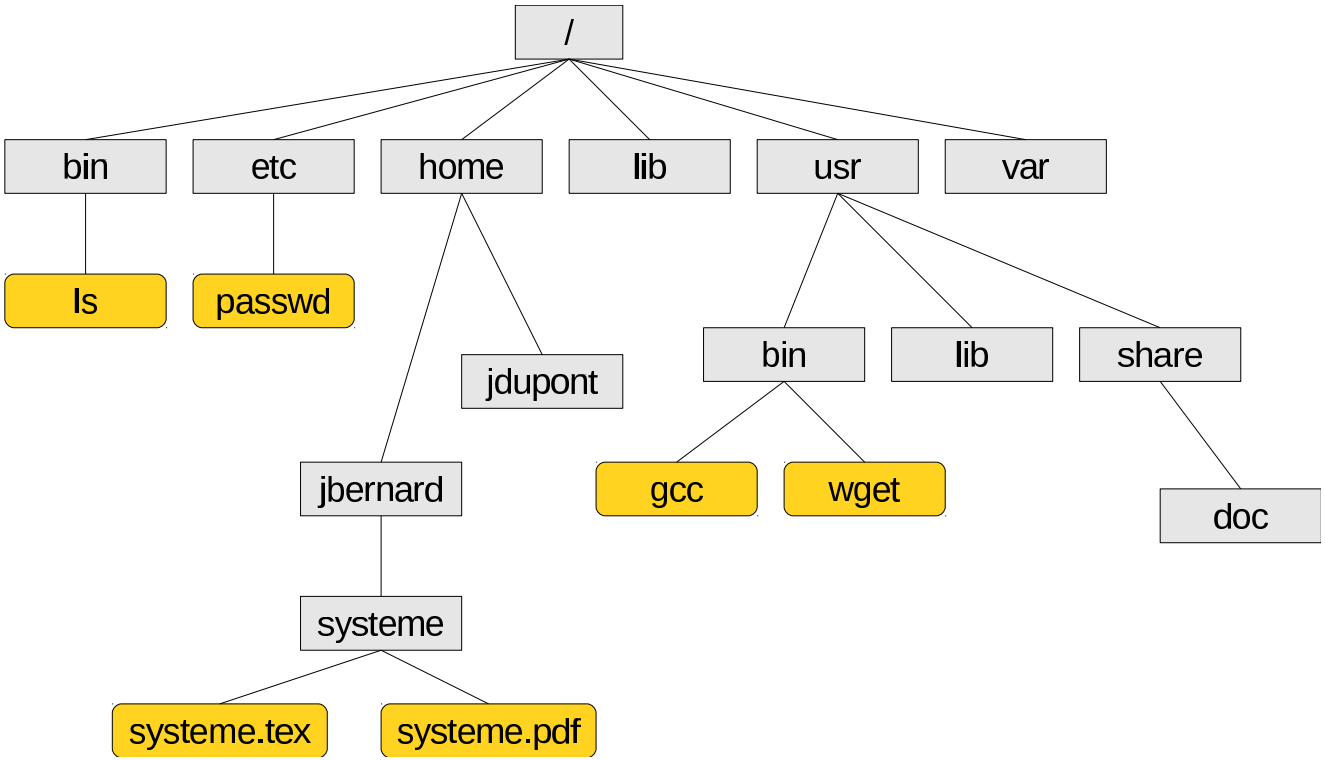
Qu'est-ce qu'un système de fichier ?

Généralités

★★★

Définition (Système de fichier)

Le système de fichier est un ensemble de fichiers et de répertoires (qui contiennent des fichiers et d'autres répertoires) organisés en arbre dont la racine est le répertoire /. Le séparateur de répertoire est /.



Filesystem Hierarchy Standard ⋆⋆

Définition (Filesystem Hierarchy Standard)

Le Filesystem Hierarchy Standard définit l'arborescence et le contenu des principaux répertoires des systèmes de fichiers de GNU/Linux et de la plupart des Unix. Exception notable : Mac OS X.

Répertoires dans /

- `/bin/` (*BINaries*) : commandes de base pour les utilisateurs
- `/boot/` : noyaux et chargeurs d'amorçage (*bootloader*)
- `/dev/` (*DEVice*) : fichiers de périphériques physiques et virtuels
- `/etc/` (*Editing Text Configuration*) : fichiers de configuration
- `/home/` : répertoires des utilisateurs
- ...

Filesystem Hierarchy Standard

★★

Répertoires dans /

- `/lib/` (*LIBrary*) : bibliothèques nécessaires pour `/bin` et `/sbin`
- `/mnt/` (*MouNT*) : point de montage temporaire
- `/opt/` (*OPTional*) : logiciels optionnels
- `/proc/` (*PROcess*) : informations sur les processus et le noyau
- `/root/` : répertoire de l'utilisateur `root`
- `/run/` : fichiers transitoires
- `/sbin/` (*System BINaries*) : commandes pour les administrateurs
- `/tmp/` (*TeMPorary*) : fichiers temporaires
- `/usr/` (*Unix System Resources*) : similaire à `/`
- `/var/` (*VARiable*) : fichiers variables

RTFM : `hier(7)`

Filesystem Hierarchy Standard

/dev/

Fichiers de périphériques physiques

- /dev/dsp : carte son
- /dev/fd0 : lecteur de disquettes
- /dev/sd* ou /dev/hd* : disques durs
- /dev/sr* : lecteurs CDROM/DVDROM
- /dev/tty* : terminaux

Fichiers de périphériques virtuels

- /dev/random : générateur de nombres (vraiment) aléatoires
- /dev/urandom : générateur de nombres (pas tout à fait) aléatoires
- /dev/null : «trou noir» (plutôt en écriture)
- /dev/zero : flux de zéros (plutôt en lecture)

RTFM : random(4), null(4)

Filesystem Hierarchy Standard

/etc/

/etc/

Quelques fichiers importants :

- /etc/passwd : informations sur les comptes utilisateurs
- /etc/shadow : mots de passe chiffrés
- /etc/group : groupes du système
- /etc/fstab : informations sur les systèmes de fichiers
- /etc/services : liste des services réseau Internet
- /etc/issue : message de bienvenue lors d'une connexion

RTFM : passwd(5), shadow(5), fstab(5), services(5), issue(5)

Filesystem Hierarchy Standard

/home/

★

Répertoire utilisateur

Le répertoire utilisateur est le répertoire où l'utilisateur stocke ses fichiers. Sous Unix, il est généralement situé dans le répertoire `/home`. C'est le répertoire dans lequel on se trouve après connexion. Exemple : le répertoire de l'utilisateur `jbernard` est `/home/jbernard/`

Notations

- `~/` (ou `$HOME`) est une notation pour votre répertoire utilisateur
- `~jdupont/` est une notation pour le répertoire utilisateur de Jean Dupont (généralement, `/home/jdupont/`)

Filesystem Hierarchy Standard

/usr/

/usr/

Dossiers non-nécessaires au fonctionnement minimal du système.

- /usr/bin/, /usr/lib/, /usr/sbin/ : idem que pour /
- /usr/include/ : entêtes des bibliothèques partagées (Voir le chapitre «Programmation en C»)
- /usr/share/ : fichiers indépendant de la plateforme, notamment :
  - /usr/share/man/ : emplacement des manpages.
  - /usr/share/doc/ : documentation du système
- /usr/local/ : comme /usr/ pour les données additionnelles
  - /usr/local/bin/, /usr/local/lib/, /usr/local/sbin/, etc.

Filesystem Hierarchy Standard

/var/

/var/

- /var/log/ : fichiers de journalisation
- /var/lock/ : fichiers de verrouillage
- /var/run/ : fichiers temporaires des logiciels en cours d'exécution
- /var/mail/ : boîtes aux lettres utilisateurs
- /var/spool/ : files d'attente des services
- /var/spool/mail/ : mails en cours de transit sur la machine
- /var/www/ : répertoire web

Répertoires spéciaux et chemin ★★★

Les répertoires spéciaux . et ..

Dans tous les répertoires, il existe deux répertoires spéciaux :

- . représente le répertoire courant ;
- .. représente le répertoire parent, c'est-à-dire le répertoire qui contient le répertoire courant.

Cas particulier de la racine /

Le répertoire parent de / est lui-même.

Définition (Chemin)

Le chemin (*path*) d'un fichier (ou d'un répertoire) est une chaîne de caractères décrivant sa position dans le système de fichier. On distingue :

- les chemins relatifs, c'est-à-dire par rapport au répertoire courant ;
- les chemins absolus, c'est-à-dire par rapport à la racine.

Exemples de chemin

Exemples

On considère que le répertoire courant est `/home/jbernard/`.

- `/bin/ls` est un chemin absolu vers la commande `ls`
- `./` est un chemin relatif vers le répertoire courant
- `../` est un chemin relatif vers le répertoire parent
- `../../bin/ls` est un chemin relatif vers la commande `ls`
- `./systeme/../../jbernard/../../jbernard/` est un chemin relatif (bizarre) vers le répertoire courant

Plan

Système de fichier
Information

Commandes d'information

- `pwd` : *Print Working Directory*
  Affiche le nom du répertoire courant
- `ls` : *LiSt*
  Fait la liste de tous les fichiers et répertoires du répertoire courant
  - `ls -a` : fait la liste de tous les fichiers, y compris les fichiers cachés
  - `ls -l` : fait la liste détaillée des fichiers avec :
    - les permissions liées à ce fichier
    - le nombre de «liens»
    - l'utilisateur et le groupe propriétaires du fichier
    - la taille du fichier
    - la date et l'heure de la dernière modification
    - le nom du fichier
- `ls [dir]` : *LiSt*
  Pareil que `ls` mais pour le répertoire `dir`

Système de fichier
Information

Exemple

```
$ pwd
/home/jbernard/Bazaar/jbernard/lectures/systeme/cours
$ ls -l
total 392
drwxr-xr-x 2 jbernard jbernard   4096  5 janv. 00:33 bits
drwxr-xr-x 2 jbernard jbernard   4096 16 janv. 12:51 contrib
-rw-r--r-- 1 jbernard jbernard    266 18 janv. 15:39 Makefile
drwxr-xr-x 4 jbernard jbernard   4096 18 janv. 17:18 media
-rw-r--r-- 1 jbernard jbernard 374865 19 janv. 16:20 systeme.pdf
-rw-r--r-- 1 jbernard jbernard   1957 19 janv. 11:35 systeme.tex
```

Système de fichier
Navigation

Commandes de navigation

- `cd [dir]` : *Change Directory*
  Se déplacer dans le répertoire `dir` (qui peut être relatif ou absolu)
- `cd` : *Change Directory*
  Se déplacer dans le répertoire utilisateur (équivalent à `cd ~`)

Exemple

```
$ cd
$ pwd
/home/jbernard
$ cd ..
$ pwd
/home
```

Plan

Fichiers, utilisateurs, groupes et permissions ⋆⋆⋆

Fichiers, utilisateurs, groupes et permissions

Chaque fichier (ou répertoire) du système de fichiers possède :

- un utilisateur et un groupe propriétaires du fichier
- un ensemble de permissions relatives à la lecture, l'écriture et l'éxécution du fichier

→ Comment s'organisent ces éléments les uns avec les autres ?

Utilisateurs et groupes

★★★

Utilisateur et UID

- Toute entité (personne physique ou programme) devant interagir avec un système Unix est authentifiée par un utilisateur (*user*).
- À chaque utilisateur sont associés un nom d'utilisateur et un numéro unique appelé numéro d'utilisateur (UID, *User ID*).
- La correspondance entre le nom d'utilisateur et le numéro d'utilisateur se trouve dans le fichier `/etc/passwd`.

Groupe et GID

- Chaque utilisateur fait partie d'un (ou plusieurs) groupe (*group*).
- À chaque groupe sont associés un nom de groupe et un numéro unique appelé numéro de groupe (GID, *Group ID*)
- La correspondance entre le nom de groupe et le numéro de groupe se trouve dans le fichier `/etc/group`.

Le super-utilisateur root

★★

Le super-utilisateur root

Il existe un utilisateur particulier appelé super-utilisateur ou root. Il possède tous les pouvoirs sur le système :

- Il peut lire, écrire et exécuter n'importe quel fichier.
- Il peut prendre l'identité de n'importe quel utilisateur.

L'utilisateur root a pour UID le numéro 0.

Le groupe root

Le groupe du super-utilisateur root se nomme également root et a pour GID le numéro 0. Généralement, seul l'utilisateur root fait partie du groupe root.

Permissions

★★★

Les trois permissions

Chaque fichier ou répertoire possède trois types de permissions :

- Lecture (`r`, *Read*)
  - Fichier : lire le contenu du fichier
  - Répertoire : lister le contenu du répertoire
- Écriture (`w`, *Write*)
  - Fichier : écrire le contenu du fichier
  - Répertoire : créer, supprimer, changer le nom des fichiers du répertoire
- Exécution (`x`, *eXecute*)
  - Fichier : exécuter le fichier
  - Répertoire : accéder au répertoire

Représentation symbolique

On représente les permissions par un triplet de lettres représentant la permission (`rwx`) ou un tiret (`-`) pour l'absence de permission.

Permissions

★★★

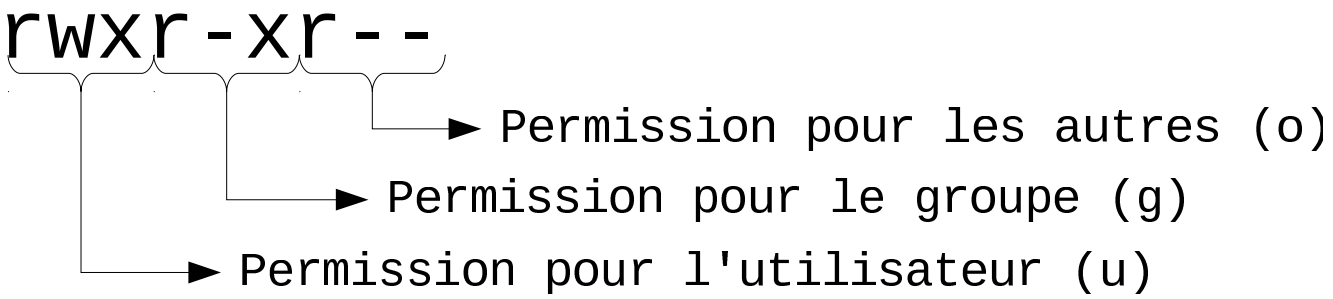
Application des permissions

Chaque permission peut s'appliquer à :

- L'utilisateur propriétaire du fichier (`u`, *User*)
- Le groupe propriétaire du fichier (`g`, *Group*)
- Les autres (`o`, *Other*)

Représentation symbolique

On représente les permissions associées à un fichier par trois triplets de lettres ou un tiret (-) pour l'absence de permission.



Permissions spéciales

Les permissions spéciales

- SUID (`s` ou `S`, *Set User ID*) : `--s------` ou `--S------`
  Permission d'exécution spéciale qui alloue les permissions de l'utilisateur propriétaire du fichier durant son exécution.
  `s` remplace `x` et `S` remplace `-`.
- SGID (`s` ou `S`, *Set Group ID*) : `-----s---` ou `-----S---`
  Permission d'exécution spéciale qui alloue les permissions du groupe du propriétaire du fichier durant son exécution.
  `s` remplace `x` et `S` remplace `-`.
- *Sticky Bit* (`t` ou `T`) : `--------t` ou `--------T`
  Sur un répertoire, il interdit la suppression d'un fichier qu'il contient à tout autre utilisateur que le propriétaire du fichier.
  `t` remplace `x` et `T` remplace `-`.
  Exemple classique : `/tmp`

Permissions

Représentation octale

★★

Représentation octale

Un triplet de permission peut être représenté par un chiffre octal (0 à 7), chaque bit représentant une des permissions :

$$\texttt{r} = 4 = 100_2 \mid \texttt{w} = 2 = 010_2 \mid \texttt{x} = 1 = 001_2$$

0	`---`	pas de permission
1	`--x`	exécution
2	`-w-`	écriture
3	`-wx`	écriture et exécution
4	`r--`	lecture
5	`r-x`	lecture et exécution
6	`rw-`	lecture et écriture
7	`rwx`	lecture, écriture et exécution

Permissions

Représentation octale

★★

Représentation octale des permissions

On représente l'ensemble des permissions d'un fichier avec trois chiffres en octal (équivalents au trois triplets). Un quatrième chiffre octal peut être ajouté pour représenter les permissions spéciales.

Représentation octale des permissions spéciales

- SUID : 4000_8
- SGID : 2000_8
- Sticky Bit : 1000_8

Exemples

- `rw-r--r--` correspond à 644
- `rwsr-xr-x` correspond à 4755

Commandes relatives aux utilisateurs/groupes ⋆

Commandes d'identification

- `id` : *IDentify*
  Afficher les identifiants de l'utilisateur et de ses groupes

Commandes de modifications d'utilisateurs/groupes

- `chown [user] [file]` : *CHange OWNer*
  Change l'utilisateur propriétaire du fichier `file` pour qu'il appartienne désormais à l'utilisateur `user`. Seul `root` peut généralement appeler cette commande.
- `chgrp [group] [file]` : *CHange GRouP*
  Change le groupe propriétaire du fichier `file` pour qu'il appartienne désormais au groupe `group`. Un utilisateur peut changer le groupe d'un fichier uniquement vers un groupe auquel il appartient.

RTFM : `id(1)`, `chown(1)`, `chgrp(1)`

Commandes relatives aux permissions

`umask`

★★

Le *umask*

Le *umask* est une valeur sur quatre chiffres en octal qui est utilisée pour déterminer les permissions initiales d'un fichier ou d'un répertoire :

- Pour un fichier : `permission` $= 0666_8$ & $\sim$*umask*
- Pour un répertoire : `permission` $= 0777_8$ & $\sim$*umask*

Généralement, le *umask* est à 0022_8, ce qui donne les permissions :
$\rightarrow 0644_8$ pour les fichiers et 0755_8 pour les répertoires.

Commande `umask`

- `umask` :
  Affiche le *umask* courant
- `umask [mask]` :
  Définit le *umask* courant à `mask`

Commandes relatives aux permissions

`chmod`

★★

Commande `chmod`

- `chmod [mask] [file]` : *CHange MODe*
  Définit les permissions du fichier `file` à `mask` (en notation octale)
- `chmod [symbol] [file]` : *CHange MODe*
  Change les permissions du fichier `file` selon la notation `symbol`

Notation symbolique

La notation symbolique comprend 3 champs :

- La ou les parties concernées : une ou plusieurs lettres parmi `ugoa`
- L'action : + pour activer, - pour désactiver, = pour définir
- La ou les permissions : une ou plusieurs lettres parmi `rwxst`

RTFM : `chmod(1)`

Retour sur `ls -l`

Exemple

```
drwxr-xr-x 4 jbernard jbernard   4096 18 janv. 17:18 media
-rw-r--r-- 1 jbernard jbernard 374865 19 janv. 16:20 systeme.pdf
-rw-r--r-- 1 jbernard jbernard   1957 19 janv. 11:35 systeme.tex
```

Signification du dixième caractère : le type de fichier

- `-` : fichier régulier
- `d` (*Directory*) : répertoire
- `b` (*Block*) : fichier de périphérique en mode bloc
- `c` (*Character*) : fichier de périphérique en mode caractère
- `l` (*Link*) : lien symbolique
- `p` (*Pipe*) : tube nommé (ou fifo)
- `s` (*Socket*) : socket

Plan

Inode
Définition

★★

Définition (Inode)

Un inode (*index node*) est une structure de données contenant des informations concernant un fichier stocké dans le système de fichiers. À chaque fichier correspond un numéro d'inode (*i-number*) dans le système de fichiers dans lequel il réside, unique au périphérique sur lequel il est situé. Les inodes sont créés lors de la création du système de fichiers.

Numéro d'inode d'un fichier

Pour connaître le numéro d'inode d'un fichier,

- `ls -i [file]` :
  Donne uniquement le numéro d'inode du fichier `file`
- `stat [file]` :
  Donne des informations contenues dans l'inode du fichier `file`

RTFM : `stat(1)`

Inode

Informations contenues dans un inode

Informations contenues dans un inode

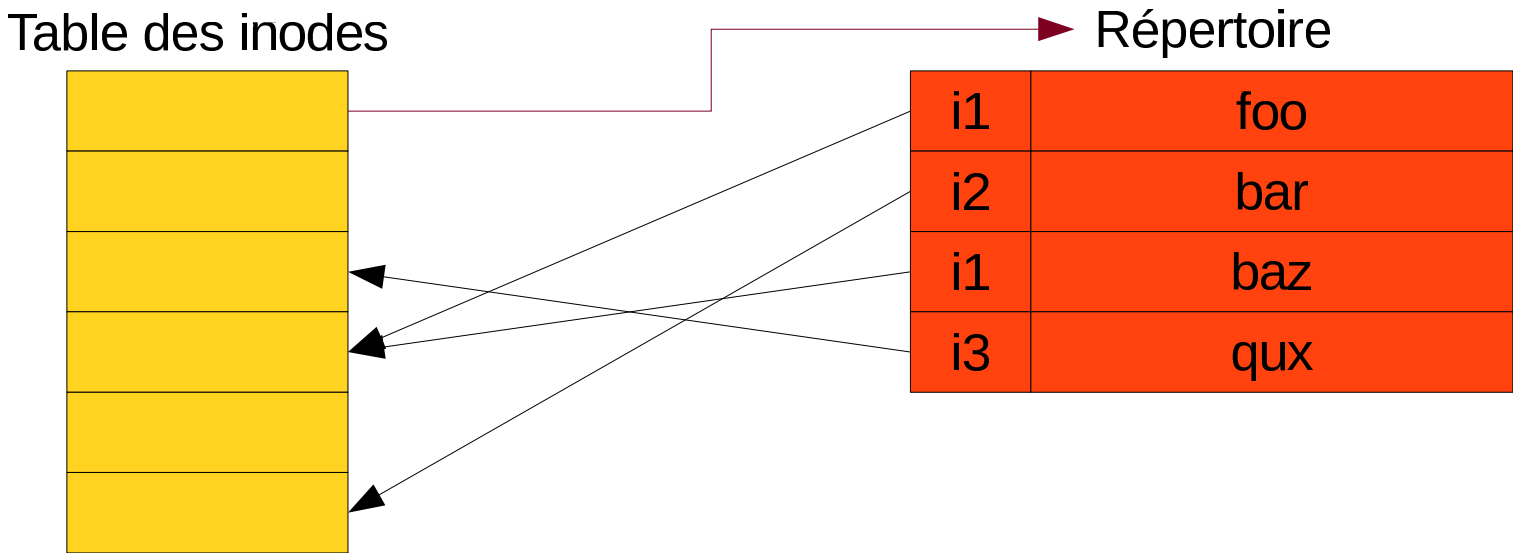
- Identifiant du périphérique
- Numéro d'inode
- Permissions du fichier
- Compteur indiquant le nombre de liens durs sur cet inode.
- Identifiant de l'utilisateur propriétaire du fichier
- Identifiant du groupe propriétaire du fichier
- Taille du fichier
- Date de dernier accès `atime` (*Access TIME*)
- Date de dernière modification `mtime` (*Modify TIME*)
- Date de dernier changement `ctime` (*Change TIME*)

→ Il n'y a pas le nom du fichier !

Inode d'un répertoire

Inode de répertoire

Un inode de répertoire pointe vers un bloc constitué d'une liste d'entrées. Chaque entrée est composée d'un numéro d'inode et d'un nom de fichier.



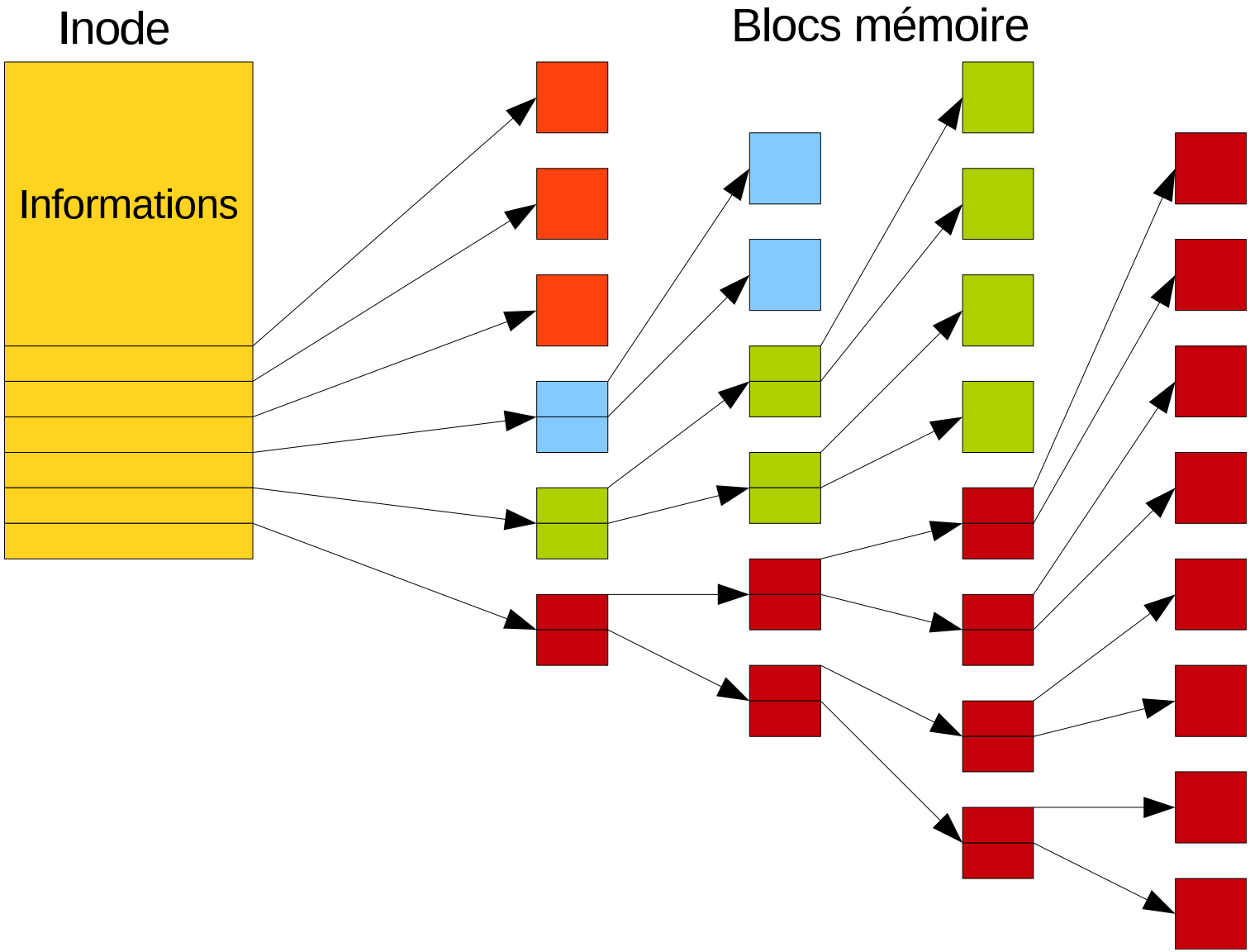
Inode d'un fichier régulier

Inode d'un fichier

Un inode de fichier contient 15 champs d'adresses de blocs mémoire :

- Les 12 premiers blocs contiennent les données du fichier
- Le 13è bloc est un bloc d'indirection qui contient des adresses vers des blocs de données du fichier
- Le 14è bloc est un bloc de double-indirection qui contient des adresses vers des blocs d'indirection
- Le 15è bloc est un bloc de triple-indirection qui contient des adresses vers des blocs de double-indirection

Inode d'un fichier régulier



Plan

Système de fichiers
Gestion de fichiers

Commandes de gestion de fichiers

- `cp [file1] [file2]` : *CoPy*
  Copie le fichier `file1` vers le fichier `file2`
- `cp [file1] [...] [filen] [dir]` : *MoVe*
  Copie les fichiers `file1`, ..., `filen` dans le répertoire `dir`
- `mv [file1] [file2]` : *MoVe*
  Déplace/renomme le fichier `file1` en `file2`
- `mv [file1] [...] [filen] [dir]` : *MoVe*
  Déplace les fichiers `file1`, ..., `filen` dans le répertoire `dir`
- `rm [file]` : *ReMove*
  Supprime le fichier `file`

RTFM : `cp(1)`, `mv(1)`, `rm(1)`

Système de fichiers
Gestion de répertoires

Commandes de gestion de répertoires

- `mkdir [dir]` : *MaKe DIRectory*
  Crée le répertoire `dir` dans le répertoire courant
- `mkdir -p [dir]` : *MaKe DIRectory*
  Crée le répertoire `dir` et tous les répertoires intermédiaires
- `rmdir [dir]` : *ReMove DIRectory*
  Efface le répertoire (vide) `dir` du répertoire courant
- `rmdir -p [dir]` : *ReMove DIRectory*
  Efface le répertoire (vide) `dir` et tous les répertoires intermédiaires

RTFM : `mkdir(1)`, `rmdir(1)`

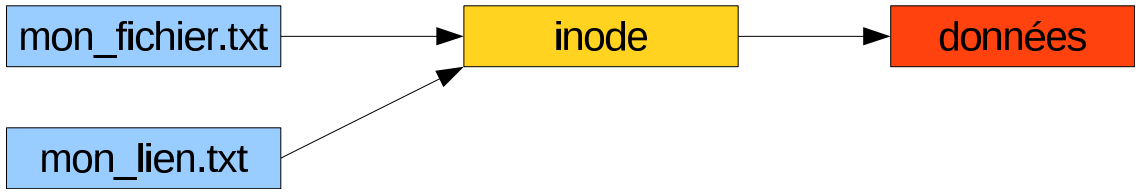
Plan

Système de fichiers
Lien dur/Lien symbolique

★★★

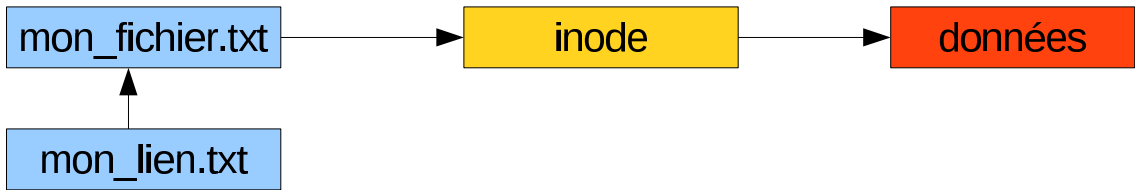
Définition (Lien dur)

Un lien dur (*hard link*) est un pointeur sur un inode qui compte pour un dans le nombre de références sur le fichier.



Définition (Lien symbolique)

Un lien symbolique (*symlink*) est une entrée spéciale de répertoire qui permet de créer un alias vers un fichier.



Système de fichiers
Commandes relatives aux liens

★★

Commandes de création de lien

- `ln [target] [link]` : *LiNk*
  Crée un lien vers `target` appelé `link`
- `ln -s [target] [link]` : *LiNk*
  Crée un lien symbolique vers `target` appelé `link`

Commandes de desctruction de lien

- `unlink [file]` :
  Détruit le nom `file` et éventuellement le fichier associé (s'il n'y a plus de référence dessus)

RTFM : `ln(1)`, `unlink(1)`

Questions de la fin

À propos des inodes de fichier

Admettons que chaque bloc mémoire fasse 2048 octets et que chaque adresse de bloc soit codée sur 4 octets. Quelle est la taille maximale d'un fichier ?

À propos des inodes de répertoire

Quel est le numéro d'inode de la racine ?

Troisième partie

Shell et scripts shell

Plan de ce cours

6. Commandes
   - Flux standard
   - Valeur de retour
   - Variables
   - Développement de la ligne de commande

7. Programmation Shell
   - Principes
   - Tests et structures de contrôle
   - Fonctions

Plan

Flux standard

Notion de flux

★★★

Définition (Flux)

Un flux est une séquence d'octets. Généralement, un flux est sous forme textuelle ($\neq$ binaire). On peut voir un flux comme un «tuyau de données».

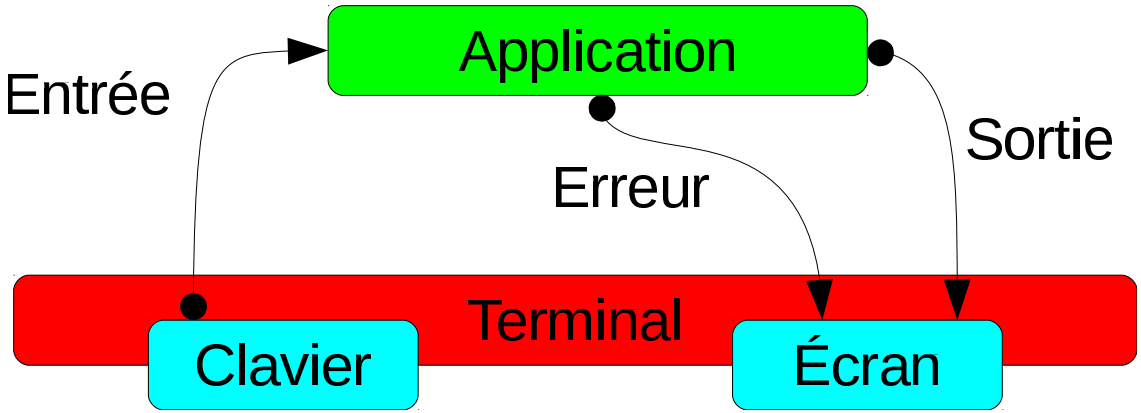
Autre vision d'un programme

«Faites de chaque programme un filtre.» = un programme peut être vu comme un filtre qui manipule des flux.

Flux standard

Flux standard

★★★

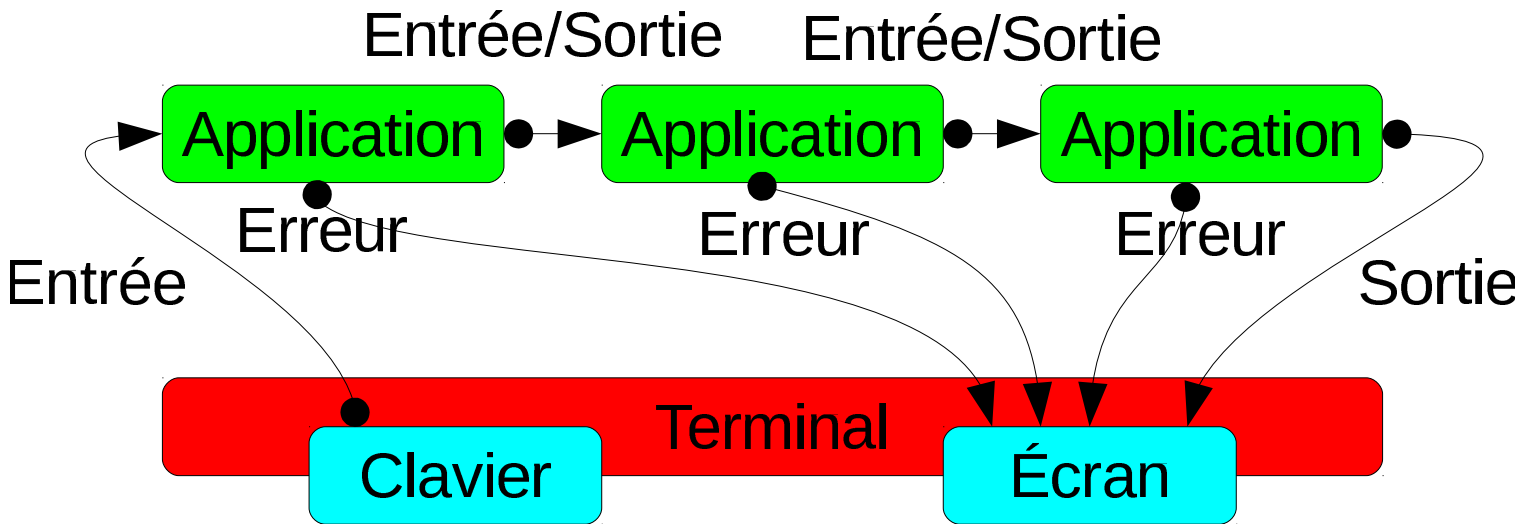


Flux standard

- Entrée standard (0, *standard input*, *stdin*) : flux d'entrée par lequel du texte ou toute autre donnée peut être entré dans un programme.
- Sortie standard (1, *standard output*, *stdout*) : flux de sortie dans lequel les données sont écrites par le programme.
- Erreur standard (2, *standard error*, *stderr*) : le flux de sortie permettant aux programmes d'émettre des messages d'erreur.

Tube

Principe d'un tube



Définition (Tube)

Un tube (*pipeline* ou *pipe*) est un mécanisme qui permet de chaîner la sortie standard d'une application à l'entrée standard d'une autre application. On utilise le caractère | pour créer un tube.

```
$ commande1 | commande2 | commande3
```

Tube
Exemples

Exemples

- `$ ps aux | more`
  Permet d'afficher la liste de tous les processus page par page.
- `$ echo -e "login\npassword" | ftp example.com`
  Permet d'envoyer le nom et le mot de passe à travers la ligne de commande pour le programme de connexion à un ftp.
- `$ ls -1 | wc -l`
  Affiche les fichiers présents dans le répertoire courant (un sur chaque ligne) puis compte le nombre de lignes : compte le nombre de fichier dans le répertoire.

Redirection
Principes

★★

Définition (Redirection)

Une redirection est un mécanisme qui permet de dérouter un flux.

Types de redirection

- `$ commande < fichier`
  Redirige l'entrée standard depuis `fichier`. Les données seront lues depuis `fichier` plutôt que sur le clavier.
- `$ commande > fichier`
  Redirige la sortie standard dans `fichier`. Les données seront écrites dans `fichier` plutôt que sur l'écran.
- `$ commande >> fichier`
  Idem, mais `fichier`, s'il existe, n'est pas écrasé, les données sont mises à la suite.

Redirection
Principes

★★

Types de redirection

- `$ commande < fichier1 > fichier2`
  Les deux à la fois.
- `$ commande 2> fichier`
  Redirige l'erreur standard dans `fichier`. Les messages d'erreur seront écrits dans `fichier` plutôt que sur l'écran.
- `$ commande 2>> fichier`
  Idem, mais `fichier`, s'il existe n'est pas écrasé, les données sont mises à la suite.
- `$ commande 2>&1`
  Redirige l'erreur standard sur la sortie standard.
- `$ commande 1>&2`
  Redirige la sortie standard sur l'erreur standard (peu courant).

Redirection
Exemples

Exemples

- `$ man man > man.txt`
  Redirige la page de manuel `man(1)` dans le fichier `man.txt`
- `$ cat fichier.txt > copie.txt`
  Mauvaise manière de copier un fichier.
- `$ cat fichier1.txt fichier2.txt fichier3.txt > tous.txt`
  Concatène les fichiers et met le résultat dans `tous.txt`
- `$ commande < fichier.txt`
  est fonctionnellement équivalent à :
  `$ cat fichier.txt | commande`
- `$ commande1 > tmp; commande2 < tmp`
  est fonctionnellement équivalent à :
  `$ commande1 | commande2`

Interception de flux

La commande tee

Interception de flux

Il est possible d'intercepter un flux et de le rediriger dans un fichier en insérant la commande tee entre deux commandes.

```
$ commande1 | tee fichier.txt | commande2
```

Exemple

```
$ echo "Vivent les tubes." | tee fichier.txt | wc
 1 3 18
$ cat fichier.txt
Vivent les tubes.
```

Plan

Valeur de retour
Signification

★★★

Valeur de retour

Chaque programme renvoie une valeur de retour qui indique le bon fonctionnement du programme (0), ou alors une erreur dans le programme ($\neq 0$). La valeur des erreurs dépend du programme. On peut récupérer la valeur de retour de la dernière commande appelée en utilisant $?

Exemple

```
$ test -d /etc/passwd
$ echo $?
1
```

Valeur de retour

Enchaînement

★★

Enchaînement

Il est possible d'enchaîner des commandes tant que les commandes ne font pas d'erreur, c'est-à-dire tant que la valeur de retour de la commande est 0, grâce à && (Attention ! C'est différent de ;) :

```
$ commande1 && commande2
```

Exemples

```
$ test -d /etc && echo "C'est un répertoire"
C'est un répertoire
$ test -d /etc/passwd && echo "C'est un répertoire"
$
```

Valeur de retour

Gestion des erreurs

★★

Gestion des erreurs

Il est possible d'appeler une commande en cas d'erreur d'une commande, c'est-à-dire quand la valeur de retour de la commande est $\neq 0$, grâce à `||` :

```
$ commande1 || commande2
```

Exemples

```
$ test -f /etc || echo "Ce n'est pas un fichier"
Ce n'est pas un fichier
$ test -f /etc && echo "Oui" || echo "Non"
Non
$ test -f /etc/passwd && echo "Oui" || echo "Non"
Oui
```

Plan

Variables

Affectation d'une variable

★★

Variable de shell

Le shell permet l'utilisation de variables. Une variable de shell est une variable non-typée, dont le nom contient uniquement des lettres (généralement en majuscule), des chiffres et le caractère _. Exemple : `PATH`

Affectation d'une variable

L'affectation d'une valeur à une variable se fait de la manière suivante :

`$ VAR=`*value*

- Attention ! Pas de $ devant le nom de la variable !
- Pas d'espace entre le nom de la variable et le = !

Exemple

```
$ PI=3.1415
$ MY_NAME=Julien
```

Variables

Contenu d'une variable

★★

Contenu d'une variable

On lit le contenu d'une variable à l'aide de $. On peut éventuellement entourer le nom de la variable de { et }.

```
$ echo $VAR
$ echo ${VAR}
```

Exemple

```
$ echo PI
PI
$ echo $PO

$ echo $PI
3.1415
$ echo "${PI}957"
3.1415957
```

Environnement ⋆⋆

Définition (Environnement)

L'environnement du shell est l'ensemble des variables d'environnement, c'est-à-dire l'ensemble des variables prédéfinies du shell. On peut les consulter avec la commande `env`.

La variable d'environnement PATH

La variable PATH contient une liste de répertoires séparés par `:` qui contiennent des programmes que l'on peut exécuter.

Exemple

```
$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/games
```

RTFM : `env(1)`

Variables
Types de variables

★★

Types de variables

- Variable d'environnement
  - pré-définie par le shell
  - accessible par les commandes externes
- Variable utilisateur
  - définie par l'utilisateur
  - inaccessible par les commandes externes

`export`

Pour transformer une variable utilisateur en variable d'environnement, on utilise la commande `export`.

```
$ export VAR
```

Variables

Lecture sur l'entrée standard

Lecture sur l'entrée standard

La commande `read` permet de lire une ou plusieurs variables sur l'entrée standard.

```
$ read VAR
$ read VAR1 VAR2 VAR3
```

C'est la variable d'environnement IFS (*Internal Field Separator*) qui désigne les caractères qui servent de séparateurs entre les variables (' \t\n' par défaut).

Exemple

```
$ IFS=';'
$ read J M A
14;07;1789
$ echo "Date: $J/$M/$A"
Date: 14/07/1789
```

Plan

Développement de la ligne de commande
Définition

★★

Développement de la ligne de commande

Le développement de la ligne de commande est l'ensemble des opérations réalisées par le shell avant d'exécuter réellement la commande.

- Développement du tilde
  → ~ est remplacé par le nom du répertoire utilisateur
- Développement des paramètres et des variables
  → les variables sont remplacées par leur contenu
- Substitution de commandes
- Développement arithmétique
- Découpage en mots (suivant la valeur de `IFS`)
- Développement des chemins

Développement de la ligne de commande

Substitution de commande

★★

Substitution de commande

La substitution de commande permet de remplacer une commande par son résultat (sortie standard). Il existe deux formes :

- `‘commande‘`
- `$(commande)`

Exemple

```
$ echo "Nombre d'utilisateurs : ‘cat /etc/passwd | wc -l‘"
Nombre d'utilisateurs : 32
$ DIR=$(pwd)
```

Développement de la ligne de commande

Développement arithmétique

★★

Développement arithmétique

Le développement arithmétique permet de remplacer une expression arithmétique par le résultat de son évaluation. Son format est : $((expression))

Exemple

```
$ A=$((3 + 4))
$ echo $(($A - 2))
5
```

Développement de la ligne de commande

Développement des chemins

★★

Développement des chemins

Le développement des chemins permet la substitution de certains motifs (*pattern*) par une liste de fichiers.

Liste de motifs

- `*` remplace n'importe quelle chaîne de caractères
- `?` remplace n'importe quel caractère
- `[abc]` remplace un caractère qui peut être a, b ou c
- `[a-z]` remplace un caractère qui peut être a, b, . . . , z
- `[[:classe:]]` remplace un caractère d'une classe parmi `alnum alpha ascii blank cntrl digit graph lower print punct space upper word xdigit`

RTFM : `isalpha(3)`

Développement de la ligne de commande

Développement des chemins

★★

Exemples

- `*` désigne tous les fichiers
- `*.c` désigne tous les fichiers dont l'extension est `.c`
- `exemple?` désigne tous les fichiers dont le nom commence par `exemple` suivi d'un caractère
- `*.[Jj][Pp][Gg]` désigne tous les fichiers JPEG dont l'extension peut être écrite en majuscule ou en minuscule
- `/var/log/mysql.log.[[:digit:]].gz`
- `[[:alnum:]]` est équivalent à `[a-zA-Z0-9]`

RTFM : `glob(7)`

Plan

Scripts shell
Définition

★★

Définition (Script shell)

Un script shell est un fichier texte contenant des commandes shell à exécuter. On lui donne généralement l'extension `.sh`. Un script shell commence obligatoirement par un shebang (`#!`) suivi du nom de l'interpréteur de commandes :

```
#!/bin/sh
```

Pour exécuter un script shell :

`$ . ./script.sh` ou `$ ./script.sh`

Shell et script shell

Le shell implémente un langage de programmation assez complet qu'il est possible d'utiliser sur la ligne de commande ou dans un script. Tout ce qui est faisable dans un script est faisable sur la ligne de commande et vice-versa.

Scripts shell
Hello World

Exemple (Hello World en shell)

```
#!/bin/sh
echo "Hello World"
```

Scripts shell

Commandes interne/externe

★★

Définition (Commande interne)

Une commande interne est une commande que le shell connaît et qu'il exécute dans le même processus que le shell. L'environnement du shell peut être modifié.

Définition (Commande externe)

Une commande externe est une commande fournie par un programme du système. Un nouveau processus est créé pour exécuter la commande. L'environnement du shell n'est pas modifié.

Exécution d'une commande externe

- Dans le `PATH` : `$ commande`
- Dans le répertoire courant : `$ ./commande`
- Dans un répertoire quelconque : `$ /chemin/vers/commande`

Scripts shell

Commandes interne/externe

Interne ou externe ?

La commande `type` permet de savoir si une commande est interne ou externe, et dans ce dernier cas indique le programme.

```
$ type commande
```

Exemples

```
$ type cd
cd is a shell builtin
$ type pwd
pwd is a shell builtin
$ type mkdir
mkdir is /bin/mkdir
```

Paramètres du script ⋆

Paramètres du script

Les paramètres du script sont dans les variables (paramètres positionnels) $0, $1, $2, ... En particulier la variable $0 contient le nom du script (ou le nom du shell si on n'est pas dans un script).
La commande `shift` permet de décaler tous les paramètres vers la gauche, l'ancien paramètre $0 est alors perdu.

Exemple

```
$ cat script.sh
echo $0
$ ./script.sh
./script.sh
$ . ./script.sh
dash
```

Variables spéciales

Variables spéciales

- `$#` : nombre de paramètres
- `$*` : liste des paramètres
- `$?` : valeur de retour du dernier processus
- `$PWD` : répertoire courant
- `$RANDOM` : nombre aléatoire
- `$SECONDS` : nombre de secondes depuis le lancement du shell

Exemple

```
$ echo "$RANDOM $RANDOM $RANDOM"
11351 3948 3381
```

Plan

Tests : commande test

test ou [

- `-d FICHIER` : `FICHIER` existe et est un répertoire
- `-f FICHIER` : `FICHIER` existe et est un fichier ordinaire
- `-n CHAINE` : `CHAINE` est non-vide
- `CHAÎNE1 = CHAÎNE2` : les deux chaînes sont égales
- `ENTIER1 -eq ENTIER2` : `ENTIER1` et `ENTIER2` sont égaux
- `! EXPR` : `EXPR` est fausse
- `EXPR1 -a EXPR2` : `EXPR1` et `EXPR2` sont vraies
- `EXPR1 -o EXPR2` : `EXPR1` ou `EXPR2` sont vraies

Exemple

```
$ test -d "$1" -a -x "$1"
```

Teste si `$1` est un répertoire exécutable.

RTFM : `test(1)`

Structures de contrôle

Structure conditionnelle

if

```
if commande
then
  commandes
else
  commandes
fi
```

Exemple

```
if test -d "/etc/passwd"
then
  echo "/etc/passwd is a directory"
else
  echo "/etc/passwd is not a directory"
fi
```

Structures de contrôle

Structure répétitive `for`

for

```
for var in list
do
  commandes
done
```

Exemple

```
for I in ‘seq 0 9‘
do
  echo "${I} is a digit"
done
```

Structures de contrôle

Structure répétitive `while`

while

```
while commande
do
  commandes
done
```

Exemple

```
REP=""
while test -z "$REP"
do
  read REP
done
```

Structures de contrôle

Structure répétitive `until`

until

```
until commande
do
  commandes
done
```

Exemple

```
REP=""
until [ -n "$REP" ]
do
  read REP
done
```

Plan

Ensemble de commandes

★★

Ensemble de commandes

Il est possible de grouper un ensemble de commandes par `{` et `}` ou par `(` et `)`. Chaque commande doit être terminée par un `;`. Un ensemble de commandes peut être vu comme une seule commande, notamment du point de vue des redirections.

Exemple

```
$ { echo -n "Hello"; echo " World"; } > message.txt
```

Fonctions

Fonctions

Une fonction est défini dans un script shell de la manière suivante :
`nom () commande`
où *commande* est une commande ou un ensemble de commandes.
Une fonction a accès à toutes les variables définies dans le script. Elle peut avoir des arguments auxquels elle accède via les variables $1, $2, ... Une fonction s'appelle de la même manière qu'une commande normale (sans parenthèses !).

Exemple

```
#!/bin/sh
hello () {
  echo "Hello $1";
}
hello "World"
```

Questions de la fin

À propos de la sortie standard d'erreur

Comment faire pour n'afficher aucune erreur quand on exécute une commande ?

À propos des commandes ?

Comment faire pour pouvoir exécuter n'importe quel programme du répertoire courant sans avoir à préfixer le nom du programme par `./` ?

Quatrième partie

Programmation en C

Plan de ce cours

8. Le langage C
   - Généralités
   - Types et opérateurs
   - Structures de contrôle
   - Tableaux et pointeurs
   - Chaînes de caractères
   - Fonctions
9. Bibliothèque standard
   - Généralités
   - Entrée/Sortie simple
   - Allocation mémoire
   - Manipulation de chaînes de caractères
   - Fonctions mathématiques
10. Production de programme
    - Production simple

Plan

Langage C
Histoire et normes

Historique

- 1973 : Création du langage C par B. Kernighan et D. Ritchie : C K&R
- 1989 : Normalisation par l'ANSI, puis par l'ISO : ANSI C ou C89
- 1999 : Mise à jour de la norme : C99
- 2011 : Mise à jour de la norme : C11

Langage C
Caractéristique du langage

Caractéristiques

- Langage impératif : description des opérations en termes de séquences d'instructions pour modifier l'état du programme.
- Langage procédural : découpage du programme en terme de procédures (ou fonctions) qui peuvent être appelées n'importe où.
- Langage bas niveau : travail avec des adresses en mémoire.
- Gestion explicite de la mémoire.

Bernard Cassagne
Introduction au langage C.
http://www-clips.imag.fr/commun/bernard.cassagne/Introduction_ANSI_C.html

Langage C
Hello World !

★★

Exemple (helloworld.c)

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

Exemple (Compilation et exécution)

```
$ gcc -o helloworld helloworld.c
$ ./helloworld
Hello World!
$ echo $?
0
```

Plan

Langage C
Types de base

★★

Types de base en C

- `char` : entier signé sur 8 bits
- `short` : entier signé sur 16 bits
- `int` : entier signé sur 16 ou 32 ou 64 bits
- `long` : entier signé sur 32 ou 64 bits
- `unsigned` *type* : version non-signée des types entiers
- `float` : flottant simple précision
- `double` : flottant double précision
- `void` : type générique vide, sans valeur

Taille des types

`sizeof(`*type*`)` renvoie la taille du type

Langage C

Opérateurs de base : comme en Java

Opérateurs en C

- opérateurs arithmétiques pour les types numériques :
  +, -, /, *, %
- opérateurs d'incrémentation/décrémentation (préfixés et postfixés) :
  ++, --
- opérateurs relationnels :
  ==, !=, <, >, <=, >=
- opérateurs logiques :
  &&, ||, !
- opérateurs bit à bit :
  &, |, ~, <<, >>

Langage C
Booléen

★★★

Booléen en C

Il n'existe pas de type `boolean` ou équivalent en C (et par conséquent ni de `true`, ni de `false`). Quand on utilise une variable `x` de type entier en tant que booléen, on a l'équivalence suivante :

`x` vaut *vrai* si et seulement si `x != 0`

Le type boole existe !!!
_Bool ou bool en utilisant
#include <stdbool.h>

Langage C
Types structurés

★★

struct

```
struct nom {
   type1 nom1;
   type2 nom2;
   ...
};
```

Exemple (Déclaration)

```
struct date {
   int day;
   int month;
   int year;
};
```

Exemple (Utilisation)

```
struct date d1;
d1.day = 14;
d1.month = 7;
d1.year = 1789;

struct date d2 = { 14, 7, 1789 };
```

Langage C
Types énumérés

★★

enum

```
enum nom {
   NOM1,
   NOM2,
   ...
};
```

Exemple (Déclaration)

```
enum color {
   RED,
   GREEN,
   BLUE
};
```

Exemple (Utilisation)

```
enum color c1, c2;
c1 = RED;
c2 = GREEN;
```

Variable const

★★

const

Le mot-clef `const` ajouté au début de la déclaration d'une variable de type simple, structuré ou énuméré signifie que la valeur de la variable ne peut pas changer. Il faut donc nécessairement attribuer une valeur à cette variable au moment de la déclaration.

Exemples

```
const int two = 2;
const struct date d = { 14, 7, 1789 };
const enum color c = RED;
```

Alias de type

★★

typedef

Il est possible de créer des alias de type à l'aide du mot-clef `typedef`, c'est-à-dire de substituer le nom d'un type par un autre. Il est fortement recommandé de choisir un nom finissant par `_t`. Déclaration de la forme : `typedef` *type identifiant*;

Exemple (Déclaration)

```
typedef int integer_t;
typedef struct date date_t;
typedef enum color color_t;

typedef struct {
  double real;
  double imaginary;
} complex_t;
```

Exemple (Utilisation)

```
integer_t i = 3;
date_t d = { 14, 7, 1789 };
color_t c = RED;

complex_t j = { 0., 1. };
```

Plan

Langage C

Structures conditionnelles : comme en Java

if

```
if (condition) {
  intructions;
}
```

if/else

```
if (condition) {
  intructions;
} else {
  intructions;
}
```

Exemple

```
int x = 0;
if (!x) {
  printf("This will be printed.\n");
} else {
  printf("This will *not* be printed.\n");
}
```

Langage C

Structure répétitive : comme en Java (encore)

while

```
while (condition) {
  intructions;
}
```

do ...while

```
do {
  intructions;
} while (condition);
```

Exemple

```
int i = 37;
while (i != 1) {
  if (i % 2 == 0) {
    i = i / 2;
  } else {
    i = 3 * i + 1;
  }
}
```

Langage C

Structure répétitive : comme en Java (comme d'hab')

for

```
for (initialisation; condition; mise à jour) {
  intructions;
}
```

Exemple

```
int i, j;
for (i = 0; i < 10; i++) {
  for (j = 0; j < i; ++j) {
    printf("#");
  }
  printf("\n");
}
```

Langage C

Structure de choix : comme en Java (vous avez compris ?)

switch

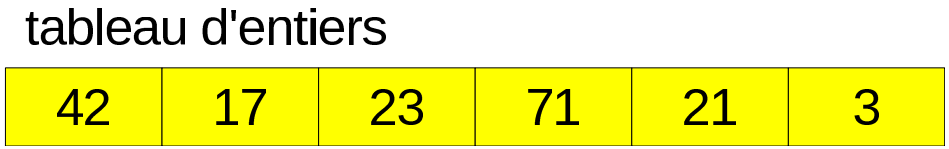
```
switch (valeur) {
  case valeur1 :
    intructions;
    break;
  case valeur2 :
    intructions;
    break;
  ...
  default:
    intructions;
    break;
}
```

Exemple

```
int i = ...;
switch (i) {
  case 0:
    printf("i vaut 0.\n");
    break;
  case 1:
    printf("i vaut 1.\n");
    break;
  default:
    printf("i vaut ?.\n");
}
```

Plan

Langage C
Tableaux



Définition (Tableau)

Un tableau est un espace mémoire contenant plusieurs éléments contigus du même type. Il existe deux types de tableaux en C, les tableaux statiques dont la taille est connue à la compilation et les tableaux dynamiques dont la taille est connue à l'exécution. Les tableaux sont indicés à partir de 0.
«*type nom*[*N*]» = tableau statique de *N* élément de type *type*.

Langage C

Déclaration et utilisation d'un tableau

★★

Exemples (Déclaration)

`struct color colors[3];`

`colors` est un tableau de 3 `struct color`

`char str[] = { 'a', 'b, 'c', 'd', 'e' };`

`str` est un tableau de 5 `char` (automatique).

`int array[10] = { 1, 2 };`

`array` est un tableau de 10 `int`.

Exemple (Accès aux éléments)

`array[0]`

Accès au premier élément du tableau `array`.

`str[10]`

Accès au 11è élément du tableau `str` ! Pas de vérification !

Langage C
Pointeur

Définition (Pointeur)

Un pointeur est une adresse vers un espace mémoire typé.
«*type* *» = pointeur vers *type*

Définition (`NULL`)

`NULL` représente l'adresse 0, c'est-à-dire l'adresse sur rien. Équivalent au `null` de Java.

Langage C
Déclaration d'un pointeur

★★

Exemples

`int *p;`
  `p` est un pointeur vers un `int`. Ou plutôt `*p` est un `int`.

`int *p, q;`
  `q` est un `int` !

`int *p, *q;`
  `*p` et `*q` sont des `int`.

`struct date *birthday;`
  `birthday` est un pointeur vers une structure `date`.

`void *buf;`
  `buf` est un pointeur générique.

Langage C
Opérateur * et &

★★

Référencement

L'opérateur & permet de récupérer l'adresse d'une variable.

Exemple

```
int i = 3;
int *p = &i;
```

p contient l'adresse de i.

Déréférencement

L'opérateur * permet de récupérer le contenu de la variable pointée.

Exemple

```
int j = *p;
```

j contient le contenu de ce qui est pointé par p.

Langage C
Pointeurs et structures

Accès aux membres d'une structure

L'opérateur -> permet d'accéder aux membres d'une structure via un pointeur sur cette structure.

Exemples

```
struct date *birthday = ...;
birthday->month
```

Permet d'accéder au champ `month` de la structure `date`.

```
(*birthday).month
```

Idem.

Langage C
Pointeurs et const

★★

Exemples

`const int *p;`
- `p` est un pointeur vers un `const int`.
- `p` pourra être modifié !
- `*p` ne pourra pas être modifié !

`int * const q;`
- `q` est un pointeur constant vers un `int`.
- `q` ne pourra pas être modifié !
- `*q` pourra être modifié !

`const int * const r;`
- `r` est un pointeur constant vers un `const int`.
- `r` ne pourra pas être modifié !
- `*r` ne pourra pas être modifié !

Langage C

Arithmétique sur les pointeurs : addition

★★

Addition d'un pointeur et d'un entier

Il est possible d'ajouter un entier signé n à un pointeur P de type $\mathcal{T}$. Le résultat $P + n$ est un pointeur du même type $\mathcal{T}$ qu'on a avancé de n fois la taille de $\mathcal{T}$.

Exemples

```
int array[10];
int *p = &array[0];
```

p pointe sur le premier élément du tableau array.

```
int *q = p + 1;
```

q pointe sur le deuxième élément du tableau array.

```
q++;
```

q pointe sur le troisième élément du tableau array.

Langage C

Arithmétique sur les pointeurs : soustraction

★★

Soustraction de deux pointeurs

Il est possible de soustraire deux pointeurs P et Q de même type $\mathcal{T}$. Le résultat $P - Q$ est un entier signé de type `ptrdiff_t` qui contient le nombre d'objet de type $\mathcal{T}$ entre P et Q.

Exemples

```
int array[10];
int *p = &array[0];
int *q = p + 3;
ptrdiff_t diff1 = q - p;
```

diff1 vaut 3.

```
ptrdiff_t diff2 = p - q;
```

diff2 vaut -3.

Langage C
Équivalence tableau/pointeur

★★

Équivalence tableau/pointeur

En C, un tableau est un pointeur vers le premier élément du tableau. Et inversement, un pointeur peut être vu comme un tableau.

Exemples

`int array[10];`
  `array` a pour type `int * const`.
`array[2]`
  Strictement équivalent à `*(array + 2)`.
`int *p = ...;`
`p[2]`
  Strictement équivalent à `*(p + 2)`.
`p[0]`
  Strictement équivalent à `*p`.

Langage C
Détails sur les pointeurs

Voir UE «Algorithmique sur les données».

Plan

Langage C
Chaîne de caractères

★★★

Définition (Chaîne de caractères)

Une chaîne de caractère est un tableau de `char` dont le dernier élément est le caractère `'\0'`.

Exemples

```
char str1[] = {'a', 'b', 'c', '\0' };
```
`str1` est une chaîne de caractères.
```
char str2[] = "abc";
```
`str2` est la même chaîne de caractères que `str1`.
```
char *str3 = "abc";
```
Idem. Mais `str3` ne sera pas modifiable.
```
""
```
Représente la chaîne vide. Équivalent à `{ '\0' }`.

Plan

Langage C
Définition et déclaration d'une fonction

★★

Définition d'une fonction

```
type nom(type1 par1, ..., typeN parN) {
  intructions;
  return something;
}
```

Exemple

```
int max(int i, int j) {
  if (i > j) {
    return i;
  }
  return j;
}
```

Langage C
Définition et déclaration d'une fonction

★★

Déclaration d'une fonction

La déclaration d'une fonction est une description de la fonction sous forme de prototype, c'est-à-dire son type de retour, son nom et ses paramètres. Un prototype indique qu'une fonction existe et qu'on peut l'utiliser.

type nom(*type1* par1, ..., *typeN* parN);

Exemple

```
int max(int i, int j);
```

Exemple (Cas particulier d'une fonction sans paramètre)

```
int do_something(void);
```

Langage C

Argument par valeur / par référence

★★

Argument par valeur / par référence

Tous les arguments des fonctions en C sont passés par valeur. Pour passer un argument par référence, il est nécessaire d'utiliser un pointeur sur cet argument.

Exemple (Échange du contenu de deux variables de type `int`)

```
void exchange(int *i, int *j) {
  int tmp = *i;
  *i = *j;
  *j = tmp;
}

int x, y;
exchange(&x, &y);
```

Langage C
Fonction principale

★★

Fonction principale

La fonction principale d'un programme en C est appelé `main` et a un des prototypes suivants :

```
int main(void);
int main(int argc, char *argv[]);
```

où `argc` est le nombre de paramètres et `argv` est un tableau de `argc` chaînes de caractères, chaque chaîne représentant un argument du programme. La valeur renvoyée par `main` est la valeur de retour du programme.

Remarque sur `argc` et `argv`

Il y a toujours au moins un argument ($\texttt{argc} \geq 1$), `argv[0]` : le nom du programme.

Plan

Bibliothèque standard

Qu'est-ce que c'est ?

★★

Définition (Bibliothèque standard)

La bibliothèque standard est un ensemble de fonctions de base nécessaire dans la plupart des programmes. Pour pouvoir les utiliser, il est nécessaire d'inclure des en-têtes, c'est-à-dire des fichiers qui contiennent les prototypes des fonctions souhaitées. Il existe 24 fichiers d'en-têtes standard.

Exemples

```
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Plan

Bibliothèque standard

Fonctions d'entrée/sortie basiques

★★

Fonctions d'écriture sur le terminal : `printf(3)`

```
#include <stdio.h>
int printf(const char *format, ...);
```

La fonction `printf` permet d'écrire un texte formaté sur la sortie standard. Elle prend un nombre variable d'arguments qui dépend de la chaîne de format.

Fonctions de lecture d'un terminal : `scanf(3)`

```
#include <stdio.h>
int scanf(const char *format, ...);
```

La fonction `scanf` permet de lire un texte formaté sur l'entrée standard. Elle prend un nombre variable d'arguments qui dépend de la chaîne de format. L'utilisation de cette fonction est à éviter au maximum !

Bibliothèque standard

Chaîne de format

★★

Définition (Chaîne de format)

Une chaîne de format est une chaîne de caractères qui contient des séquences de contrôle (commençant par %) permettant d'interpréter le type des arguments et de les insérer correctement dans la chaîne.

Séquences de contrôle

- `%d` ou `%i` pour les valeurs de types `int` ;
- `%f` pour les valeurs de types `double` ;
- `%c` pour les valeurs de types `char` ;
- `%s` pour les valeurs de types `char *` ;
- `%p` pour les valeurs de types `void *` ;
- `%%` pour afficher le caractère %.

RTFM : `printf(3)`

Bibliothèque standard

Exemples

Exemple (printf)

```
printf("Ceci est une chaîne sans séquence\n");
$ Ceci est une chaîne sans séquence

printf("%s a %i ans et mesure %f m\n", "Alice", 20, 1.70);
$ Alice a 20 ans et mesure 1.700000 m

char c = 'a';
printf("'%c' a pour code ascii %i\n", c, c);
$ 'a' a pour code ascii 97
```

Exemple (scanf)

```
int i;
scanf("%i", &i);
```

Plan

Bibliothèque standard
Mémoires

★★

Types de mémoires et allocation

Il existe trois grands types de mémoires en C :

Type	**Où**	**Quand**	**Comment**
statique	segment	compilation	implicite
automatique	pile (stack)	exécution	implicite
dynamique	tas (heap)	exécution	explicite

Voir la dernière partie de ce cours : «Mémoire virtuelle»

Allocation/Libération de la mémoire

Seule la mémoire dynamique a besoin d'une allocation et d'une libération explicite. Ce sont des fonctions de la librairie standard qui s'en chargent.

Bibliothèque standard

Allocation et libération dans le tas

★★

Allocation : `malloc(3)`

```
#include <stdlib.h>
void *malloc(size_t size);
```

La fonction `malloc` alloue `size` octets, et renvoie un pointeur sur la mémoire allouée. La mémoire allouée n'est pas initialisée. En cas d'échec, la fonction renvoie le pointeur `NULL`.

Libération : `free(3)`

```
#include <stdlib.h>
void free(void *ptr);
```

La fonction `free` libère l'espace mémoire pointé par `ptr`. Si `ptr` est `NULL`, aucune opération n'est effectuée.

Exemple d'allocation/libération dans le tas

Exemples

```
int *p = malloc(sizeof(int));
if (p != NULL) {
  *p = 42;
}
free(p);
p = NULL;

struct date *birthday = malloc(sizeof(struct date));
if (birthday != NULL) {
  do_something_with(birthday);
}
free(birthday);
birthday = NULL;
```

Bibliothèque standard

Allocation de tableau dynamique

★★

Allocation de tableau dynamique : `calloc(3)`

```
#include <stdlib.h>
void *calloc(size_t nmemb, size_t size);
```

La fonction `calloc` alloue la mémoire nécessaire pour un tableau de `nmemb` éléments de `size` octets, et renvoie un pointeur vers la mémoire allouée. Équivalent à : `malloc(nmemb * size)`:

Exemple

```
char *str = calloc(255, sizeof(char));
if (str != NULL) {
  bla_blah(str);
}
free(str);
str = NULL;
```

Erreurs classiques

★★

Erreurs classiques et règles pour les éviter

- Pointeur `NULL` : Non vérification de la réussite de `malloc`. Règle : toujours vérifier la valeur de retour des fonctions !
- Fuite mémoire (*memory leak*) : Pas de libération d'un espace mémoire alloué dynamiquement. Règle : à chaque `malloc`, un `free`.
- Pointeur invalide (*dangling pointer*) : Utilisation d'un pointeur après libération. Règle : affecter `NULL` au pointeur juste après le `free`.
- Double libération (*double free*) : Libération d'un pointeur déjà libéré. Règle : idem (c'est un cas particulier de pointeur invalide).

Plan

Bibliothèque standard

Fonctions sur les chaînes de caractères

★★

Longueur d'une chaîne de caractères : `strlen(3)`

```
#include <string.h>
size_t strlen(const char *s);
```

La fonction `strlen` calcule la longueur de la chaîne de caractères `s`, sans compter le caractère '\0' final.

Copie d'une chaîne de caractères : `strcpy(3)`

```
#include <string.h>
char *strcpy(char *dest, const char *src);
```

La fonction `strcpy` copie la chaîne pointée par `src`, y compris le caractère '\0' final dans la chaîne pointée par `dest`.

Bibliothèque standard

Conversion depuis une chaîne de caractères

★★

Conversion chaîne de caractères → entier : `atoi(3)`

```
#include <stdlib.h>
int atoi(const char *nptr);
```

La fonction `atoi` convertit le début de la chaîne pointée par `nptr` en entier de type `int`.

Exemple

```
int main(int argc, char *argv[]) {
  int n = 0;
  if(argc > 1) {
    n = atoi(argv[1]);
  }
  printf("n = %d\n", n);
  return 0;
}
```

Plan

Bibliothèque standard
Fonctions mathématiques

★★

Fonctions mathématiques

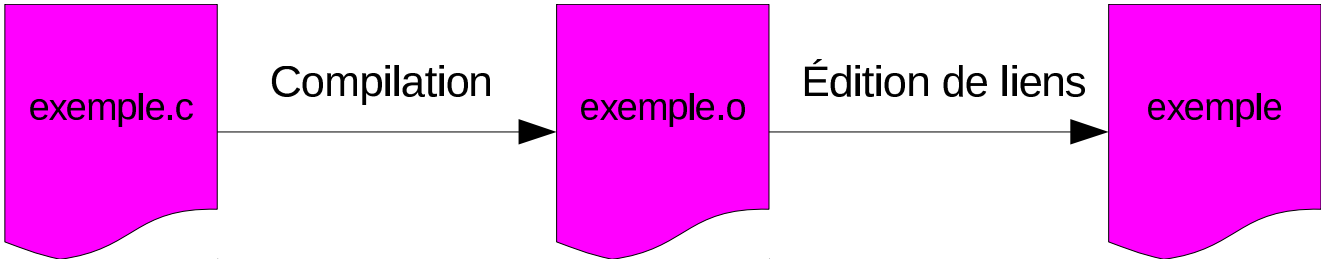
Les principales fonctions mathématiques sont définies dans l'en-tête `<math.h>` (attention, pas de s !). Parmi les fonctions qui prennent un `double` et qui renvoie un `double` : `cos`, `sin`, `tan`, `acos`, `asin`, `atan`, `sqrt`, `log`, `log2`, `log10`, `exp`, etc. Mais aussi : `fmod(double, double)`, `pow(double, double)`, etc.

Utilisation des fonctions mathématiques

Les fonctions mathématiques sont définies dans la `libm`, il est donc nécessaire de lier le programme avec l'option `-lm`.

Plan

Étapes de production



Étapes de production

1. Compilation : transformer le fichier source `.c` en fichier objet `.o`.
   ```
   $ gcc -c -o exemple.o exemple.c
   ```
2. Édition de liens : transformer le fichier objet `.o` en exécutable, en le liant aux bibliothèques externes requises.
   ```
   $ gcc -o exemple exemple.o
   ```

Retour sur Hello World !

Hello World !

Exemple (helloworld.c)

```
#include <stdio.h>

int main(int argc, char* argv[]) {
  printf("Hello World!\n");
  return 0;
}
```

Exemple (Compilation et exécution)

```
$ gcc -o helloworld helloworld.c
$ ./helloworld
Hello World!
$ echo $?
0
```

Questions de la fin

La fonction mystère

Que fait la fonction suivante ?

Exemple (`mystere.c`)

```
void mystere(char *d, const char *s) {
  while (*d++ = *s++);
}
```

Cinquième partie

Développement en C

Plan de ce cours

11. Bonnes pratiques de développement
    - Préprocesseur
    - Unité de compilation
    - Production avancée

12. Make
    - Principe d'un Makefile
    - Makefile avancé
    - Générateur de Makefile

Plan

Préprocesseur

Qu'est-ce que c'est ?

★★

Définition (Préprocesseur)

Le préprocesseur est un programme qui procède à des transformations sur un code source, avant l'étape de compilation. Il interprète des directives qui commencent par le caractère # et les remplace par du texte qui est ensuite envoyé au compilateur.

Remarques

- Il n'y a pas besoin d'appeler le préprocesseur directement, le compilateur s'en charge tout seul.
- Si on veut voir le fichier source à la sortie du préprocesseur, il est possible de passer l'option `-E` à gcc qui arrêtera le processus juste après l'étape du préprocesseur.

Préprocesseur
Inclusion de fichier

★★

Inclusion de fichier avec `#include`

La directive `#include` permet d'inclure le contenu d'un fichier dans un autre. Il y a deux types d'inclusion :

- Inclusion des fichiers systèmes :
  `#include <fichier.h>`
- Inclusion des fichiers locaux (par rapport au fichier courant) :
  `#include "fichier.h"`

Exemple

```
#include <assert.h>
#include <math.h>
#include <stdlib.h>
#include <stdio.h>
#include <string.h>
```

Préprocesseur

Définition de constantes

★★

Définition de constantes avec `#define`

La directive `#define` permet de définir des constantes qui seront remplacées dans le code source.

```
#define M_PI 3.14159265358979323846
#define DEBUG
```

Exemple (Dans le code source)

```
double x = M_PI / 2.;
```

Exemple (Après le passage du préprocesseur)

```
double x = 3.14159265358979323846 / 2.;
```

Constantes prédéfinies particulières

Le préprocesseur définit les constantes `__FILE__`, `__LINE__` qui représentent le fichier courant et la ligne courante dans le fichier.

Préprocesseur
Compilation conditionnelle

★★

Compilation conditionnelle

Les directives `#if`, `#ifdef` et `#ifndef` permettent de compiler conditionnellement, c'est-à-dire de laisser après passage du préprocesseur uniquement le code qui remplit une certaine condition.

```
#if COND
#else
#endif
```

Exemples

```
#if LIB_VERSION >= 100
#if defined(DEBUG)
#ifdef DEBUG
#ifndef DEBUG
```

Préprocesseur
Définition de macro

★★

Macros

La directive `#define` permet de définir une macro. Une macro ressemble à une fonction mais n'en est pas une !

```
#define OP(a,b) ((a) + 3 * (b))
```

Exemple (Dans le code source)

```
int x = OP(4,6);
```

Exemple (Après le passage du préprocesseur)

```
int x = ((4) + 3 * (6));
```

Préprocesseur

Les pièges classiques des macros

★★

Règles incontournables de définition d'une macro

- Une macro ne doit pas dépasser une ligne (sauf exception)
- Chaque paramètre est parenthésé dans la définition de la macro
- La définition de la macro doit elle-même être parenthésée

Exemple (Sans parenthèses autour des paramètres)

```
#define OP(a,b) (a + 3 * b)
int x = OP(4, 3 + 3);
int x = (4 + 3 * 3 + 3); /* COIN! */
```

Exemple (Sans parenthèses autour de la macro)

```
#define OP(a,b) (a) + 3 * (b)
int x = 2 * OP(4, 6);
int x = 2 * (4) + 3 * (6); /* COIN! */
```

Préprocesseur
Différences macros/fonctions

★★

Différences macros/fonctions

- Une fonction a un code binaire en mémoire, une macro n'en a pas, c'est une simple substitution de texte.
- Une fonction évalue une seule fois ses paramètres, une macro peut évaluer plusieurs fois ses paramètres.

Exemple (Évaluation multiple)

```
#define MAX(a,b) ((a) > (b)? (a) : (b))
MAX(f(),g());
```

Deux évaluations de `f()` ou `g()`.

Préprocesseur
Autres fonctionnalités

Ligne de commande

L'option `-DFOO` du compilateur permet de définir la macro `FOO`, c'est-à-dire a le même effet qu'un `#define FOO`. On peut aussi attribuer une valeur avec `-DFOO=3`.

```
$ gcc -c -DDEBUG -o fichier.o fichier.c
```

Autres fonctionnalités

Il existe des fonctionnalités plus avancées, notamment pour les macros. Mais attention ! Le préprocesseur est un outil à la fois puissant s'il est bien utilisé et dangereux s'il est mal utilisé. À utiliser de manière très raisonnée !

Plan

Unité de compilation

Qu'est-ce que c'est ?

★★★

Définition (Unité de compilation)

Une unité de compilation est un ensemble cohérent de fonctions dont la définition est dans un fichier source `.c` et la déclaration (quand elle est nécessaire) est dans un fichier d'en-tête `.h`.

À quoi ça sert ?

Au moment de la conception, un exécutable est découpé de manière logique en plusieurs morceaux. Chaque morceau est alors implémenté dans une unité de compilation. Un exécutable = plusieurs unités de compilation.

Recommandations

Une unité de compilation ne doit pas avoir plus de 20 fonctions. Il faut toujours chercher à avoir des unités de compilation simples de manière à pouvoir retrouver une fonction facilement parmi toutes les unités.

Unité de compilation

Fichier d'en-tête

★★

Définition (Fichier d'en-tête)

Un fichier d'en-tête (*header*) est un fichier regroupant les déclarations (prototypes) des fonctions d'une unité de compilation ainsi que les types associés aux fonctions de l'unité.

Pourquoi ?

Une fonction peut être appelée si la définition (corps) ou la déclaration (prototype) se trouvent avant dans le même fichier. Un fichier d'en-tête sert donc à déclarer les fonctions pour d'autres unités de compilation. Il a vocation à être inclus avec une directive `#include`.

Remarque

Une fonction peut avoir plusieurs déclarations (identiques) mais une seule définition.

Unité de compilation

include guard

★★

Problème des inclusions

Un problème peut survenir si un fichier `A` inclut un fichier `B` qui inclut le fichier `A`. Une boucle d'inclusion récursive est créée.

Solution : *include guard*

Exemple (foo.h)

```
#ifndef FOO_H
#define FOO_H

#include "bar.h"

/* types and functions */

#endif
```

Exemple (bar.h)

```
#ifndef BAR_H
#define BAR_H

#include "foo.h"

/* types and functions */

#endif
```

Unité de compilation

Exemple pratique

Exemple (Jusqu'à présent)

```
#include <stdio.h>

void say_hello(char *who) {
  printf("Hello %s!", who);
}

int main() {
  say_hello("world");
  return 0;
}
```

Unité de compilation

Exemple pratique

★★

Exemple (hello.h)

```
#ifndef HELLO_H
#define HELLO_H

void say_hello(char *who);

#endif /* HELLO_H */
```

Exemple (hello.c)

```
#include "hello.h"
#include <stdio.h>

void say_hello(char *who) {
  printf("Hello %s!", who);
}
```

Exemple (main.c)

```
#include "hello.h"
int main() {
  say_hello("world");
  return 0;
}
```

Plan

Problématique de la production de programmes ⋆⋆

Situation actuelle

Jusqu'à présent, vos programmes sont constitués d'un seul fichier. Généralement, votre programme n'a pas besoin d'autres fonctions que celles qui sont définies dans votre fichier source (en dehors des fonctions de la bibliothèque standard).

Problématique

Vous allez avoir à produire des programmes de plus en plus complexes, utilisant de nombreuses fonctionnalités existantes. Cela pose deux problèmes :

- Comment produire des programmes constitués de plusieurs fichiers ?
- Comment mutualiser des fonctions utilisées par plusieurs programmes ?

Production d'un exécutable à partir de plusieurs fichiers ⋆⋆

Étapes de production

Étapes de production

1. Préprocesseur (`cpp(1)`) : interprétation des directives
2. Compilateur (`cc(1)`) : fichier source `.c` → fichier assembleur `.s`.
3. Assembleur (`as(1)`) : fichier assembleur `.s` → fichier objet `.o`.
4. Éditeur de liens (`ld(1)`) : liaison aux bibliothèques externes requises.

Pilote de compilation

`gcc` est un pilote de compilation (*compiler driver*), c'est-à-dire qu'il va appeler successivement tous ces programmes. Il est possible d'arrêter le processus après chacune des étapes via les options respectives : `-E`, `-S`, `-c`.

Production d'un exécutable à partir de plusieurs fichiers ⋆⋆

Fichier objet et liaison

Définition (Fichier objet)

Un fichier objet est un fichier contenant le code machine correspondant au fichier source ainsi que des informations sur les symboles (fonctions, variables) utilisés mais non définis dans le fichier.

Définition (Liaison)

La liaison est un processus qui permet de résoudre les symboles, c'est-à-dire d'associer une adresse aux symboles utilisés. La liaison peut s'effectuer de deux manières :

- soit en trouvant le symbole dans une bibliothèque externe, on parle alors de liaison dynamique
- soit en trouvant le symbole dans un autre fichier objet, on parle alors de liaison statique

Production d'un exécutable à partir de plusieurs fichiers ⋆⋆

Comment faire ?

Comment faire ?

Pour créer un exécutable à partir de plusieurs fichiers, il suffit de :

1. Compiler les différents fichiers `.c` en fichiers objets `.o`.
2. Lier tous les fichiers `.o` ensemble pour créer l'exécutable

Exemple

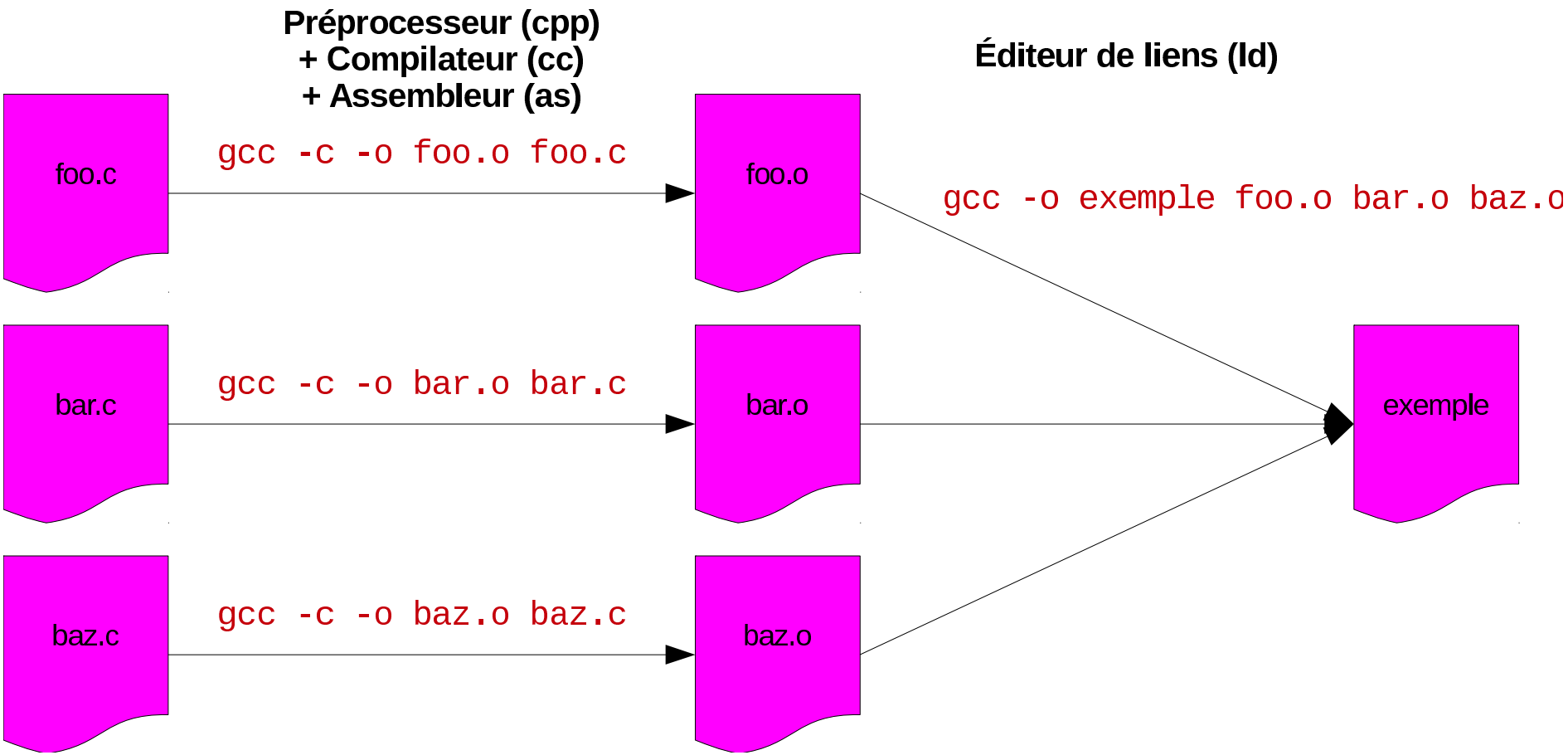
Un programme `exemple` est constitué de trois fichiers sources : `foo.c`, `bar.c`, `baz.c`. On le produit de la manière suivante :

```
$ gcc -c -o foo.o foo.c
$ gcc -c -o bar.o bar.c
$ gcc -c -o baz.o baz.c
$ gcc -o exemple foo.o bar.o baz.o
```

Production d'un exécutable à partir de plusieurs fichiers

★★★

Synthèse



Programme et bibliothèque ⋆⋆

Définition (Programme)

Un programme (ou application) est un exécutable qu'on peut appeler depuis la ligne de commande, c'est-à-dire qu'il contient une fonction `main`.

Définition (Bibliothèque)

Une bibliothèque est un fichier qui regroupe un ensemble de fonctions mais qui ne contient pas de fonction `main`. L'intérêt d'une bibliothèque est de ne pas avoir à réécrire sans cesse les mêmes fonctions. Il existe deux types de bibliothèque : statique et dynamique. Le nom d'une bibliothèque commence toujours par `lib`.

Bibliothèque standard C

La bibliothèque standard C est appelée `libc` et est liée automatiquement à tous les programmes C.

Bibliothèque statique

★★

Définition (Bibliothèque statique)

Une bibliothèque statique (ou archive) est un fichier qui contient des fichiers objets copiés dans chacun des programmes qui utilisent cette bibliothèque. Une bibliothèque statique a généralement l'extension `.a`.

Production d'une bibliothèque statique

Pour produire une bibliothèque statique, on fait appel à la commande `ar(1)` (*ARchive*) qui sert à créer des archives. Elle dispose de nombreuses options de manipulation des archives. Cependant les options utiles pour la création d'une bibliothèque statique sont `cru`.

Exemple

```
$ ar cru libqux.a foo.o bar.o baz.o
```

RTFM : `ar(1)`

Bibliothèque dynamique ⋆⋆

Définition (Bibliothèque dynamique)

Une bibliothèque dynamique est une bibliothèque qui est liée aux programmes qui l'utilisent sans être copiée. Une bibliothèque dynamique a généralement l'extension `.so` (*Shared Object*).

Production d'une bibliothèque dynamique

Pour produire une bibliothèque dynamique, on fait appel à la commande `gcc` (qui appelle `ld`) avec l'option `-shared`.

Exemple

```
$ gcc -shared -o libqux.so foo.o bar.o baz.o
```

Utilisation d'une bibliothèque ⋆⋆

Comment utiliser `libqux` ?

Pour utiliser la bibliothèque `libqux` (statique ou dynamique), il faut :

- inclure l'en-tête dans le fichier source de manière à pouvoir utiliser les fonctions de la bibliothèque
  → l'option `-I` permet d'ajouter un répertoire non-standard dans la liste des répertoires de recherche des en-têtes
- au moment de l'édition de liens, lier à la bibliothèque avec l'option `-l` suivi du nom de la bibliothèque sans le `lib` initial
  → l'option `-L` permet d'ajouter un répertoire non-standard dans la liste des répertoires de recherche des bibliothèques

```
$ gcc -lqux -o exemple exemple.c
```

Chargement d'un programme

★★

Chargement d'un programme

Au lancement d'un programme, le chargeur de programme, appelé `ld.so(8)`, effectue les actions suivantes (liaison dynamique) :

1. il trouve et charge les bibliothèques dynamiques requises par le programme
2. il résout les symboles contenus dans le programme grâce aux bibliothèques dynamiques
3. il lance le programme

Plan

Makefile ⋆⋆

Définition (Makefile)

Un Makefile est un fichier qui décrit les différentes actions nécessaires à la production d'un logiciel à partir de données sources. La commande `make(1)` permet d'exécuter le Makefile du répertoire courant. Un Makefile permet d'automatiser la production d'un logiciel, c'est-à-dire d'éviter de taper les commandes de production à chaque changement dans le code source : la commande `make(1)` se charge de tout !

Différence avec un script shell

La principale différence avec un script shell est l'optimalité de la production : on ne recompile que ce qui est nécessaire !

Makefile
Fonctionnement

★★★

Fonctionnement

Un Makefile est composé d'une suite de règles qui ont la forme suivante :

```
cible: dépendances
⟼actions
```

où les dépendances sont les fichiers nécessaires à la production de la cible et les actions sont les commandes nécessaires pour construire la cible à partir des dépendances. À l'appel de `make(1)` :

1. les dépendances sont analysées récursivement
2. si les dépendances sont plus récentes que la cible alors on exécute les actions de manière à produire la cible

Makefile
Exemple

Exemple (Makefile)

```
hello: hello.o main.o
	gcc -g -o hello hello.o main.o
hello.o: hello.c hello.h
	gcc -c -Wall -g -o hello.o hello.c
main.o: main.c hello.h
	gcc -c -Wall -g -o main.o main.c
```

Exécution des règles

`$ make`

Par défaut, exécute la première règle, donc `hello`

`$ make hello.o`

Exécute la règle `hello.o`

Makefile
Bonnes pratiques

Bonnes pratiques

- Il existe toujours une règle `all` qui sert de règle par défaut et qui dépend de la règle principale qui produit l'ensemble du projet
- Il existe toujours une règle `clean` qui sert à nettoyer tous les fichiers générés au cours de la production, à l'exception du fichier final
- Il existe parfois une règle `mrproper` qui sert à nettoyer tous les fichiers générés au cours de la production, y compris le fichier final
- Il existe souvent une règle `install` qui permet d'installer le logiciel (programmes, bibliothèques, en-têtes) sur le système

Makefile
Exemple amélioré

Exemple (Makefile)

```
all: hello
hello: hello.o main.o
	gcc -g -o hello hello.o main.o
hello.o: hello.c hello.h
	gcc -c -Wall -g -o hello.o hello.c
main.o: main.c hello.h
	gcc -c -Wall -g -o main.o main.c
clean:
	rm -f *.o
mrproper: clean
	rm -f hello
```

Plan

Makefile
Variables

★★

Variables dans un Makefile

On définit des variables de la même manière que dans un script shell

Variables classiques

- `CC` : compilateur C (généralement `gcc`)
- `CFLAGS` : options de compilation (utilisée avec `gcc -c`)
- `LDFLAGS` : options de liaison (utilisée avec `gcc` pour l'édition de lien)

Options classiques

- `-g` : inclut les symboles de debug
- `-Wall` : active tous les warnings (obligatoire !)
- `-O3` : optimisation de niveau 3 (maximum)
- `-Os` : optimisation de la taille

Makefile

Exemple avec variables

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -g
LDFLAGS=-g
TARGET=hello

all: ${TARGET}
hello: hello.o main.o
	${CC} ${LDFLAGS} -o ${TARGET} hello.o main.o
hello.o: hello.c hello.h
	${CC} -c ${CFLAGS} -o hello.o hello.c
main.o: main.c hello.h
	${CC} -c ${CFLAGS} -o main.o main.c
clean:
	rm -f *.o
mrproper: clean
	rm -f ${TARGET}
```

Makefile
Variables spéciales

★★

Variables spéciales (dans les actions)

Une variable spéciale permet de remplacer un ou plusieurs éléments de la règle de manière générique :

- `$@` : nom de la cible
- `$<` : première dépendance
- `$^` : liste de toutes les dépendances
- `$?` : liste de toutes les dépendances plus récentes que la cible
- `$*` : nom de la cible sans extension

Makefile

Exemple avec variables spéciales

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -g
LDFLAGS=-g
TARGET=hello

all: ${TARGET}
hello: hello.o main.o
	${CC} ${LDFLAGS} -o $@ $^
hello.o: hello.c hello.h
	${CC} -c ${CFLAGS} -o $@ $<
main.o: main.c hello.h
	${CC} -c ${CFLAGS} -o $@ $<
clean:
	rm -f *.o
mrproper: clean
	rm -f ${TARGET}
```

Règles d'inférence

★★

Définition (Règle d'inférence)

Une règle d'inférence est une règle générique de production qui permet de mutualiser les règles de production utilisées habituellement. On utilise le symbole % pour désigner un nom générique dans la cible et dans les dépendances.

Règle d'inférence prédéfinies

Il existe des règles d'inférence prédéfinies par `make(1)` :

```
%.o: %.c
	$(CC) $(CPPFLAGS) $(CFLAGS) -c $<

%: %.o
	$(CC) $(LDFLAGS) $< $(LOADLIBES) $(LDLIBS)
...
```

Makefile

Exemple avec règles d'inférence

Exemple (Makefile)

```
CC=gcc
CFLAGS=-Wall -g
LDFLAGS=-g
TARGET=hello

all: ${TARGET}
hello: hello.o main.o
	${CC} ${LDFLAGS} -o $@ $^
hello.o: hello.h
main.o: hello.h
%.o: %.c
	${CC} -c ${CFLAGS} -o $@ $<
clean:
	rm -f *.o
mrproper: clean
	rm -f ${TARGET}
```

Plan

Problématique

Pourquoi générer un Makefile ?

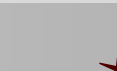
Il est difficile de prendre en compte certaines spécificités du système ou certains choix de développement (debug, production) via les Makefile. Il est alors nécessaire d'avoir un script de configuration (généralement appelé `configure`) qui va analyser le système et les arguments pour générer un Makefile adapté.

Exemple

Pour générer un projet et l'installer sur le système, on procède généralement de la manière suivante :

```
$ ./configure
$ make
# make install
```

Autotools



Autotools

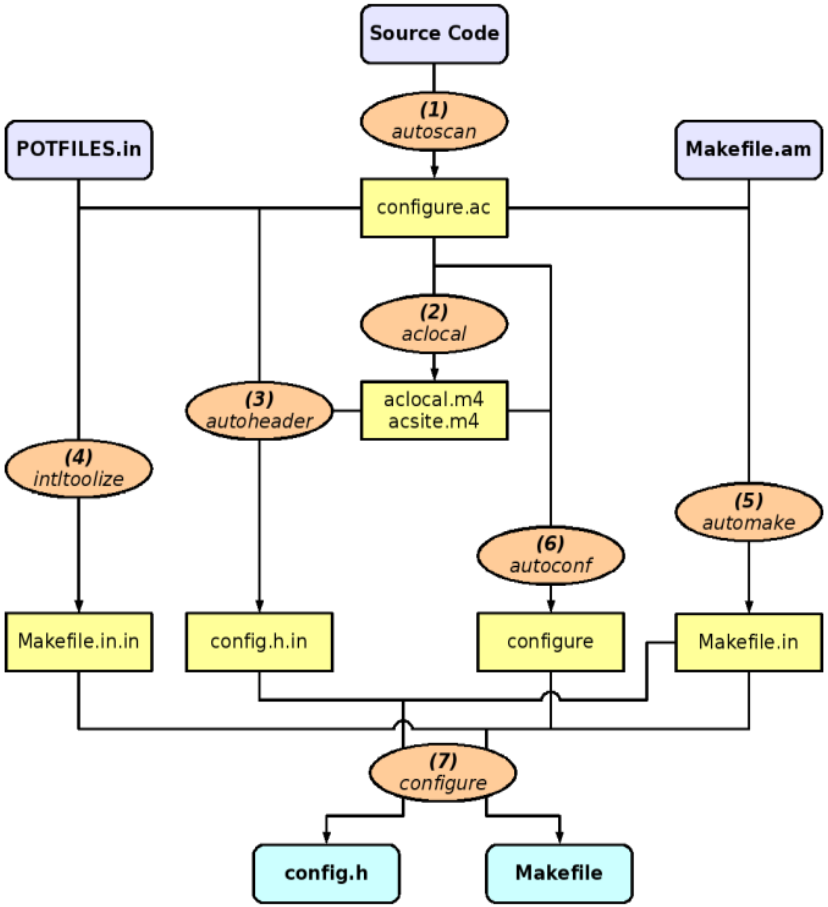
Les autotools sont un ensemble de programmes capables de générer un script `configure` portable et les Makefile associés :

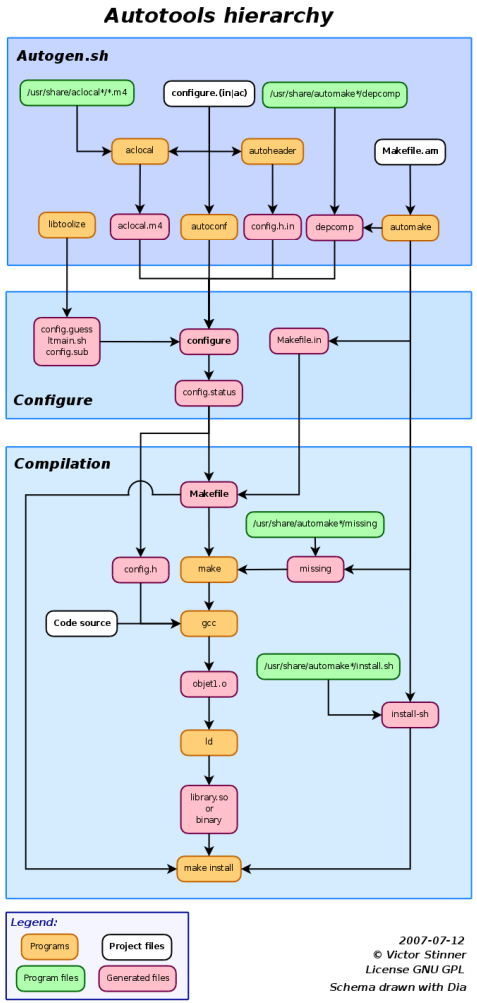
- `automake` : prend des `Makefile.am` et génère des `Makefile.in`
- `autoconf` : prend un `configure.ac` et génère un script `configure`
- `./configure` : prend les `Makefile.in` et génère les `Makefile`

Les autotools sont basés sur le shell (pour le script `configure`) et sur le langage de script M4 (pour les fichiers `Makefile.am` et `configure.ac`)

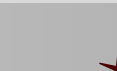
- Principal avantage : Le script `configure` généré peut être exécuté sur n'importe quel Unix (portabilité)
- Principal inconvénient : Cet ensemble d'outils est très complexe à prendre en main et à maîtriser

Autotools


Source Code
(1) autoscan
POTFILES.in
Makefile.am
configure.ac
(2) aclocal
(3) autoheader
aclocal.m4
acsite.m4
(4) intltoolize
(5) automake
(6) autoconf
Makefile.in.in
config.h.in
configure
Makefile.in
(7) configure
config.h
Makefile


Autotools hierarchy
Autogen.sh
/usr/share/aclocal*/*.m4
configure.(in|ac)
/usr/share/automake*/depcomp
aclocal
autoheader
Makefile.am
libtoolize
aclocal.m4
autoconf
config.h.in
depcomp
automake
config.guess
ltmain.sh
config.sub
configure
Makefile.in
config.status
Configure
Compilation
Makefile
/usr/share/automake*/missing
config.h
make
missing
Code source
gcc
/usr/share/automake*/install.sh
objet1.o
install-sh
ld
library.so
or
binary
make install
Legend:
Programs
Project files
Program files
Generated files
2007-07-12
© Victor Stinner
License GNU GPL
Schema drawn with Dia

CMake



CMake

CMake est un programme qui remplace les autotools et le script `configure`. Il génère des Makefile mais est capable de générer d'autres formats de projets. Il utilise des fichiers `CMakelists.txt` qui utilisent un langage très simple avec de très nombreuses commandes pour faire des tâches de base.

Exemple (CMakelists.txt)

```
project(HELLO)
set(CMAKE_C_FLAGS "${CMAKE_C_FLAGS} -Wall -g")
add_executable(hello hello.c main.c)
```

Questions de la fin

À propos des bibliothèques dynamiques

Quels sont les avantages d'une bibliothèque dynamique par rapport à une bibliothèque statique ?

Sixième partie

Manipulation de fichiers

Plan de ce cours

13. Fichiers en C
    - Généralités
    - Descripteur de fichier
    - Flux `FILE*` et `DIR*`

14. Manipulation de fichiers texte
    - Généralités
    - Sélection
    - Modification
    - Affichage
    - Information

Plan

Fichiers en C

Fichiers en C

- «Tout est fichier» : fichiers réguliers, répertoires, périphériques, etc
- Le langage C est lié à Unix : créé au départ pour implémenter Unix
- «Faites de chaque programme un filtre»

Conséquences

Il existe une API générique pour manipuler les fichiers en C. Deux niveaux d'abstraction pour les fichiers :

- Descripteur de fichier : contact direct avec le système d'exploitation via des appels systèmes
- Descripteur de flux : encapsulation des appels systèmes dans des fonctions (à peine) plus haut niveau

Plan

Descripteur de fichier

★★★

Définition (Descripteur de fichier)

Un descripteur de fichier est un entier (`int`) qui représente un fichier dans le système de fichiers. Cet entier est en fait un index dans la table des descripteurs de fichier (unique à chaque processus).

Descripteurs spéciaux

- 0 : entrée standard (`STDIN_FILENO`)
- 1 : sortie standard (`STDOUT_FILENO`)
- 2 : erreur standard (`STDERR_FILENO`)

RTFM : `stdin(3)`

Ouverture d'un fichier

open(2)

★★

open(2)

```
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
```

Ouvre le fichier identifié par `pathname` et renvoie le descripteur de fichier correspondant ou -1 en cas d'échec. L'attribut `flags` peut être une combinaison des valeurs suivantes :

- `O_RDONLY` : Ouvre le fichier en lecture seule
- `O_WRONLY` : Ouvre le fichier en écriture seule
- `O_RDWR` : Ouvre le fichier en lecture/écriture
- `O_APPEND` : Ouvre le fichier en ajout
- `O_CREAT` : Crée le fichier s'il n'existe pas (2è version)
- `O_TRUNC` : Tronque le fichier

Exemple d'ouverture d'un fichier

Exemple

```
int fd; /* file descriptor */

fd = open("foo.txt", O_RDONLY);

if (fd == -1) {
  printf("Error while opening %s!\n", "foo.txt");
  exit(EXIT_FAILURE);
}
```

Création d'un fichier

creat(2)

★★

creat(2)

```
int creat(const char *pathname, mode_t mode);
```

Crée le fichier identifié par pathname. Équivalent à open(2) avec les flags suivants : O_CREAT | O_WRONLY | O_TRUNC

Exemple

```
int fd = creat("bar.h", 0644);

if (fd == -1) {
  printf("Error while creating %s!\n", "bar.h");
  exit(EXIT_FAILURE);
}
```

Fermeture d'un fichier

close(2)

★★

close(2)

```
int close(int fd);
```

Ferme le descripteur de fichier `fd`.

Exemple

```
int fd = open("baz.c", O_WRONLY);

if (fd == -1) {
  printf("Error while opening %s!\n", "baz.c");
  exit(EXIT_FAILURE);
}

do_something_useful_with(fd);

close(fd);
```

Lecture d'un fichier

read(2)

★★

read(2)

```
ssize_t read(int fd, void *buf, size_t count);
```

Lit jusqu'à count octets depuis le descripteur de fichier fd dans le tampon pointé par buf.

Exemple

```
#define BUFSIZE 256
char buf[BUFSIZE]; /* buffer */

int fd = open("passwd", O_RDONLY);

ssize_t s = read(fd, buf, BUFSIZE);
printf("I read %zd bytes from %s.\n", s, "passwd");
```

Écriture d'un fichier

write(2)

★★

write(2)

```
ssize_t write(int fd, const void *buf, size_t count);
```

Lit au maximum count octets dans la zone mémoire pointée par buf, et les écrit dans le fichier référencé par le descripteur fd.

Exemple

```
char *str = "This a generated README file. Fill it.\n";

int fd = open("README", O_WRONLY);

ssize_t s = write(fd, str, strlen(str));
printf("I wrote %zd bytes to %s.\n", s, "README");
```

Exemple complet de lecture d'un fichier ⋆⋆⋆

Exemple

```
char buf[BUFSIZE];
ssize_t sz = 0;

int fd = open("file.txt", O_RDONLY);

while ((sz = read(fd, buf, BUFSIZE)) > 0) {
  do_something_with(buf, sz);
}

close(fd);
```

Plan

Descripteur de flux

★★★

Définition (Descripteur de flux)

Un descripteur de flux (ou plus simplement flux) est un pointeur sur une structure opaque de type `FILE` qui représente un fichier dans le système de fichier. La structure `FILE` encapsule un descripteur de fichier.

Descripteurs spéciaux

- `stdin` : entrée standard
- `stdout` : sortie standard
- `stderr` : erreur standard

RTFM : `stdin(3)`

Ouverture d'un fichier

fopen(3)

★★

fopen(3)

```
FILE *fopen(const char *path, const char *mode);
```

Ouvre le fichier dont le nom est contenu dans la chaîne pointée par `path` et lui associe un flux. L'attribut `mode` est une des chaînes de caractères suivantes et précise le mode d'ouverture :

- `"r"` : Ouvre le fichier en lecture
- `"r+"` : Ouvre le fichier en lecture et écriture
- `"w"` : Tronque le fichier
- `"w+"` : Crée un fichier et l'ouvre en lecture et écriture
- `"a"` : Ouvre le fichier en ajout
- `"a+"` : Ouvre le fichier en lecture et ajout

Exemple d'ouverture d'un fichier

Exemple

```
FILE *fp; /* stream */

fp = fopen("foo.txt", "r");

if (fp == NULL) {
  printf("Error while opening %s!\n", "foo.txt");
  exit(EXIT_FAILURE);
}
```

Fermeture d'un fichier

fclose(3)

★★

fclose(3)

```
int fclose(FILE *fp);
```

Vide et ferme le flux pointé par `fp`.

Exemple

```
FILE *fp = fopen("baz.c", "w");

if (fp == NULL) {
  printf("Error while opening %s!\n", "baz.c");
  exit(EXIT_FAILURE);
}

do_something_useful_with(fp);

fclose(fp);
```

Lecture d'un fichier

fread(3)

★★

fread(3)

```
size_t fread(void *ptr, size_t size, size_t nmemb, FILE
*stream);
```

Lit nmemb éléments de données, chacun d'eux représentant size octets de long, depuis le flux pointé par stream, et les stocke à l'emplacement pointé par ptr.

Exemple

```
#define BUFSIZE 256
char buf[BUFSIZE]; /* buffer */

FILE *fp = fopen("passwd", "r");

size_t s = fread(buf, sizeof(char), BUFSIZE, fp);
printf("I read %zu __elements__ from %s.\n", s, "passwd");
```

Écriture d'un fichier

fwrite(3)

★★

fwrite(3)

```
size_t fwrite(const void *ptr, size_t size, size_t nmemb,
FILE *stream);
```

Écrit `nmemb` éléments de données, chacun d'eux représentant `size` octet de long, dans le flux pointé par `stream`, après les avoir récupérés depuis l'emplacement pointé par `ptr`.

Exemple

```
char *str = "This a generated README file. Fill it.\n";

FILE *fp = fopen("README", "w");

size_t s = fwrite(str, sizeof(char), strlen(str), fp);
printf("I wrote %zu __elements__ to %s.\n", s, "README");
```

Exemple complet de lecture d'un fichier ⋆⋆⋆

Exemple

```
char buf[BUFSIZE];
size_t sz = 0;

FILE *fp = fopen("file.txt", "r");

while (!feof(fp)) {
  sz = fread(buf, sizeof(char), BUFSIZE, fp);
  do_something_useful_with(buf, sz);
}

fclose(fp);
```

Comparaison des deux API

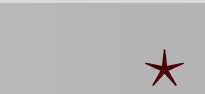
descripteur *vs* flux

Comparaison des deux API

descripteur	flux
`open(2)`	`fopen(3)`
`close(2)`	`fclose(3)`
`read(2)`	`fread(3)`
`write(2)`	`fwrite(3)`
	`feof(3)`

Sortie formattée

fprintf(3)



fprintf(3)

```
int fprintf(FILE *stream, const char *format, ...);
```

Idem que `printf(3)` mais permet d'écrire dans n'importe quel flux.

Exemple

```
fprintf(stdout, "%s a %i ans et mesure %f m\n",
    "Alice", 20, 1.70);

fprintf(stderr, "ERROR! WARNING! KILL YOURSELF!\n");

FILE *fp = fopen("README", "w");
fprintf(fp, "README generated by %s\n", __FILE__);
fclose(fp);
```

Descripteur de répertoire

Définition

Un descripteur de répertoire est un descripteur de flux particulier, dédié au parcours d'un répertoire. C'est un pointeur sur une structure opaque de type `DIR` qui représente un répertoire dans le système de fichier. La structure `DIR` encapsule un descripteur de fichier.

Ouverture d'un flux de répertoire

opendir(3)

★★

opendir(3)

```
DIR *opendir(const char *name);
```

Ouvre un flux répertoire correspondant au répertoire name

Exemple

```
DIR *dir; /* directory */

dir = opendir("/etc");

if (dir == NULL) {
  fprintf(stderr, "Failed to open: %s\n", "/etc");
  exit(EXIT_FAILURE);
}
```

Fermeture d'un flux de répertoire

closedir(3)

★★

closedir(3)

```
int closedir(DIR *dirp);
```

Ferme le flux du répertoire associé à dirp

Exemple

```
DIR *dir = opendir("/usr/bin");

if (dir == NULL) {
  fprintf(stderr, "Failed to open: %s\n", "/usr/bin");
  exit(EXIT_FAILURE);
}

do_something_with(dir);

closedir(dir);
```

Lecture des entrées d'un répertoire

readdir(3)

★★

readdir(3)

```
struct dirent *readdir(DIR *dirp);
```

Renvoie un pointeur sur une structure dirent représentant l'entrée suivante du flux répertoire pointé par dirp

Exemple

```
DIR *dir = opendir("/home");

struct dirent *info = readdir(dir);
while (info != NULL) {
  printf("%s\n", info->d_name);
  info = readdir(dir);
}

closedir(dir);
```

Relations avec les descripteurs de fichiers ⋆

Depuis un descripteur de flux : `fileno(3)`

```
int fileno(FILE *stream);
```

Renvoie le descripteur de fichier associé au descripteur de flux `stream`

Depuis un descripteur de répertoire : `dirfd(3)`

```
int dirfd(DIR *dirp);
```

Renvoie le descripteur de fichier associé au flux de répertoire `dirp`

Attention !

Ne jamais mélanger des opérations sur les flux et des opérations sur les descripteurs de fichiers !

Plan

Commandes filtres ⋆⋆

Commandes filtres

- Toutes les commandes qui suivent prennent en paramètres un fichier texte. Si ce fichier est omis, alors c'est l'entrée standard qui est lue (qui peut donc être la sortie d'un tube...).
- Toutes les commandes qui suivent ont une fonction simple et identifiée. Pour réaliser des opérations plus complexe, on associera plusieurs de ces commandes (notamment grâce à des tubes).

Fichier texte

Le fichier texte peu structuré est le format d'échange préféré quand on utilise les utilitaires de manipulation de fichiers texte. Peu structuré signifie que les informations sont rangées par ligne, éventuellement avec un caractère séparateur pour les différents champs sur chaque ligne.

Plan

Recherche de motif

grep(1)

★★

grep(1)

`grep pattern file`

Recherche le motif `pattern` dans le fichier `file`

- un caractère représente lui-même
- `.` remplace n'importe quel caractère
- `[abc]` remplace `a`, `b` ou `c` et `[a-z]` remplace `a`, ..., `z`
- `^` et `$` remplace le début et la fin de ligne
- Répétitions :
  - `?` : 0 ou 1
  - `*` : 0 ou plus
  - `+` : 1 ou plus

Exemple

```
$ grep zz /usr/share/dict/french
```

Sélection des champs sur une ligne

cut(1)

★★

cut(1)

`cut [options] file`

Sélectionne des champs dans le fichier `file`

- `-d SEP` : utiliser `SEP` en tant que séparateur
- `-f LIST` : sélectionner les champs de `LIST`
  - `N` : N^e champs
  - `N,M` : N^e et M^e champs
  - `N-` : du N^e champs jusqu'à la fin de la ligne
  - `N-M` : du N^e champs jusqu'au M^e champs

Exemple

```
$ cut -d: -f1 /etc/passwd
```

Plan

Remplacement de motif

sed(1)

★★

sed(1)

`sed -e 's/pattern/replacement/' file`

Remplace le motif `pattern` par `replacement` dans `file`

Exemple

```
$ sed -e 's/connection/connexion/' mondevoir.txt
```

Tri d'un fichier

`sort(1)`

★★

`sort(1)`

`sort [options] file`

Tri le fichier `file` (ordre lexicographique par défaut)

- `-n` : ordre numérique
- `-r` : ordre inverse

Exemple

```
$ sort students.txt
```

Élimination des lignes répétées

`uniq(1)`

★★

`uniq(1)`

`uniq [options] file`

Élimine les lignes répétées dans `file`

- `-c` : préfixer les lignes par le nombre d'occurrences
- `-u` : n'afficher que les lignes uniques

S'utilise généralement en sortie de `sort(1)`

Exemple

```
$ uniq answers.txt
```

Plan

Affichage page par page

more(1) or less(1)

more(1)

more file

Affiche le fichier file page par page.

less(1)

less file

Affiche le fichier file page par page mais mieux.

Exemple

```
$ less /usr/share/dict/french
```

Affichage du début d'un fichier

`head(1)`

★★

`head(1)`

`head [options] file`

Affiche les 10 premières lignes de `file`

- `-n K` : affiche les `K` premières lignes

Exemple

```
$ head /var/log/dmesg
```

Affichage de la fin d'un fichier

`tail(1)`

★★

`tail(1)`

`tail [options] file`

Affiche les 10 dernières lignes de `file`

- `-n K` : affiche les `K` dernières lignes

Exemple

```
$ tail /var/log/syslog
```

Plan

Afficher le nombre de lignes/mots/octets

★★

`wc(1)`

`wc(1)`

`wc [options] file`

Compte le nombre de lignes/mots/octets du fichier `file`

- `-c` : affiche le nombre d'octets
- `-w` : affiche le nombre de mots
- `-l` : affiche le nombre de lignes

Exemple

```
$ wc index.html
```

Questions de la fin

Attention les yeux !

Que fait la ligne de commande suivante ?

```
$ cut -d: -f7 /etc/passwd | sort | uniq -c | sort -r -n
```

Septième partie

Processus

Plan de ce cours

15 Processus
- Création et exécution d'un processus
- Signaux
- Contrôle des processus
- Processus et shell

Plan

Définition d'un processus

★★★

Définition (Processus)

Un processus est une opération complexe qui comprend :

- Un ensemble d'instructions à exécuter (programme)
- Un espace d'adressage en mémoire vive (pile, tas, etc)
- Un ensemble de ressources associées (fichiers, sockets, etc)

Un processus est identifié par un PID (*Process IDentifier*)

Exécution

Le processeur est chargé de l'exécution du processus. Sur les systèmes d'exploitation multi-tâches, il est possible d'exécuter plusieurs processus «en même temps». Le noyau est chargé de choisir les processus qui vont être exécutés sur les processeurs, on appelle ce mécanisme l'ordonnacement (*scheduling*).

Ordonnancement ⋆⋆

Ordonnancement

On distingue deux cas :

- Multi-tâche coopératif : les processus décident eux-même de rendre la main au noyau pour qu'il choisissent le prochain processus à exécuter
- Multi-tâche préemptif : le noyau interrompt le processus en cours pour exécuter un autre processus. Deux stratégies :
  - Temps partagé : chaque processus est exécuté pendant une certaine durée à tour de rôle
  - Temps réel : chaque processus est exécuté de manière à ce qu'il termine à une date donnée

Création d'un processus

★★★

Création d'un processus

Un processus peut créer des processus, appelés processus fils et a lui-même un processus père dont l'identifiant est appelé PPID (*Parent Process IDentifier*). Les processus sont ainsi organisés en arbre.

`init(8)`

La racine de l'arbre est le processus `init(8)` qui est lancé par le noyau au démarrage. Il a le PID 1.

Voir le chapitre «Administration système».

RTFM : `pstree(1)`

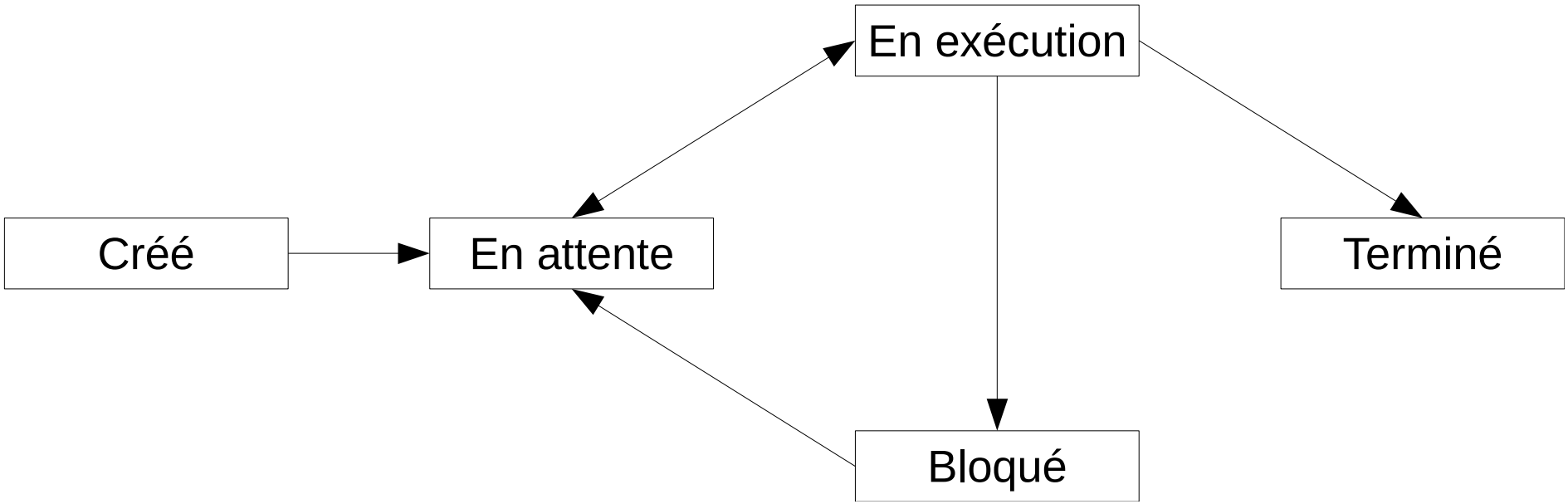
États d'un processus

★★★

États d'un processus

- **Créé** : le processus est créé et attent d'être mis en attente (automatique en temps partagé, pas automatique en temps réel)
- **En attente** : le processus est chargé en mémoire, il est placé dans une file et attent d'être exécuté par un processeur
- **En exécution** : le processus est choisi dans la file d'attente pour être exécuté sur un processeur
- **Bloqué** : le processus a été interrompu (par un signal, etc) ou attend un événement (entrée/sortie, etc)
- **Terminé** : le processus est terminé, soit normalement, soit anormalement, et envoie le code de retour à son père

États d'un processus



Code de retour, zombies et orphelins ⋆⋆

Fin d'un processus

Quand un processus termine, il envoie un code de retour via l'appel système `exit(2)` (ou `return`) que le processus père doit récupérer.

Définition (Processus zombie)

Un processus zombie est un processus mort (terminé) mais dont le père n'a pas encore lu le code de retour. Il existe donc une structure dans le noyau qui conserve ce code de retour. Un processus zombie a toujours son PID.

Définition (Processus orphelin)

Un processus orphelin est un processus dont le père est terminé. Il est alors attaché au processus `init`.

RTFM : `exit(2)`, `exit(3)`

Le système de fichiers /proc ⋆⋆

/proc

Le système de fichier /proc est un système de fichiers virtuel qui contient des informations sur les processus en cours (un répertoire par processus nommé selon le PID) :

- cmdline : ligne de commande complète du processus
- cwd : lien symbolique vers le répertoire courant du processus
- environ : l'environnement du processus
- exe : lien symbolique vers la commande en cours d'exécution
- status : informations sur le processus
- ...

RTFM : proc(5)

Processus, utilisateurs et permissions ⋆⋆

Processus, utilisateurs et permissions

Un processus possède l'UID, le GID et les permissions de l'utilisateur qui lance le processus. Donc, un processus lancé par un utilisateur ne peut pas faire plus que ce que l'utilisateur a le droit de faire.

SUID et SGID

- SUID : quand un programme a le bit SUID, le processus a l'UID et les permissions du *propriétaire* du fichier.
- SGID : quand un programme a le bit SGID, le processus a le GID et les permissions du *groupe propriétaire* du fichier.

Exemple

Le programme `/bin/rm` appartient à `root` et au groupe `root`. Le processus ne peut s'appliquer que sur les fichiers sur lesquels l'utilisateur a la permission d'écriture.

Processus et descripteurs de fichiers ⋆

Processus et descripteurs de fichiers

- Chaque processus possède sa propre table des descripteurs de fichiers.
- Chaque processus possède 3 descripteurs ouverts à sa création : 0 (entrée standard), 1 (sortie standard) et 2 (erreur standard).
- Un processus fils hérite de tous les descripteurs de son père.

Plan

Définition d'un signal

★★

Définition (Signal)

Un signal est une forme limitée de communication inter-processus. Il s'agit d'une notification qui peut être envoyée à un processus pour le prévenir qu'un événement (souvent grave) s'est produit.

À l'envoi d'un signal

Quand un signal est envoyé :

- Le processus est interrompu là où il en est
- Si le processus a renseigné un gestionnaire de signal, il est exécuté
- Sinon, c'est le gestionnaire de signal par défaut qui est exécuté

RTFM : `signal(7)`

Envoi d'un signal

Envoi d'un signal

Un signal peut être envoyé par :

- Un autre processus via la fonction `kill(2)`
- La commande `kill(1)`
- Le clavier à travers un terminal
- Le matériel
- Le système

Gestionnaires de signaux par défaut ⋆⋆

Gestionnaires de signaux par défaut

Il existe un certain nombre de gestionnaires par défaut :

- `Term` : Terminer le processus
- `Ign` : Ignorer le signal
- `Core` : Créer un fichier `core` et terminer le processus
- `Stop` : Arrêter le processus
- `Cont` : Continuer le processus s'il est actuellement arrêté

RTFM : `signal(7)`, `core(5)`

Les principaux signaux

Signaux d'interruption d'un processus

★★

Définition (Interruption d'un processus)

L'interruption d'un processus est la terminaison depuis un terminal.

SIGINT

- Interromp le processus
- Combinaison de touche `CTRL+C`
- Par défaut : `Term`

SIGQUIT

- Interromp le processus et *dump* le processus («*Core Dumped*»)
- Combinaison de touche `CTRL+\`
- Par défaut : `Core`

Les principaux signaux

Signaux de terminaison d'un processus

★★

Définition (Terminaison d'un processus)

La terminaison d'un processus est le passage dans l'état `Terminé`.

SIGTERM

- Demande de terminer le processus
- Par défault : `Term`

SIGKILL (9)

- Tue le processus (termine immédiatement)
- Par défault : `Term` mais impossible à modifier !

SIGCHLD

- Signale qu'un processus fils est terminé
- Par défault : `Ign`

Les principaux signaux

Signaux d'arrêt d'un processus

★★

Définition (Arrêt d'un processus)

L'arrêt d'un processus est la suspension en attendant une reprise.

SIGSTOP

- Arrête le processus temporairement
- Par défaut : `Stop` mais impossible à modifier !

SIGTSTP

- Arrête le processus depuis un terminal par la combinaison `CTRL+Z`
- Par défaut : `Stop`

SIGCONT

- Reprend le processus arrêté
- Par défaut : `Cont`

Les principaux signaux

Signaux d'exception

★★

SIGBUS

- Erreur de bus (mémoire) c'est-à-dire problème d'alignement mémoire, problème d'adresse physique, etc.
- Par défault : `Core`

SIGFPE

- Erreur dans une opération arithmétique («Floating Point Exception»), flottante ou non, c'est-à-dire division par zéro, dépassement de capacité d'un entier signé, etc.
- Par défault : `Core`

SIGSEGV

- Erreur de segmentation («Segfault»), c'est-à-dire problème d'adresse virtuelle, lecture et/ou écriture à l'adresse 0 (`NULL`)
- Par défault : `Core`

Synthèse sur les signaux ★★

Synthèse sur les principaux signaux

Signal	Défaut	Gest.	Clavier
SIGINT	Term		CTRL+C
SIGQUIT	Core		CTRL+\
SIGTERM	Term		
SIGKILL	Term	Non	
SIGCHLD	Ign		
SIGSTOP	Stop	Non	
SIGTSTP	Stop		CTRL+Z
SIGCONT	Cont		
SIGBUS	Core		
SIGFPE	Core		
SIGSEGV	Core		

Plan

La commande ps(1)

ps(1)

`ps [options]`

Affiche les processus en cours avec diverses informations

- `ps aux` : affiche tous les processus en cours (syntaxe BSD)
- `ps -ef` : affiche tous les processus en cours (syntaxe Posix)
- `ps -u jbernard` : affiche les processus de l'utilisateur `jbernard`

Exemple

```
$ ps
  PID TTY          TIME CMD
 2257 pts/1    00:00:00 bash
 2997 pts/1    00:00:48 okular
 3949 pts/1    00:00:00 ps
```

La commande top(1)

top(1)

top
Affiche les processus dynamiquement (toutes les secondes)

Exemple

```
top - 23:58:51 up  4:35,  3 users,  load average: 0.09, 0.18, 0.21
Tasks: 138 total,   2 running, 136 sleeping,   0 stopped,   0 zombie
Cpu(s): 14.0%us, 2.7%sy, 0.0%ni, 79.7%id, 3.7%wa, 0.0%hi, 0.0%si, 0.0%st
Mem:   3955660k total,  1907672k used,  2047988k free,    51984k buffers
Swap:  6530384k total,        0k used,  6530384k free,   752684k cached

  PID USER      PR  NI  VIRT  RES  SHR S %CPU %MEM    TIME+  COMMAND
 2444 jbernard  20   0 1398m 565m  29m S   26 14.6  49:00.30 firefox-bin
 1320 root      20   0  246m  36m 8036 S    4  1.0  11:29.09 Xorg
 2255 jbernard  20   0  326m  31m  16m S    1  0.8   0:16.07 konsole
...
```

La commande `kill(1)` ⋆⋆

`kill(1)`

`kill [-s signal] pid`

Envoie un signal au processus dont le PID est *pid*. Par défaut, le signal envoyé est `SIGTERM`.

- `-s KILL`, `-KILL`, `-9` : envoie le signal `SIGKILL`

Exemple

```
$ kill 2997
```

La commande killall(1)

killall(1)

`killall command`

Envoie un signal au processus correspondant à la commande *command*. Si plusieurs commandes ont été lancées, un signal est envoyé à tous les processus correspondants. Par défaut, le signal envoyé est SIGTERM.

- `-s KILL`, `-KILL`, `-9` : envoie le signal `SIGKILL`

Exemple

```
$ killall okular
```

La commande pidof(1) ⋆⋆

pidof(1)

`pidof command`

Donne le PID du processus correspondant à la commande *command*. Si plusieurs commandes ont été lancées, une liste des PID séparés par des espaces est renvoyée.

Exemple

```
$ pidof okular
2997
$ pidof bash
3903 2206
```

La commande `nice(1)` ⋆⋆

`nice(1)`

`nice -n N command ...`

Exécute une commande avec une priorité modifiée. Ajoute `N` à la priorité par défaut (0). Les priorités vont de -20 (haute priorité) à 19 (basse priorité) et interviennent dans l'odonnancement des processus.

Exemple

```
$ nice -n 19 ./gros-calcul
```

Plan

Lancement de processus en arrière plan ⋆⋆

Lancement de processus en arrière plan

- Une commande lancée dans le shell est toujours en avant plan, c'est-à-dire que le shell rend la main quand le processus est terminé.
- Il est possible de lancer une commande en arrière plan en ajoutant `&` à la fin de la ligne. Le shell rend alors la main à l'utilisateur.
- Le numéro de `job` est indiqué suivi du PID du processus.
- Attention à ne pas confondre `&` et `&&` !

Exemple

```
$ okular &
[2] 3728
```

Suspension d'un processus ⋆⋆

Suspension d'un processus

- Il est possible de suspendre un processus qui s'exécute en avant plan en lui envoyant le signal `SIGTSTP` via la combinaisons de touches `CTRL+Z`.
- Le numéro de `job` est indiqué.

Exemple

```
$ okular
^Z
[3]+  Stopped                 okular
```

La commande bg

★★

bg

bg
Reprend l'exécution du dernier processus suspendu (en lui envoyant le signal `SIGCONT`) et le met en arrière plan

Exemple

```
$ okular
^Z
[3]+  Stopped                 okular
$ bg
[3]+ okular &
$
```

La commande fg

★★

fg

fg
Reprend l'exécution du dernier processus suspendu (en lui envoyant le signal SIGCONT) et le met en avant plan

Exemple

```
$ okular
^Z
[3]+  Stopped                 okular
$ fg
okular
```

Questions de la fin

À propos des signaux

Quel signal permet de prévenir qu'un pipe n'a pas de processus connecté à l'autre bout ?

À propos de `killall(1)`

Comment implémenter une version simplifié de `killall(1)` en shell à l'aide de `kill(1)` et `pidof(1)` ?

Huitième partie

Programmation système

Plan de ce cours

16. Programmation système
    - Duplication et recouvrement de processus
    - Signaux
    - Tubes et redirections

17. Daemons
    - Définition d'un daemon
    - Session et groupe de processus

Plan

Création de processus

★★★

Création de processus

L'API pour créer des processus est principalement composée de deux opérations :

- La duplication : un processus est cloné à l'identique
- Le recouvrement : un processus est remplacé par un autre

Il n'y a donc pas d'opération pour lancer un autre processus directement, il faut pour cela d'abord dupliquer le processus courant, puis dans le fils, recouvrir le processus avec le nouveau processus à lancer.

Duplication de processus

`fork(2)`

★★

`fork(2)`

```
pid_t fork(void);
```

Crée un nouveau processus en copiant le processus appelant. Au retour de l'appel, les deux processus reprennent au même endroit. C'est donc le code de retour de `fork(2)` qui indique si on est dans le fils ou dans le père. Les codes de retour possibles sont :

- -1 : indique une erreur, c'est-à-dire aucun fils n'a été créé.
- 0 : indique qu'on est dans le fils.
- $p > 0$: indique qu'on est dans le père, le code de retour correspondant au PID du fils.

Duplication de processus

Exemple

Exemple

```
pid_t pid = fork();

if (pid == -1) {
  printf("Error!\n");
} else if (pid == 0) {
  printf("I am your son!\n");
} else {
  printf("I am your father!\n");
  printf("Your PID is %d.\n", pid);
}
```

Recouvrement de processus

execve(2)

★★

execve(2)

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

Exécute le programme correspondant au fichier *filename*. Le processus appelant est remplacé par le nouveau, donc cette fonction ne revient jamais, à moins d'une erreur, auquel cas la fonction renvoie -1.

- `argv` correspond au tableau des arguments de la fonction `main`. Son dernier élément doit être `NULL`.
- `envp` correspond au tableau des variables d'environnement, sous la forme `NOM=VALEUR`. Son dernier élément doit être `NULL`.

Recouvrement de processus

Exemple

Exemple

```
char *arg[] =
    { "/bin/cp", "/etc/passwd", "/tmp/passwd", NULL };
char *env[] = { NULL };

int err = execve("/bin/cp", arg, env);
if (err) {
  printf("Error!\n");
}
```

Recouvrement de processus

Variantes

★★

La famille exec*

En plus de l'appel système `execve(2)`, il existe des fonctions qui permettent de simplifier le recouvrement de processus :

- `int execl(const char *path, const char *arg, ...);`
- `int execlp(const char *file, const char *arg, ...);`
- `int execle(const char *path, const char *arg, ..., char * const envp[]);`
- `int execv(const char *path, char *const argv[]);`
- `int execvp(const char *file, char *const argv[]);`

RTFM : `exec(3)`

Attente d'un processus fils

`wait(2)` et `waitpid(2)`

★★

`wait(2)`

```
pid_t wait(int *status);
```

Attends la terminaison d'un des processus fils.

- *status* indique l'état du fils et peut être interrogé par plusieurs macros pour savoir la manière dont le fils s'est terminé (normal, signal, etc).

`waitpid(2)`

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Attends la terminaison du processus fils dont le PID est *pid*.

- *status* a le même rôle que pour `wait(2)`.
- *options* permet de spécifier l'attente d'autres types de changements d'état du fils.

Obtention de PID

getpid(2) et getppid(2)

★★

getpid(2)

```
pid_t getpid(void);
```

Renvoie le PID du processus courant.

getppid(2)

```
pid_t getppid(void);
```

Renvoie le PID du processus père.

Exemple

```
pid_t pid = getpid();
pid_t ppid = getppid();

printf("I am %d and my father is %d.\n", pid, ppid);
```

Terminer un processus normalement

`exit(3)`

★★

`exit(3)`

```
void exit(int status);
```

Termine normalement le processus en cours et envoie la valeur *status* (< 256) au parent. Les fonctions de rappels (*callback*) enregistrées avec `atexit(3)` sont appelées. La fonction `exit(3)` ne revient jamais.

`atexit(3)`

```
int atexit(void (*function)(void));
```

Enregistre une fonction de rappel qui sera appelée à la terminaison du processus.

Utilisation de atexit(3)

Exemple

```
void bye() {
  printf("Bye!\n");
}

int main() {
  atexit(bye);
  exit(EXIT_SUCCESS);
}
```

Exemple complet

Exemple

```
int main() {
  pid_t pid = fork();
  if (pid == -1) {
    printf("Error\n");
  } else if (pid == 0) {
    printf("Child beginning\n");
    do_something_long();
    printf("Child exiting\n");
    exit(0);
  } else {
    printf("Father %d waiting for child: %d\n", getpid(), pid);
    int status;
    pid = wait(&status);
    printf("Father %d resuming after wait: %d\n", getpid(), pid);
  }
  return 0;
}
```

Plan

Envoyer et attendre un signal

kill(2) et pause(2)

★★

kill(2)

```
int kill(pid_t pid, int sig);
```

Envoie le signal *sig* au processus dont le PID est *pid*.

pause(2)

```
int pause(void);
```

Arrête le processus jusqu'à ce qu'un signal soit envoyé. Si le signal est géré, alors la fonction renvoie -1.

Programmer un réveil

`alarm(2)`

★★

`alarm(2)`

```
unsigned int alarm(unsigned int nb_sec);
```

Programme une alarme qui enverra le signal `SIGALARM` dans (au moins) *nb_sec* secondes.

Cas typique d'utilisation

Le programme doit rendre une réponse en un temps donné, on lance le calcul et au moment où le signal est envoyé, on rend la meilleure réponse obtenue jusqu'à présent. Exemple : jeu de dames, jeu d'échecs, etc.

Gestion des signaux

signal(2)

★★

signal(2)

```
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

Installe le gestionnaire de signal *handler* pour le signal *signum*. Le gestionnaire de signal peut être :

- `SIG_IGN` pour ignorer le signal
- `SIG_DFL` pour le gestionnaire de signal par défaut
- Une fonction avec le prototype `void func(int sig);`

Rappel : les signaux `SIGKILL` et `SIGSTOP` ne peuvent être ni ignorés, ni interceptés.

Gestion avancée des signaux

`sigaction(2)`

★★

`sigaction(2)`

```
int sigaction(int signum, const struct sigaction *act,
struct sigaction *oldact);
```

Modifie l'action effectuée par un processus à la réception du signal *signum*.
L'action *act* est installée et l'ancienne action est copiée dans *oldact*.

Différences entre `signal(2)` et `sigaction(2)`

- ☺ `signal(2)` est plus simple d'utilisation
- ☹ `signal(2)` n'a pas le même comportement sur tous les Unix
- ☺ `sigaction(2)` permet une interception plus fine des signaux
- ☺ `sigaction(2)` permet d'avoir un gestionnaire de signaux plus élaboré

Plan

Créer un tube

`pipe(2)`

★★

`pipe(2)`

```
int pipe(int pipefd[2]);
```

Crée un tube. Les deux descripteurs de fichiers placés dans *pipefd* font référence aux deux extrêmités du tube :

- `pipefd[0]` fait référence à l'extrêmité en lecture (sortie du tube)
- `pipefd[1]` fait référence à l'extrêmité en écriture (entrée du tube)

RTFM : `pipe(7)`

Exemple de tube

Exemple

```
int pipefd[2];
char buf;

pipe(pipefd);
pid_t pid = fork();
if (pid == 0) {
  close(pipefd[1]);
  while (read(pipefd[0], &buf, 1) > 0) {
    write(STDOUT_FILENO, &buf, 1);
  }
  close(pipefd[0]);
} else {
  const char *str = "Hello World!\n";
  close(pipefd[0]);
  write(pipefd[1], str, strlen(str));
  close(pipefd[1]);
  wait(NULL);
}
```

Dupliquer un descripteur de fichier

dup(2) et dup2(2)

★★

dup(2)

```
int dup(int oldfd);
```

Duplique le descripteur *oldfd* en utilisant le plus petit descripteur non utilisé. La fonction renvoie le nouveau descripteur.

dup2(2)

```
int dup2(int oldfd, int newfd);
```

Duplique le descripteur *oldfd* dans le descripteur *newfd*, qui est fermé auparavant. La fonction renvoie le nouveau descripteur.

Remarque

Dans les deux cas, le processus possède deux descripteurs sur le même fichier. Toute opération sur un descripteur est donc similaire à la même opération sur l'autre descripteur.

Comment faire une redirection avec dup2(2) ? ⋆

Exemple

```
int fd = open("out.txt", O_WRONLY);

int err = dup2(fd, 1);
if (err == -1) {
  printf("Error in redirection\n");
  exit(EXIT_FAILURE);
}

printf("This will be redirected in the file out.txt\n");

close(fd);
```

Plan

Qu'est-ce qu'un daemon ?

Définition (daemon)

Un daemon (*Disk And Execution MONitor*) est un programme qui tourne en tâche de fond (quasiment) en permanence. Le nom d'un daemon finit généralement par la lettre `d`.

Quelques exemples

- `httpd` : le serveur Web Apache
- `sshd` : le serveur SSH OpenSSH
- `crond` : le planificateur de tâche Cron

Comment devenir un daemon ?

Daemon

Un daemon est généralement attaché au processus `init(8)`, soit directement au démarrage de la machine, soit parce qu'il est devenu orphelin, et qu'il n'est plus attaché à un terminal. En plus, un daemon effectue les actions suivantes :

- Changer le répertoire courant à `/` via `chdir(2)`
- Changer le umask à 0 via `umask(2)`
- Fermer tous les descripteurs de fichiers des flux standard

Plan

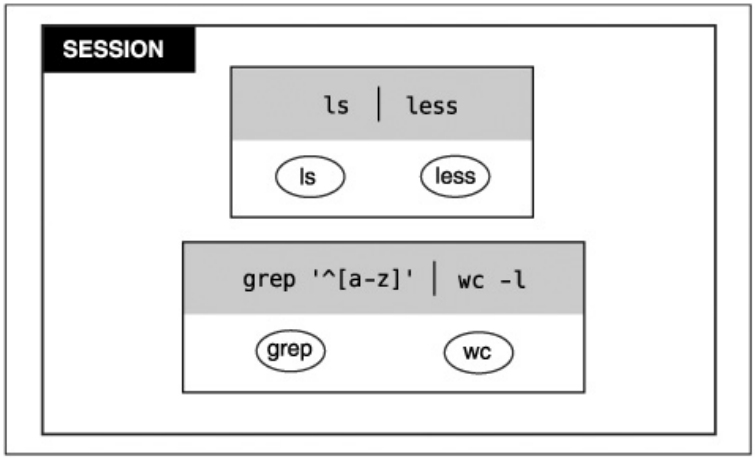
Session et groupe de processus ⋆

Session et groupe de processus

Les processus sont organisés hiérarchiquement en sessions et en groupes :

- Un processus appartient à un groupe de processus
- Un groupe de processus appartient à une session
- Une session est attaché à un terminal (physique ou virtuel)

RTFM : `credentials(7)`



Identifiants de session et de groupe ⋆

Identifiants de session et de groupe

Chaque processus a un identifiant de session (SID, *Session ID*) et un identifiant de groupe de processus (PGID, *Process Group ID*), de type `pid_t`.

`getsid(2)`

```
pid_t getsid(pid_t pid);
```

Obtenir l'identifiant de session du processus dont le PID est *pid*

`getpgrp(2)`

```
pid_t getpgrp(void);
```

Obtenir l'identifiant de groupe de processus du processus courant

RTFM : `credentials(7)`

Groupe de processus

Groupe de processus

Un groupe de processus (ou tâche ou *job*) est un ensemble de processus ayant le même PGID.

- Un groupe de processus est créé par le shell pour exécuter une commande ou un pipeline
- Le processus dont le PID est égal à son PGID est le leader du groupe de processus
- Tous les processus d'un même groupe appartiennent à la même session

Session

Session

Une session est un ensemble de processus ayant le même SID.

- Le processus dont le PID est égal à son SID est le leader de session
- Une session est créée avec `setsid(2)`, le créateur devient leader de session
- Quand le leader d'une session termine, tous les processus de la session reçoivent le signal `SIGHUP` et sont interrompus
- Dans le cas d'un shell, le shell est leader de session

Cas d'un daemon

Lancement d'un daemon

- Le processus principal forke et attends son fils
- Le fils devient leader de session via `setsid(2)`
- Le fils forke et quitte via `exit(2)`, rendant le petit-fils orphelin
- Le petit-fils n'est pas leader de session, ce qui l'empêche d'être rattaché à un terminal
- Le processus principal termine, laissant le petit-fils être le daemon

Questions de la fin

À propos des processus zombie

Comment créer un processus zombie en C ?

Neuvième partie

Administration système

Plan de ce cours

18. Administration système
    - Système d'initialisation
    - Tâches planifiées

19. Gestion des disques et du système de fichiers
    - Montage
    - Formats de système de fichiers
    - Gestion des disques

Plan

Initialisation : `init(8)`

`init(8)`

Le programme `init(8)` est le premier processus du système (PID 1), il est lancé directement par le noyau. Il est chargé de lancer au démarrage des services présents sur le système.

Systèmes d'initialisation

Il existe deux familles de système d'initialisation :

- Famille BSD (BSD, Arch, Slackware)
  - ☺ Simple à mettre en œuvre
  - ☹ Rigide
- Famille SystemV (Debian, RedHat, SuSe)
  - ☹ Complexe
  - ☺ Flexible

Init BSD

Init BSD

Le processus `init(8)` BSD est très simple, il se contente de lancer le script de démarrage `/etc/rc`, qui démarre les `tty` et lance tous les services. Les différents services ont un script de démarrage dans le répertoire `/etc/rc.d`. Le programme `rcorder(8)` calcule l'ordre de démarrage en prenant en compte les dépendances entre les scripts.

Init SystemV

★★

Init SystemV

Le processus `init(8)` SystemV exécute une série d'actions décrites dans le fichier `/etc/inittab`, dont le démarrage des `tty`. Les scripts de démarrage sont placés dans le répertoire `/etc/init.d` puis sont liés dans les répertoires `/etc/rc?.d` en fonction des runlevels.

Init SystemV
Runlevel

★★

Définition (Runlevel)

Un runlevel est une configuration logicielle dans laquelle se trouve le système après avoir démarré. Il existe 7 runlevels : 0-6. Exemples :

- mode mono-utilisateur
- mode multi-utilisateur sans réseau
- mode multi-utilisateur avec réseau
- mode multi-utilisateur avec interface graphique

Les runlevels standard sont :

- 0 : arrêt du système
- 1 : mode mono-utilisateur
- 6 : redémarrage du système

Un système Unix se place généralement (en mode multi-utilisateur) dans un runlevel situé entre 2 et 5.

Nouvelle génération

Améliorations

Il existe des tentatives pour améliorer le système `init(8)`, notamment en permettant l'exécution concurrente des scripts d'initialisation.

- `pinit` : Mandriva
- `insserv` : Debian

De nouveaux systèmes d'initialisation

De nouveaux systèmes sont apparus pour compenser les défauts d'`init(8)` ou pour apporter de nouvelles fonctionnalités.

- `SystemStarter` et `launchd` : MacOSX
- `Upstart` : Ubuntu
- `Mudur` : Pardus
- `systemd` : RedHat

Plan

Planification de tâches répétitives

cron(8)

★★

cron(8)

cron(8) est un daemon de planification de tâches répétitives. Il permet d'exécuter des scripts à certains moment ou à certaines dates. Il sert principalement pour l'administration système. Les tâches sont décrites dans le fichier /etc/crontab et dans les fichiers du répertoire /etc/cron.d/.

Fonctionnement de cron(8)

cron(8) fonctionne de la manière suivante :

1. Il calcule la durée jusqu'à la prochaine action à éxécuter
2. Il s'endort pendant cette durée
3. Il se réveille et vérifie qu'il doit exécuter une action
4. Il exécute l'action puis repart en 1

Fichier de planification de tâches

`crontab(5)`

★★

`crontab(5)`

Le fichier `crontab(5)` contient des tables permettant de décrire les tâches et leur périodicité. Chaque ligne est décomposée en 7 champs :

- Minute (0–59)
- Heure (0–23)
- Jour du mois (1–31)
- Mois (1–12)
- Jour de la semaine (0–6, 0 = dimanche)
- Année (optionnelle)
- Commande à exécuter

Fichier de planification de tâches

Exemple

Exemples

```
# tous les jours à 23h30
30 23 * * * command

# toutes les heures, passées de 5 minutes
5 * * * * command

# tous les premiers du mois à 23h30
30 23 1 * * command

# tous les lundis à 22h28
28 22 * * 1 command
```

Planification des tâches simplifiées

Planification des tâches simplifiées

Il existe quatre répertoires dans lesquels il est possible de déposer des scripts qui seront exécutés à intervalle de temps régulier :

- `/etc/cron.hourly/` : exécutés toutes les heures
- `/etc/cron.daily/` : exécutés tous les jours
- `/etc/cron.weekly/` : exécutés toutes les semaines
- `/etc/cron.monthly/` : exécutés tous les mois

Planification de tâches uniques

at(1)

★★

at(1)

Exécute la commande reçue sur l'entrée standard à une date fixée à l'avance. La commande est transmise au daemon atd(8) qui fonctionne de la même manière que cron(8).

Exemples

```
$ echo "command" | at 1145
$ echo "command" | at 4pm + 3 days
$ echo "command" | at 10am Jul 31
$ echo "command" | at noon tomorrow
```

Plan

Notion de montage ⋆⋆

Système de fichier, partitions et disques

Le système de fichier Unix est composée d'une seule arboresence dont la racine est /. Cette arborescence peut être constituée de plusieurs partitions et/ou disques.

Définition (Montage)

Le montage est le fait d'attacher un système de fichier sur une partition et/ou un disque au système de fichier principal. On dit alors que le système est «monté».

Définition (Démontage)

Le démontage est le fait de détacher le système de fichier sur une partition et/ou un disque du système de fichier principal. On dit alors que le système est «démonté».

Point de montage ⋆⋆

Définition (Point de montage)

Un point de montage est le répertoire racine du système de fichier monté.

Points de montage (FHS)

- `/mnt` : pour les points de montage des systèmes de fichiers temporaires
- `/media` : pour les points de montage des médias amovibles (clefs USB, disque externe, etc).

Exemples

- `/mnt/floppy` : point de montage des disquettes
- `/media/cdrom` : point de montage des disques optiques

Informations sur les systèmes de fichier

`fstab(5)`

★★

`fstab(5)`

Le fichier `/etc/fstab` contient des informations sur les systèmes de fichiers qui peuvent être montés. Chaque ligne indique :

- Le nom du fichier qui décrit le périphérique à monter (exemple : `/dev/sda2`)
- Le point de montage (exemple : `/usr`)
- Le nom du type de système de fichier (exemple : `ext3`)
- Les options de montage

Montage

mount(8)

★★

mount(8)

`mount -t type dev dir`

Monte le système de fichiers situé sur le périphérique *dev* de type *type* au point de montage *dir*.

Exemple

```
$ mount -t ext3 /dev/sda2 /usr
```

Démontage

umount(8)

★★

umount(8)

`umount dir`

`umount dev`

Démonte le système de fichier monté à partir du périphérique *dev* ou au point de montage *dir*

Exemple

```
$ umount /dev/sda2
$ umount /usr
```

Montage à partir d'un fichier ⋆

Montage à partir d'un fichier

La commande `mount(8)` permet de monter un système de fichiers contenu dans un fichier. Pour cela, il est nécessaire de passer l'option `-o loop` qui utilisera un périphérique de boucle (`/dev/loop?`). Un périphérique de boucle est un fichier de périphérique virtuel qui permet d'accéder à un fichier comme à un périphérique en mode bloc. La principale utilité de cette fonctionnalité est de monter une image ISO d'un CD ou d'un DVD.

Exemple

```
$ mount -o loop  /disks/dvd-image.iso /media/dvd
```

Montage d'un système de fichier virtuel ⋆

Montage d'un système de fichier virtuel

La commande `mount(8)` permet de monter un système de fichiers virtuel.
Quelques examples de système de fichiers virtuels :

- `tmpfs` : système de fichiers pour les fichiers temporaires, en mémoire, parfois utilisé pour `/tmp`
- `procfs` : système de fichiers virtuel utilisé pour `/proc`
- `sysfs` : système de fichiers virtuel utilisé pour `/sys` (spécifique Linux)

Exemple

```
$ mount -t proc proc /proc
```

Montage d'un système de fichier réseau

NFS (*Network File System*)

★

Montage d'un système de fichier réseau

La commande `mount(8)` permet de monter un système de fichiers réseau NFS. NFS est un protocole réseau qui permet de manipuler un système de fichier distant comme s'il était local.

Exemple

```
$ mount -t nfs server:/remote/dir /local/dir
```

Voir le cours «Système et Réseau» en L3

Plan

Formats de système de fichiers

Suivant le type de format

Formats de système de fichier

Il existe divers formats de système de fichiers qui décrivent comment sont organisés les fichiers sur le disque.

- Non-journalisés : format les plus simples et historiquement les plus anciens. Exemples : `ext2`, `FAT`, `HFS`
- Journalisés : ajout d'un journal des modifications pour une meilleure fiabilité. Exemples : `XFS`, `ext3`, `ext4`, `NTFS`, `HFS+`, `ReiserFS`
- À snapshot : ajout de la possibilité de faire des sauvegardes instantanée du système. Exemples : `Btrfs`, `ZFS`, `HAMMER`

Formats de système de fichiers

Suivant le système d'exploitation

Formats suivant le système d'exploitation

- Linux : `ext2`, `ext3`, `ext4`, `Btrfs`
- Solaris : `ZFS`
- MacOSX : `HFS`, `HFS+`
- Windows : `FAT`, `NTFS`

Création d'un système de fichiers

`mkfs(8)`

★★

`mkfs(8)`

`mkfs -t type dev`

Crée le système de fichier avec le format `type` sur le périphérique `dev`. En fait, `mkfs(8)` est une interface pour diverses commandes qui sont nommées `mkfs.type` où `type` est le format du système de fichiers.

Ce programme est surtout utilisé à l'installation d'un système Linux, en coopération avec un gestionnaire de partition.

Exemple

```
$ mkfs -t ext3 /dev/sda2
```

Vérifier et réparer un système de fichier

`fsck(8)`

★★

`fsck(8)`

`fsck dev`

Vérifier et éventuellement répare le système de fichier sur le périphérique `dev`. En fait, `fsck(8)` est une interface pour diverses commandes qui sont nommées `fsck.type` où `type` est le format du système de fichiers.

Ce programme est utilisé à chaque démarrage et, au bout d'un certain nombre de montages, il fait une vérification poussée du système de fichier.

Exemple

```
$ fsck /dev/sda2
```

Outils spécifiques pour les formats ext*

tune2fs(8) et dumpe2fs(8)

tune2fs(8)

Ajuste les paramètres du système de fichiers :

- Durées entre deux vérifications poussées, en nombre de montages ou en temps
- Espace réservé à root, en % ou en nombre de blocs
- Option du journal : taille, emplacement
- Nom du volume
- Quota

dumpe2fs(8)

Affiche des informations techniques spécifiques sur le système de fichiers :

- Superblocs
- Groupes de blocs

Plan

Espace disque occupé

du(1) (*Disk Usage*)

★★

du(1)

Calcule la taille occupée par un fichier ou une arborescence dans le système de fichier (par défaut, le répertoire courant)

- `-s` : afficher seulement la somme des fichiers ou répertoires indiqués
- `-h` : afficher dans un format lisible par un humain

Exemple

```
$ du -sh .
7.0M    .
```

Espace disque libre

`df(1)` (*Disk Free*)

★★

`df(1)`

Calcule l'espace disque occupé et libre sur les systèmes de fichiers

- `-h` : afficher dans un format lisible par un humain

Exemple

```
$ df -h
Filesystem                Size  Used Avail Use% Mounted on
/dev/sda2                 144G   53G   84G  39% /
```

Copier et convertir un fichier

dd(1) (*Data Description*)

★★

dd(1)

Copie un fichier en le convertissant.

- `if=` : fichier d'entrée
- `of=` : fichier de sortie
- `bs=` : taille de bloc de lecture et écriture
- `count=` : nombre de blocs à lire et/ou écrire
- `conv=` : options de conversion

Exemples

```
$ dd if=/dev/cdrom of=myCD.iso bs=2048 conv=sync
$ dd if=/dev/sda2 of=/dev/sdb2 bs=4096 conv=noerror
$ dd if=/dev/zero of=file1G.tmp bs=1M count=1024
$ dd if=/dev/sdb2 | ssh user@host "dd of=partition.image"
```

Archivage

`tar(1)` (*Tape ARchive*)

★★

`tar(1)`

Gère des archives.

- Une option de compression parmi :
  - aucune option si on ne veut pas de compression
  - `z` pour compresser avec `gzip`
  - `j` pour compresser avec `bzip2`
- Une action parmi :
  - `c` pour créer une archive avec des fichiers
  - `x` pour extraire les fichiers d'une archive
  - `t` pour lister le contenu d'une archive
  - `r` pour ajouter des fichiers dans une archive
  - `u` pour mettre à jour des fichiers dans une archive
- L'option `f` qui indique qu'on utilise un fichier dont le nom est indiqué

Dixième partie

Programmation système avancée

Plan de ce cours

20 Communication inter-processus
- Introduction à la communication inter-processus
- Tube nommé
- File de messages

Plan

Communication inter-processus

★★★

Définition (Communication inter-processus)

La communication inter-processus (IPC, *Inter-Process Communication*) est l'ensemble des mécanismes qui permettent à des processus concurrents de communiquer. On peut séparer ces mécanismes en deux familles :

- l'échange des données
- la synchronisation

Pourquoi ?

- Partage d'information
- Vitesse d'exécution
- Modularité
- Séparation des privilèges

Mécanismes de communication inter-processus ⋆

Exemples de mécanismes connus

- Fichiers
- Signaux
- Tubes

Autres exemples de mécanismes

- Tubes nommés
- Mémoire partagée
- Sémaphores
- Files de messages
- Sockets
- RPC (*Remote Procedure Call*)

Mécanismes de communication inter-processus ⋆

Exemples de mécanismes connus

- Fichiers
- Signaux
- Tubes

Autres exemples de mécanismes

- Tubes nommés
- Mémoire partagée
- Sémaphores
- Files de messages
- Sockets
- RPC (*Remote Procedure Call*)

API pour les mécanismes IPC

API pour les mécanismes IPC

Il existe deux API pour les IPC :

- L'API SystemV
- L'API POSIX

Cela concerne :

- Mémoire partagée
- Sémaphores
- Files de messages

Nous nous intéresserons à l'API POSIX.

Plan

Qu'est-ce qu'un tubes nommé ? ⋆⋆

Définition (Tube)

Un tube est un mécanisme de type FIFO (*First In First Out*) qui permet à deux processus d'échanger des informations de manière unidirectionnelle.

Définition (Tube anonyme)

Un tube anonyme est un tube qui n'existe que pendant la durée des processus et disparaît ensuite. On le crée en shell avec un `|`.

Définition (Tube nommé)

Un tube nommé est un tube qui persiste tant qu'il n'est pas explicitement détruit. On le crée en shell avec la commande `mkfifo(3)`.

RTFM : `fifo(7)`

Création de tubes nommés

`mkfifo(1)`

★

`mkfifo(1)`

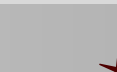
`mkfifo name`

Crée un tube nommé dont le nom est *name*.

Exemple

```
$ mkfifo foo
$ ls -l
total 0
prw-r--r-- 1 jbernard jbernard 0 29 mars  10:51 foo
$ rm foo
```

Utilisation de tubes nommés



Exemple

```
$ mkfifo bar
$ gzip -9 -c < bar > out.gz
--
$ cat file > bar
```

Exemple

```
$ mkfifo baz
$ gunzip -stdout db.gz > baz
--
LOAD DATA INFILE 'baz' INTO TABLE ronde;
```

Création d'un tube nommé en C

mkfifo(3)

mkfifo(3)

`int mkfifo(const char *pathname, mode_t mode);`

Crée le tube nommé dont le nom est `pathname` avec les permissions `mode`. Une fois créé, le tube nommé peut être ouvert avec `open(2)` comme n'importe quel fichier. Pour fonctionner, un tube nommé doit être ouvert par deux processus, un en lecture et l'autre en écriture.

Exemple

```
if (mkfifo("baz", 0644)) {
  perror("mkfifo");
  exit(EXIT_FAILURE);
}

int fd = open("baz", O_RDONLY);
/* ... */
```

Plan

Qu'est-ce qu'une file de messages ? ⋆⋆

Définition

Une file de messages est un mécanisme qui permet à deux processus d'échanger des informations de manière bidirectionnelle. Les messages sont délivrés par ordre de priorité (0–31).

Comparaison file de message / tube

tubes	**file de message**
unidirectionnel	bidirectionnel
non-structuré	semi-structuré
pas de priorité	priorité

Les files de messages POSIX ⋆⋆

Les files de messages POSIX

Les files de messages POSIX sont une implémentation des files de messages pour les systèmes POSIX. Les files POSIX possèdent :

- un nom de la forme `/nom`
- une capacité en nombre de message
- une taille maximum de message

RTFM : `mq_overview(7)`

Ouverture

mq_open(3)

mq_open(3)

```
mqd_t mq_open(const char *name, int oflag);
mqd_t mq_open(const char *name, int oflag, mode_t mode,
struct mq_attr *attr);
```

Ouvre (ou crée) une file de message appelée *name* et le mode *oflag*. Si *oflag* contient `O_CREAT` alors, on utilise la deuxième version et on doit indiquer les permissions *mode* et des attributs de la file.

- *oflags* est un parmi `O_RDONLY`, `O_WRONLY`, `O_RDWR` associé à d'autres drapeaux éventuels
- *attr* permet de fixer la capacité de la file et la taille maximum de message

Cette fonction renvoie un descripteur de file de message.

Fermeture et suppression

mq_close(3) et mq_unlink(3)

mq_close(3)

```
int mq_close(mqd_t mqdes);
```

Ferme la file de message dont le descripteur est *mqdes*.

mq_unlink(3)

```
int mq_unlink(const char *name);
```

Supprime la file de message dont le nom est *name*.

Envoi de message

mq_send(3)

mq_send(3)

```
int mq_send(mqd_t mqdes, const char *msg_ptr,
size_t msg_len, unsigned msg_prio);
```

Envoie le message pointé par *msg_ptr* de longueur *msg_len* avec la priorité *msg_prio* sur la file dont le descripteur est *mqdes*. Si la file est pleine, alors l'appel bloque tant que la file n'est pas vidée par un autre processus.

Réception de message

mq_receive(3)

mq_receive(3)

```
ssize_t mq_receive(mqd_t mqdes, char *msg_ptr,
size_t msg_len, unsigned *msg_prio);
```

Reçoit le message dans le buffer pointé par *msg_ptr* de longueur *msg_len* depuis la file dont le descripteur est *mqdes*. La priorité du message est stocké à l'adresse *msg_prio*. Si la file est vide, alors l'appel bloque tant qu'un autre processus n'envoie pas un message.

Onzième partie

Mémoire virtuelle

Plan de ce cours

21 Mémoire virtuelle
- Généralités
- Mémoire paginée
- Mémoire segmentée
- Segments d'un programme

Plan

Intérêt de la mémoire virtuelle ★★★

Intérêt de la mémoire virtuelle

- La mémoire physique (RAM) d'un ordinateur est limitée. Or, les processus peuvent avoir besoin de beaucoup de mémoire, beaucoup plus que la mémoire physique. Il est donc nécessaire de trouver un mécanisme permettant de faire comme si la mémoire était plus grande que ce qu'elle n'est physiquement.
- La mémoire virtuelle est l'association entre la mémoire physique et une mémoire de masse (disque dur, *swap*) permettant de dépasser la capacité de mémoire physique. Concrètement, c'est l'ensemble des adresses qui peuvent être engendrées par un processeur.
- Il existe une correspondance entre la mémoire virtuelle et la mémoire physique, gérée par le noyau et le matériel (MMU, *Memory Management Unit*).

Plan

Page et adresse virtuelle ⋆⋆

Définition (Page mémoire)

Une page est une zone de mémoire virtuelle contiguë de taille fixe.

Définition (Adresse virtuelle)

Une adresse virtuelle est un couple (p, d) où p est un numéro de page et d un déplacement dans cette page

Exemple

- Si la taille de page est de 4Kio, alors le déplacement d sera codé sur 12 bits.
- Si les adresse virtuelles sont sur 32 bits (c'est-à-dire 4Gio de mémoire adressable), le numéro de page p sera codé sur 20 bits.

Cadre et adresse physique ⋆⋆

Définition (Cadre mémoire)

Un cadre (*frame*) est une zone de mémoire physique contiguë de la même taille qu'une page.

Définition (Adresse physique)

Une adresse physique est un couple (f, d) où f est un numéro de cadre et d est un déplacement dans ce cadre.

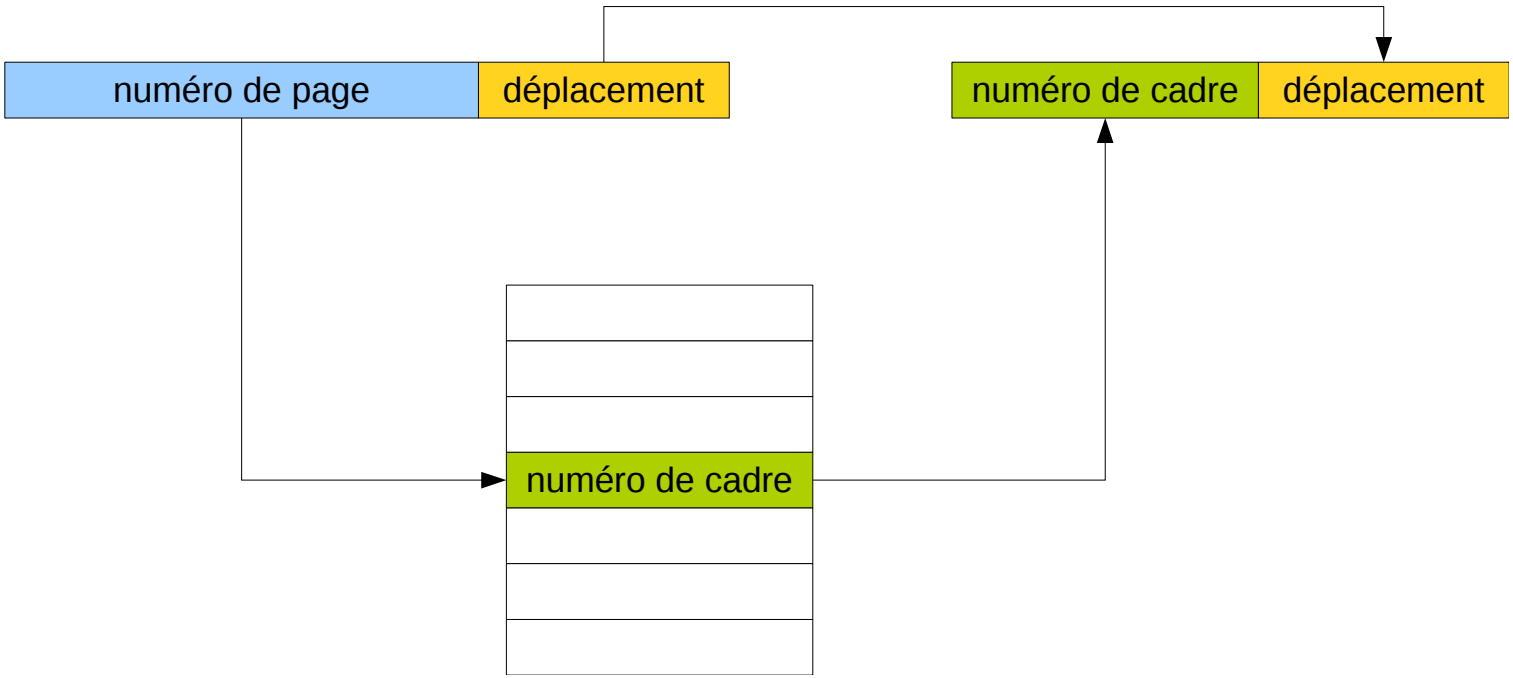
Remarques importantes

- Il peut y avoir plus de pages que de cadres (c'est tout l'intérêt !).
- Les pages qui ne sont pas dans un cadre sont placés sur une mémoire de masse.
- Il faut un mécanisme pour traduire une adresse virtuelle en adresse physique : la translation d'adresse

Translation d'adresse et table de pages ⋆⋆

Définition (Table de page)

La table de page (*page table*) est un tableau indexé par les numéros de page. Chaque case contient le numéro du cadre correspondant (si la page est en mémoire) ainsi que diverses informations (présence en mémoire physique, droits d'accès, etc)



Translation d'adresse et défaut de page ⋆⋆

Défaut de page

Quand le processeur essaie d'accéder à une adresse virtuelle :

1. Le numéro de page correspond à une entrée valide dans la table de pages et la page est chargée dans un cadre : le processeur utilise le numéro de cadre pour former une adresse physique
2. Le numéro de page correspond à une entrée valide dans la table de pages et la page est stockée sur la mémoire de masse : le processeur génère un défaut de page pour que le noyau déplace la page dans un cadre (et mette à jour l'entrée dans la table de pages).
3. Le numéro de page correspond à une entrée invalide dans la table de pages : le processeur génère un défaut de page pour que le noyau trouve un cadre libre et mette le numéro de cadre dans l'entrée de la table de pages.

Problème : si la mémoire physique est pleine ?

Politiques de remplacement ⋆⋆

Politiques de remplacement

Si la mémoire physique est pleine, un algorithme de pagination choisit une page à décharger de la mémoire physique pour la stocker sur le disque (mécanisme de remplacement).
Il existe plusieurs politiques possibles de remplacement :

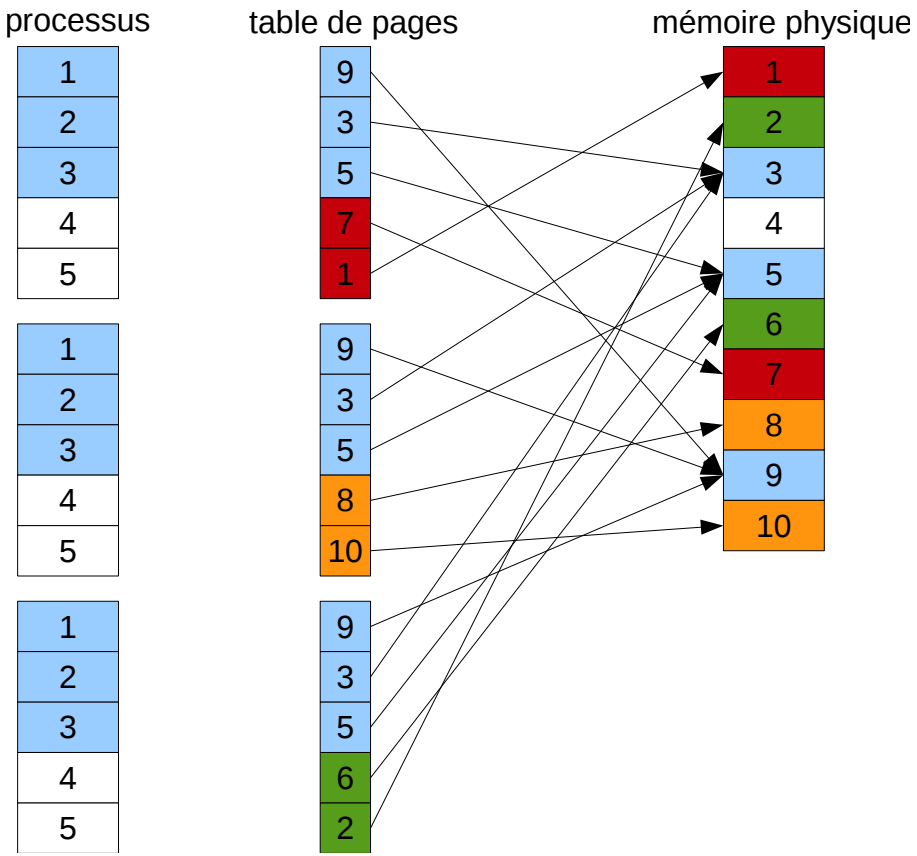
- FIFO (*First In First Out*) : la page qui a été chargé chronologiquement la première est déchargée sur le disque. Malheureusement, cet algorithme est soumis à l'anomalie de Belady : l'augmentation du nombre de cadre ne diminue pas forcément le nombre de défaut de page !
- LRU (*Least Recently Used*) : la page qui a été inutilisé le plus longtemps est déchargée sur le disque. Algorithme le plus fréquemment employé.

Un algorithme de remplacement optimal n'existe pas !

Partage de mémoire

Partage de mémoire

Des processus peuvent partager des cadres en mémoire pour des données qui sont constantes (les instructions par exemple).



Problèmes liés à la pagination

Problèmes liés à la pagination

- Taille de la table de pages : 2^{20} entrées de 10 octets = 10Mio. Solution : paginer la table de pages.
- Temps d'accès à la mémoire : deux accès en mémoire physique pour toute demande de mémoire virtuelle. Solution : TLB (*Translation Lookaside Buffer*), mémoire cache pour les adresses
- Fragmentation interne : les pages ne sont pas utilisées entièrement. Solution : trouver le bon compromis entre la taille des pages et la mémoire physique disponible.
- Phénomène de *trashing* : trop de processus provoque une mise sur le disque trop fréquente. Solution : ajouter de la mémoire physique.

Plan

Segment et adresse virtuelle ⋆⋆

Définition (Segment mémoire)

Une segment est une zone de mémoire virtuelle contiguë de taille variable dédiée à une utilisation par un processus : code, pile, tas, données, etc.

Définition (Adresse virtuelle)

Une adresse virtuelle est un couple (s, d) où s est un numéro de segment et d un déplacement dans ce segment

Table de segments

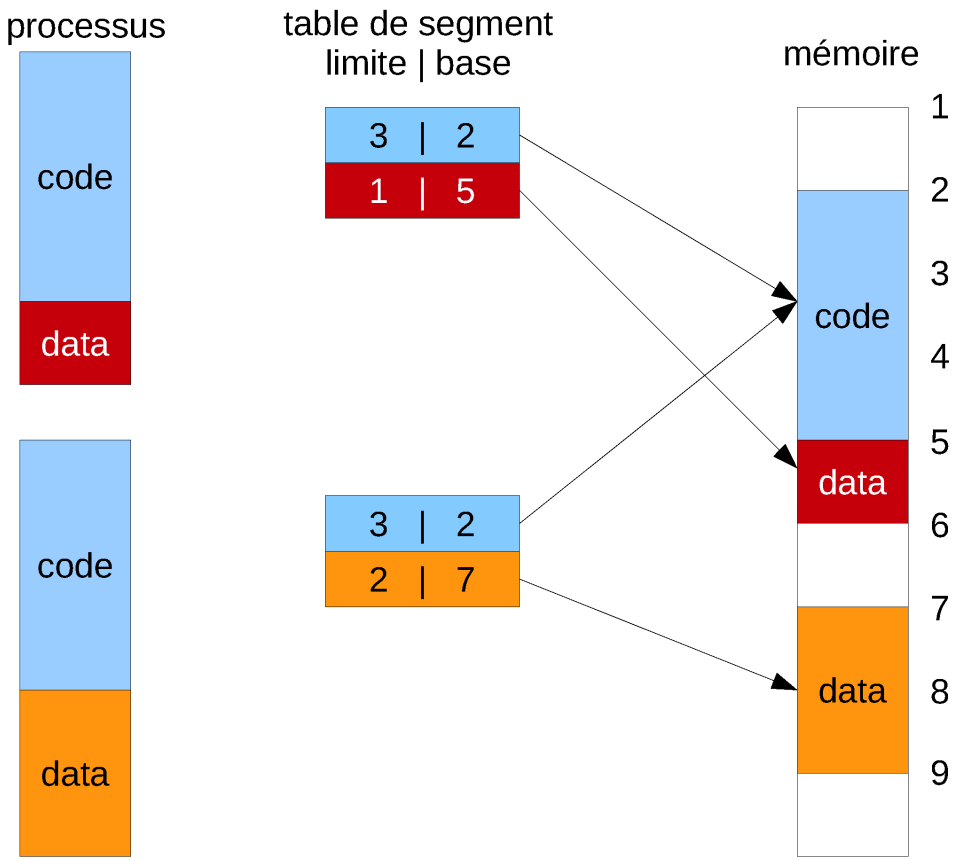
Il existe une table de segments qui fonctionne de la même manière que la table de page. La table de segment contient deux champs :

- une limite qui est la taille du segment, c'est-à-dire le déplacement maximum dans ce segment
- une base qui est l'adresse physique de début du segment

Partage de mémoire

Partage de mémoire

Des processus peuvent partager des segments en mémoire pour des données qui sont constantes (les instructions par exemple).



Considération liés à la segmentation ⋆⋆

Problèmes liés à la segmentation

- Fragmentation externe : l'allocation des segments en mémoire peut empêcher une allocation future.

Algorithme de remplacement

Il est possible de ne stocker sur le disque uniquement les segments qui contiennent des données variables. Les segments qui contiennent du code peuvent être purement et simplement supprimés. Ils seront rechargés directement à partir du programme.

Exemple d'architecture à mémoire segmentée ⋆

Exemple : processeur Intel

Le processeur Intel possède quatre registres concernant les segments :

- CS, (*Code Segment*) : pointe vers le segment contenant le programme courant.
- DS (*Data Segment*) : pointe vers le segment contenant les données du programme en cours d'exécution.
- ES (*Extra Segment*) : pointe vers le segment dont l'utilisation est laissée au programmeur.
- SS (*Stack Segment*) : pointe vers le segment contenant la pile.

Systèmes mixtes

Systèmes mixtes

Dans la réalité, de nos jours, la mémoire est mixte :

- pagination segmentée : la table des pages sera segmentée. Autrement dit, le numéro de page p du couple (p, d) de l'adresse virtuelle sera interprété comme un segment $(s, \bar{p})$.
- segmentation paginée : chaque segment sera paginé. Autrement dit, le champ déplacement d du couple (s, d) de l'adresse virtuelle sera interprété comme un numéro de page et un déplacement $(p, \bar{d})$.

Plan

Segments d'un programme ⋆⋆

Segments d'un programme

Les données d'un programme sont stockés dans différents segments qui seront chargés dans des segments mémoires distincts :

- Segment de code (ou segment de texte) :
  contient les instructions, RX
- Segment de pile (*Stack*) :
  contient la pile (LIFO, *Last In First Out*), RW
- Segment de tas (*Heap*) :
  contient la mémoire allouée dynamiquement (`malloc(3)`), RW
- Segment de données non-initialisées (*BSS*) :
  contient les données non-initialisées, RW
- Segment de données initialisées :
  contient les données initialisées, RW ou R

Exemple : adresse d'objets dans chaque segment

Exemple (mem.c)

```
int a;

int b = 42;

int main() {
  int c;

  int *d = malloc(sizeof(int));

  return 0;
}
```

C'est tout pour le moment...

Des questions ?