ASR2

Michaël Rao

1 Introduction

1.1 Présentation du cours ASR2

— On se concentre sur le S et R de ASR
— Prérequis : C et dans une moindre mesure ASR1.
— Organisation : TP/TD le mercredi à 8h en salle Europe (001), cours le jeudi à 10h.
— Projet(s) avec rendus intermédiaires (1/3 de la note)
— Examen final (2/3 de la note). Peut-être en salle machine, si les contraintes le permettent.
— Questions, suggestions, bugs : `michael.rao@ens-lyon.fr`

1.2 Références

Livres :
— Jean-Marie Rifflet et Jean-Baptiste Yunès, « UNIX : programmation et communication », Dunod, 2003.
— Andrew S. Tanenbaum et Herbert Bos, « Modern Operating Systems, 4th Ed », Pearson 2015.
— Et beaucoup d'autres...

Les pages du manuel (RTFM !)
— Pour avoir le manuel d'une commande, fonction... : `man <terme>` dans un terminal
— Attention : il y a plusieurs sections dans les pages man. Regardez dans la page manuel de man : `man man`
— Certains termes ont des entrées dans plusieurs sections.
— Pour accéder une section particulière : `man <section> <terme>`
— Pour si on ne sait pas : `man -a <terme>`

1.3 Buts du cours

— Comprendre ce qui se passe entre la machine (le matériel, la partie A de ASR) et un processus (que vous allez programmer dans la vraie vie). Dans cette superposition de couches, le système d'exploitation (OS = Operating System) joue un rôle important.

Couches : Matériel / OS / (librairies) / Processus

— Comprendre comment coopèrent les différents processus. Une chose importante est synchronisation et la communication inter-processus : au sein d'une même machine ou entre différentes machines (notamment la communication réseau).
— On n'abordera pas directement les algorithmes utilisés dans la conception OS. Le but ici n'est pas de vous apprendre à programmer un OS, mais de savoir opérer avec lui. Néanmoins, certaines problématiques seront abordés indirectement.
— Tous les exemples/exercices se feront sous GNU/Linux, et on programmera en C. La plupart des choses montrées respectent la norme POSIX, et donc fonctionnent de manière très similaire sous les autres systèmes respectant cette norme (autres Unix, OSX, Windows...).

1.4 Un OS : pourquoi ?

Que fait un système d'exploitation ?
— interface avec le matériel
— gestion de la mémoire
— gestion des périphériques
— gestion des interruptions
— gestion des système de fichiers
— ...

Ces taches sont très dépendantes du matériel, et souvent fastidieuses. L'OS est là pour nous simplifier la vie.

1.5 Que fait un système d'exploitation « moderne » ?

— multi-tâches : plusieurs processus peuvent tourner en même temps
— multi-utilisateurs : il peut y avoir plusieurs utilisateurs différents qui l'utilisent.
— gestion utilisateurs, groupes d'utilisateurs, et droits.
— il y a souvent un utilisateur particulier, le super-utilisateur (root, administrateur) qui a tous les droits.
— gestion du réseau

— essayer d'être "compatible" avec les autres OS : respect de normes.

— les interfaces graphiques sont généralement des sur-couche (e.g. : X). Les environnements graphiques sont encore des sur-sur-couches (gnome, kde, ...). On passe par des API plus haut niveau pour interagir avec eux. (Ce n'est pas le but du cours.)

1.6 Systèmes "Unix"

Introduction d'Unix (1969-, K. Thompson & D. Ritchie...)
— Rompt avec les OS propriétaires, et les programmes monolithiques.

— Introduit conjointement avec le C (1969-, D. Ritchie & B. Kernighan)
C ∼ assembleur universel.
— Rapidement, plusieurs branches de développement :
Unix, BSD, et système propriétaires (Xenis, AIX, System V...)
— Assez rapidement, une volonté de normalisation :
norme POSIX (1988-)
— Philosophie : être modulaire . Préférer des programmes simples qu'on peut composer via les entrées/sorties standards.

1.7 GNU / Linux

— GNU : "GNU is Not Unix" : projet d'OS libre (1983- R. Stallman)
— Logiciel libre : code source disponible, que l'on peut modifier et redistribuer (licence GPL "GNU Public Licence", ou similaire)
— Linux : Un noyau (cœur de l'OS) Unix libre. Initié par Linus Torvalds, inspiré de Minix
— Autres Unix libres : Minix, BSD*, Hurd...

1.8 Écosystème...

Un "écosystème" vivant dans un ordinateur c'est un ensemble de programmes en exécution (les processus), utilisant les mémoires pour lire les données à traiter, pour stocker des résultats intermédiaires de calcul, et pour écrire les résultats finaux. Ces processus peuvent éventuellement interagir entre eux, ou utiliser des périphériques pour interagir avec le monde extérieur (clavier, écran, souris...).

Les processus n'interagissent pas directement avec le matériel, ils passent par l'OS pour y avoir accès.

1.9 Mémoires

Deux types importantes de mémoire :
— la mémoire vive, "volatile"
  — non pérenne
  — accès très rapide. il y a différent niveaux, qui sont plus ou moins accessibles : registres, caches du processeur, mémoire RAM (les "barrettes", ex : DDR3)
— la mémoire "non volatile"
  — pérenne
  — disque dur mécanique, SSD, clef USB...
  — accès (beaucoup) plus lent

La mémoire non volatile est organisée sous forme arborescente, avec des fichiers et des répertoires : l'arborescence des fichiers.

2 Arborescence de fichiers et Shell

2.1 Shell

Shell : interface textuelle entre l'humain et l'OS.
Votre premier objectif : maîtriser le shell

Fonctionnement d'un shell
— prompt : attente d'une commande
`commande [argument1] [argument2] ...`
— on entre une commande, éventuellement avec arguments, et on appuie sur "entrée".
— le shell exécute la commande, puis rend la main quand la commande est terminée.
— pour quitter "proprement" un shell : `exit` (ou ctrl+d sur certains shells).
Note : si on veut un argument avec des espaces, le mettre entre guillemets " " ou ' '

2.2 Shell / terminal

Un shell se lance au travers d'un terminal. Sous un environnement graphique, on lance un shell généralement en lançant un "terminal" ou "console". Il y a donc souvent confusion entre shell et terminal.

Il existe plusieurs shells différents. Un des plus courant sous GNU/Linux est `bash`. Les shells récents ont des fonctionnalités qui simplifient la vie : historique, édition, auto-completion (avec tabulation). Ils ont aussi des fonctionnalité de scripts plus poussés.

Permet de composer facilement des processus via les redirections d'entrée/-sorties.

On peut faire des scripts en shell. Une suite de commandes, avec des tests, branchements, etc. Il s'agit donc d'une langage de programmation (interprété) en lui même.

2.3 L'arborescence des fichiers

Les fichiers sont organisés sous forme arborescente.
— La racine : `/`
— Deux principaux types de fichiers :
  — les fichiers standards = fichiers réguliers
  — les répertoires (ou dossiers).
  — (Il en existe d'autres : liens, fifo... à suivre)
— Chaque fichier possède :
  — un propriétaire et un groupe propriétaire

  - un ensemble de droits (lecture/écriture/exécution) pour l'utilisateur, le groupe, et le reste du monde.
  - une date de création, de modification, de lecture.
- Fichiers commençant par un point : fichiers cachés.

Un fichier « standard » est une chaînes d'octets.
- un octet = 8bits (un entier entre 0 et 255)
- il a une taille : le nombre d'octets
- cela peut être un programme exécutable (en langage machine), un fichier texte (un code source...), un fichier multimédia...

Répertoires spéciaux :
- `..` : répertoire parent
- `.` : répertoire courant

Chemin de la racine à un fichier : chemin absolu
- `/home/mrao/Documents/cours.pdf`

Chemin du répertoire courant à un fichier : chemin relatif
- `Documents/cours.pdf`
  = `./Documents/cours.pdf`

2.4 Organisation typique sous Unix/Linux

- /home : les répertoires des utilisateurs
- /root : le répertoire "home" du super-utilisateur
- /bin et /usr/bin les programmes (les "binaires") ;
- /sbin et /usr/sbin : les binaires système
- /lib et /usr/lib : librairies
- /usr : ressources système
- /etc : fichiers de configurations
- /dev : fichiers spéciaux (ressources, périphériques)
- /tmp : un répertoire pour les fichiers temporaires
- /var : données variables
- ...

2.5 Les shell : premiers pas...

Le shell permet de naviguer dans l'arborescence de fichiers, modifier les droits, faire des opérations simples.
- `ls` : liste les fichiers
  - option `-a` : affiche également les fichiers cachés
  - option `-l` : format long (droits, taille, propriétaire...)
- `cd` *`rep`* : entrer dans le répertoire *rep*
  - `cd ..` : retour au répertoire parent

2.6 Autres commandes de base

- s : liste les fichiers

    - options :
    - -a : affiche les fichiers cachés
    - -l : affiche les informations complètes : propriétaire, taille, droits...
    - -R : liste récursive
- cat <fichier> : affiche le contenu d'un fichier.
- touch <fichier> : crée un fichier (vide) si il n'existe pas, sinon il met sa date de modification à jour.
- rm <fichier> : efface un fichier (ATTENTION : irréversible!)
- cp <fichierA> <fichierB> : copie le fichierA vers le fichierB
    - l'option -R ou -a permet de copier récursivement (pour les répertoires)
- mv <fichierA> <fichierB> : déplace/renomme le fichierA vers le fichierB
- ln : créer un lien
- cd <répertoire> : changer de répertoire
- pwd : affiche le répertoire courant
- mkdir <répertoire> : crée un répertoire
- rmdir <répertoire> : efface un répertoire. attention, il doit être vide.
    - pour effacer un répertoire non vide (ATTENTION : irréversible!) : rm -rf <rep>
- chmod <droits> <fichiers> : change les droits d'un fichier
- chown <user> <fichier> : change le propriétaire de fichier
- chgrp <groupe> <fichier> : change le groupe propriétaire

2.7 Un peu plus sur les droits des fichiers

```
mrao@meshuggah:~/test$ ls -la
total 32
drwxr-xr-x  4 mrao users 4096 dec.  29 22:40 .
drwx------ 61 mrao users 4096 janv.  9 17:25 ..
-rwxr-xr-x  1 mrao users 6656 dec.  29 13:37 programme
-rw-r--r--  1 mrao users  173 dec.  29 13:37 programme.c
drwxr-xr-x  2 mrao users 4096 dec.  29 13:39 sousrep
mrao@meshuggah:~/test$
```

premier champ : un sous ensemble de drwxrwxrwx

- d : répertoire
- 1er triplet (rwx) : droits pour l'utilisateur (ici, mrao)
- 2eme triplet : droits pour les utilisateurs du groupe (users)
- 3eme triplet : droits pour le reste du monde
- r : droit de lecture
- w : droit d'écriture
- x : droit d'exécution (pour les répertoires : droit d'entrer)

Pour changer les droits : `chmod`

`chmod <droits> <fichier>` :

où droits = <qui><+-><permissions>

- qui : sous ensemble de u (user) g (group) o (others) a (all)
- + ajoute les permissions, - enlève.
- permissions : sous ensemble de
    - r (read)
    - w (write)

    - x (execute)
    - X (execute, pour les répertoires)
  - option `-R` : le faire de façon récursive

2.8 Un peu plus sur les droits : en octal

Les droits sont parfois représentés par un nombre de 3 chiffres, de 0 à 7 (en octal, 3 bits par caractères)

1er chiffre : droits pour l'utilisateur, 2eme chiffre : droits pour le groupe, 3eme chiffre : droits pour reste du monde.

Chaque chiffre est une somme :
- 4 pour le droit de lecture,
- 2 pour le droit d'écriture,
- 1 pour le droit d'exécution.

4eme chiffre : modes spéciaux

```
mrao@meshuggah:~/test$ ls -l
total 0
\begin{itemize}
\item w-r--r-- 1 mrao mrao 0 dec.  29 15:23 fichier
\end{itemize}
mrao@meshuggah:~/test/test2$ chmod 762 fichier
mrao@meshuggah:~/test/test2$ ls -l
total 0
\begin{itemize}
\item wxrw--w- 1 mrao mrao 0 dec.  29 15:23 fichier
\end{itemize}
```

Modes spéciaux :
- set-uid :
  - `s` dans `chmod` et `ls -l` (dans le groupe "user")
  - en octal : 04000
  - pour les exécutable
  - lors de l'exécution, prend l'uid du propriétaire du fichier
  - utile pour les programmes systèmes. Par exemple : `passwd` doit pouvoir modifier des fichiers systèmes, qu'un utilisateur ne doit pas pouvoir modifier en temps normal.
- set-gid :
  - en octal : 02000
  - pour les exécutable
  - `s` dans `chmod` et `ls -l` (dans le groupe "group")
  - lors de l'exécution, prend le gid du propriétaire du fichier
- sticky-bit :
  - en octal : 01000
  - pour les exécutable
  - `t` dans `chmod` et `ls -l`

— empêche que quelqu'un puisse effacer le fichier d'un autre
— utile pour `/tmp/`

2.9 Liens

Unix supporte des liens. Il y a deux types de liens, fondamentalement différents.

— Lien symbolique : un "pointeur" vers un autre fichier.
Il s'agit d'un type de fichier spécial.
Commande shell : `ln -s`. Appel système : `symlink`

— Lien physique : fichier correspondant à la même zone sur le disque qu'un autre.
Commandes shell : `ln`, `link`. Appel système : `link`

Plus de précisions sur les liens physiques :

Un "i-node" est un objet sur le disque qui donne les informations sur un fichier (type, propriétaire, droits, dates, et pointeur vers les données), mais pas le nom.

Un fichier régulier dans l'arborescence est en fait un lien physique vers un "i-node" Quand on crée un lien physique, on crée un autre lien vers le même i-node Un i-node est physiquement effacé du disque quand plus aucun lien physique (et aucun descripteur de fichier) ne pointe vers lui.

Le deuxième champ de `ls -l` est le nombre de lien physique vers l'i-noeud pointé par le fichier. Pour voir les informations d'un i-node : `stat` *`fichier`*

2.10 Autres fichiers spéciaux

— Fichier périphérique (device file).
Correspond à un périphérique
Généralement situé dans `/dev/` Deux types : caractères et bloc

— Fichiers tubes (ou fifo).
Pour créer un fichiers tube : `mkfifo`
(On reparlera de tubes au moment de la communication inter-processus.)

2.11 Un peu plus sur les système de fichiers

— L'arborescence des fichiers est un "patchwork" de systèmes de fichiers.
— Un système de fichier correspond généralement à une partition sur un disque sur.
— Il existe plusieurs types de système de fichier : FAT et dérivées (de Microsoft DOS/vieux Windows), NTFS (Windows NT et après), EXT2 et supérieurs (Linux)...
— Ce qui change : l'encodage des droits, la journalisation, le versionage
— Il existe des systèmes de fichiers qui ne correspondent pas à des partitions physiques : certains systèmes générés par l'OS, des systèmes de fichiers réseau...

— `mount` permet de voir tous les systèmes de fichiers montés, et de les monter
— `df` : permet de voir l'utilisation des systèmes de fichiers

2.12 Utilisateurs et groupes

Chaque utilisateur a :
— un nom (une chaîne de caractère)
— un numéro (UID = User IDentifier)
— un ou plusieurs groupes
— un répertoire HOME, généralement : `/home/<user>`
— un mot de passe, stocké de façons hashée.
`root` est le "super-utilisateur". Il a tous les droits. Son UID est 0.
Les informations des utilisateurs sont stockées (généralement) dans /etc/-passwd et /etc/shadows

Chaque groupe a un numéro (GID = Group IDentifier).
`useradd/adduser`, `userdel/deluser` : gestion des utilisateurs (en root)
`passwd` : changement du mot de passe

2.13 Les processus

Chaque processus a :
— un numéro (le PID = Process IDentifier)
— un père, généralement le processus qui l'a lancé.
— un utilisateur (généralement, celui qui l'a lancé)
— une zone mémoire qui lui a été attribué. Il peut en demander plus au système.
— certains processus peuvent être en attente.
— une entrée standard, et une sortie standard et une sortie erreur.
— À sa fin, un processus renvoie un code retour : un entier, généralement 0 s'il n'y a pas d'erreur, et $\neq 0$ sinon.

2.14 Lancer des commandes/processus dans un shell

— Certaines commandes sont interprétés directement par le shell, les builtin (comme `cd`). D'autres correspondent à des programmes exécutables (généralement situés dans `/bin/` ou `/usr/bin/`).
— Le shell va chercher les programmes dans les répertoires listés dans la variable d'environnement PATH. (Pour voir les répertoires dans PATH : echo $PATH)
— S'il le trouve, il l'exécute (le processus se lance). Sinon il renvoie un message d'erreur.
— Pour lancer un programme dans le répertoire courant (ou tout autre répertoire qui n'est pas dans PATH) il faut spécifier le répertoire avant le nom du programme.

```
mrao@meshuggah:~/test$ ls -l
\begin{itemize}
\item wxr-xr-x 1 mrao mrao 6552 dec.  29 16:46 hello
\item w-r--r-- 1 mrao mrao   68 dec.  29 16:46 hello.c
\end{itemize}
mrao@meshuggah:~/test$ hello
bash: hello : commande introuvable
mrao@meshuggah:~/test$ ./hello
Hello
mrao@meshuggah:~/test$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games
mrao@meshuggah:~/test$ PATH=$PATH:.
mrao@meshuggah:~/test$ echo $PATH
/usr/local/bin:/usr/bin:/bin:/usr/local/games:/usr/games:.
mrao@meshuggah:~/test$ hello
Hello
```

2.15 Entrées/sorties standard

Quand un processus s'exécute dans un terminal :
— l'entrée standard est (par défaut) l'entrée du terminal (le clavier)
— la sortie standard est (par défaut) affichée dans le terminal.
— la sortie erreur est (par défaut) affichée dans le terminal.

Le shell permet de facilement rediriger ces entrées/sorties.

2.16 Rediriger les E/S standards

— `commande > ficher` : redirige la sortie standard de la commande vers le fichier (écrase le fichier)
— `commande >> fichier` : redirige la sortie standard de la commande vers le fichier (rajoute à la fin du fichier)
— `commande 2> fichier` : redirige la sortie erreur de la commande vers le fichier
— `commande1 | commande2` : la sortie standard de commande1 sera regirigée vers l'entrée standard de commande2
— `tee` *`fichier`* : copie entrée standard sur la copie standard et *fichier*
— `commande < fichier` : l'entrée standard sera lue depuis le fichier
— `commande << EOF` : le shell va lire l'entrée standard, jusqu'à ce qu'il lise EOF. Ce qui est lu sera envoyé dans l'entrée standard de commande.

2.17 Quelques commandes utiles

— `sleep x`; attend x seconde (utile pour les scirpts)
— `echo` : affiche les arguments
ex. : `echo hello world` affiche : `hello world`
— `less` : permet de se déplacer dans le texte
ex. : `ls -l /etc/ | less`
— `head/tail` : affiche le début/fin de l'entrée
— `sort` : trie les ligne
ex. : `cat /etc/passwd | sort`
— grep : afficher les lignes correspondant à un motif donné
ex. : `cat /etc/passwd | grep mrao`
— `sed` : fait des recherches / remplacements

— `awk` : un truc qui fait mieux que grep / sed, mais encore plus compliqué.

2.18 Voir/gérer les processus :

Commandes shell pour voir/gérer les processus :
— `ps` affiche la liste des processus
exemple : `ps faux`
— `top` affiche la liste des processus dynamiquement
— `kill` *`pid`* : tue un processus de PID *pid* (on en reparlera dans la partie "Signaux")

2.19 Variables du shell

— Le shell manipule des variables.
— Exemples : HOME, USER, PATH
— Affecter une variable :
  — `VARIABLE=affectation`
— déréférencer une variable : la faire précéder par $
  — Ex : pour afficher une variable : `echo $VARIABLE`
— `set` : affiche toutes les variables

Certaines variables sont persistantes : les variables d'environnement. Elles seront transmises aux fils
— `export` : exporte la variable (les rend persistantes)
— `env` : affiche toutes les variables d'environnement.
En fait, chaque processus (et donc, en particulier, le shell) dispose de variables d'environnement ; *export transforme une variable interne du shell en variable d'environnement* On verra par la suite comment accéder et gérer ces variables en C.

2.20 Variables spéciales du shell

— $? : code retour de la précédente commande
— $$: PID du shell
— $! : PID du dernier processus lancé en arrière plan
— ~ : interprété par le shell comme le répertoire HOME
— ~user : interprété par le shell comme le répertoire HOME de l'utilisateur user

2.21 Jokers et echappements

— `*` dans un nom de fichier : n'importe quelle chaîne de caractère
— `?` : exactement un caractère

Pour qu'un caractère spécial ne soit pas interprété par le shell
— \ : exemple `echo \*`

— ' : exemple `echo '*'`
— " : exemple `echo "*"`
  — le shell interprète les $ et certains \dans des chaînes entre "

Le caractère "espace" peut être aussi echappé, pour ne pas séparer les arguments

2.22 Processus en arrière plan

— `commande &` lance un processus, mais le shell n'attend pas la fin du processus pour rendre la main.
— ctrl + z : stoppe un processus
— `jobs` : liste les taches (jobs) en cours d'exécution dans le shell
— `bg` : passe une tache en arrière plan (similaire à &)
— `fg` : passe une tache en premier plan (le shell rend la main au job)
— `%i` : identifie le job numéro i du shell.

2.23 `nohup`

— Si on termine un shell, tous ses jobs seront arrêtés. (Sauf si ils se sont détachées par eux même, à voir plus tard.)
— Pour qu'un processus survive à la mort de son père, on peut le lancer précédé de la commande `nohup`

2.24 Scripts : enchaînement et composition des commandes

— *commande1* ; *commande2*
  execute `commande1`, puis `commande2`
— (*listedecommandes*)
  crée un groupement de commande
— *commande1* `` `commande2` ``
  la sortie de `commande2` est donnée en argument à `commande1`

  Sur certains shells, on peut également faire ceci :
  commande1 `$(`*commande2*`)`
— *commande1* && *commande2*
  execute `commande1`, puis `commande2` si `commande1` réussi (i.e. renvoie 0)
— *commande1* || *commande2*
  execute `commande1`, puis `commande2` si `commande1` échoue
— `if` *condition* ; `then` *commande2* ; `else` *commande3* ;`fi`
`true` : réussi toujours `false` : échoue toujours

2.25 Scripts : tests

`test` *expression* permet de tester une expression conditionnelle.

Note : sur certains shells, c'est équivalent à `[` *expression* `]`

expression construite avec `( ) && || !` et des expression élémentaires.

Expression élémentaire (exemples) :
— tester si un fichier existe : `-e` *fichier*
— tester si un fichier est un répertoire : `-d` *fichier*
— tester si un fichier est un fichier régulier : `-f` *fichier*
— tester si un fichier est lisible : `-r` *fichier*
— tester si deux chaînes de caractères sont égales : *chaine1* `=` *chaine2*
— tester si expression numériques sont égales : *chaine1* `-eq` *chaine2*

2.26 Scripts : évaluer une expression

`expr` *expression* permet d'evaluer une expression

expression construite avec `( ) + - * / % = >=` ... et des expressions élémentaires (entiers...)

Attention aux echappements : `expr \( 2 + 3 \) \* 5`

Sous bash on peut utiliser directement `$((`*expression*`))`

2.27 Scripts : boucles

`while` *condition* `; do` *commandes* `; done`
— Ex : `while true ; do date ; sleep 1; done`

`for` *v* `in` *liste* `; do` *commandes* `; done`
— entre do et done, v est une variable
— Ex : `for f in *wav; do lame $f ‘basename $f wav‘mp3 ; done`
— `seq` *a b* : tous les entiers entre a et b.

Voir également : `break`, `continue`, `until`, `case`

2.28 Scripts : fichiers scripts

```
#!/bin/bash
for i in "$*" ; do
  echo $i
done
```

— #! : shebang : dit au système quel interpréteur utiliser note : ce n'est pas nécessairement un shell. Cela peut être un autre interpréteur de scripts, comme `perl`, `awk`, `python` ...
— $1 : 1er argument, $2 : 2eme argument ...
— $* : tous les arguments
— $# : nombre d'arguments

Les scirpts shells, ce n'est pas trop le but du cours. Mais néanmoins, c'est un outil très puissant. On peut faire rapidement et facilement beaucoup d'opérations complexes. Exercez vous, regardez la page manuel du shell de `bash`, ou de tout autre shell que vous voulez utiliser.

3 Programmation système en C : entrées sorties

3.1 Contexte

- À partir de maintenant, on fait du C
- But de ce "chapitre" :
  - se familiariser avec les appels système
  - se familiariser avec les descripteurs de fichiers

3.2 Les appels système

- Un appel système : le processus appelle directement une fonction du noyau.
- En interne : cela se fait par un mécanisme spécial (interruption) On verra plus précisément le mécanisme dans la prochaine section.
- En pratique, ce sont des fonctions que l'on appelle
  (comme à l'accoutumé en C)
- (On verra rapidement, dans la section "Mémoire", comment se passe un appel système.)
- Attention : un appel système est plutôt lent !
- Pour voir les appels système d'un processus :
  `strace` ou `ltrace -S`

3.3 Codes retour et erreurs

- Les appels système renvoient un code retour
- Il faut toujours vérifier si cela a marché !
- les codes erreurs sont généralement retournés dans la variable externe `errno`
- `perror` permet d'afficher de manière compréhensive une erreur système.
- regardez la page du manuel des la fonction que vous utilisez pour comprendre le code retour !

3.4 Quelques rappels en C

```
#include <stdio.h> /* pour printf() */
int main(int argc, char **argv)
{
  /* argc     : nombre d'argument dans la ligne de
     commande
```

```
                (y compris l'executable)
     argv[i] : le ieme argument  */

  int i;
  printf("le nombre d'argument est %d\n",argc);
       /* affiche sur la sortie standard */
  for(i=0;i<argc;i++)
    printf("l'argument %d est %s\n",i,argv[i]);

  return 0; /* code de retour */
}
```

En fait, `main` peut accepter un 3ème argument, qui sera l'ensemble des variables d'environnement

```
int main(int argc, char **argv, char **env)
  int i;
  for(i=0;env[i]!=NULL;i++)
    printf("%d) %s\n",i,env[i]);
  return 0;
}
```

Il y a d'autres façons de voir les variables d'environnement

- `extern char ** environ;`
- `setenv`, `getenv`...

3.5

`printf` est une fonction de la bibliothèque standard du C (libc/glibc), une couche entre l'OS et nos programmes. Il ne s'agit pas d'un appel système, mais elle va elle-même faire un appel système pour afficher le résultat via la sortie standard.

- Il y a deux niveaux de gestion des E/S et fichiers : bibliothèque standard ou par appel système.

Exemple pour ouvrir un fichier : `fopen`) ou via les fonctions du système (pour ouvrir un fichier : `open`.

- `fprintf` utilise un appel système (`write`) pour afficher la chaîne de caractères

Ce sera le cas dans d'autres domaines que les E/S sur les fichiers (pour la mémoire, par exemple) Pour les fichiers, cela peut prêter à confusion, car les fonctions semblent faire la même chose.

Pourquoi cette sur-couche ? La bibliothèque standard offre des moyens indépendant du système (par exemple, C doit pouvoir compiler sur des système non POSIX).

Les fonctions de la bibliothèque standard sont aussi généralement plus commodes. `printf` n'a pas d'équivalent. La gestion mémoire est aussi plus facile via la bibliothèque standard (`malloc`/`free`). Mais elles sont moins proches du système, moins "fines".

3.6 écriture dans un fichier en C via la bibliothèque standard

```
#include <stdio.h>
#include <assert.h>

int main() {
  FILE *out=fopen("hello.txt","w");
  assert(out!=NULL);
  fprintf(out,"hello ");
  /* ou */
  fwrite("world\n",1,6,out);
  fclose(out);
  return 0;
}
```

3.7 écriture dans un fichier en C via des fonctions système

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <assert.h>

int main() {
  int out=open("hello.txt",O_WRONLY|O_CREAT,0644);
  assert(out!=-1);
  write(out,"hello world\n",12);
  close(out);
  return 0;
}
```

3.8 Ouvrir un fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags)
```

- ouvre le fichier au chemin `pathname`
- renvoie un entier, le descripteur de fichier
- renvoie le plus petit descripteur de fichier libre
- renvoie -1 si l'ouverture échoue (fichier non trouvé, pb de droits...)
- les autres fonction d'accès prennent en paramètre ce descripteur de fichier.
- Pour fermer un descripteur : `close(descripteur)`.
- Toujours fermer quand on s'en sert plus !
- `flags` : conjonction de :

  - O_RDONLY, O_WRONLY, ou O_RDWR (lecture, écriture ou les 2)
  - O_CREAT : crée le fichier (s'il n'existe pas)
  - O_APPEND : rajoute à la fin du fichier (positionne à la fin du fichier)
  - O_TRUNC : tronque le fichier à la taille 0.
- `open` peut prendre un 3eme argument : le mode (droits "rwx" pour "ugo", en octal) si un fichier est crée

Exemple : `int fd=open("file.txt",O_WROLNY | O_CREAT | O_TRUNC, 0644);`

3.9 Entrées/sorties standards

Un processus possède à l'origine 3 descripteurs de fichiers ouverts :
- 0 : ouvert en lecture : l'entrée standard
- 1 : ouvert en écriture : la sortie standard
- 2 : ouvert en écriture : la sortie erreur

3.10 Lire dans un fichier

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- `fd` : descripteur de fichier
- `buf` : pointeur vers la zone mémoire où seront copiées les données
- `count` : nombre maximum d'octets à lire
- retour : nombre d'octets lus, -1 si erreur

Similairement : `write` pour écrire

3.11 Se déplacer dans un fichier

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- `offset` : de combien d'octets on se déplace (positif ou négatif) depuis :
  - (si `whence=SEEK_SET`) le début du fichier
  - (si `whence=SEEK_CUR`) la position courante
  - (si `whence=SEEK_END`) la fin du fichier
- retour : position dans le fichier

(Pour connaître la taille d'un fichier `lseek(fd,0,SEEK_END)`)

3.12 Dupliquer les descripteurs

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup` duplique le descripteur `oldfd`, et renvoie un nouveau descripteur
- `dup2` copie le descripteur `oldfd` dans `newfd`

Exemple :

```
int fd=open("sortie.txt",O_CREAT|O_WRONLY,0644);
dup2(fd,1);
```

3.13 Autres fonctions pour gérer les fichiers

- pour tronquer un fichier à la position courante : `truncate`, `ftruncate`
- pour créer un fichier : `open` ou `create`
- pour supprimer un fichier : `unlink`
- pour créer un lien dur : `link`
- pour renommer un fichier : `rename`

3.14 Scanner les répertories

```
#include <sys/types.h>
#include <dirent.h>

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

struct dirent {
  ino_t          d_ino; /* inode number */
  off_t          d_off; /* see man */
  unsigned short d_reclen; /* length */
  unsigned char  d_type; /* type of file */
  char           d_name[256]; /* filename */
};
```

3.15 Informations sur un fichier

```
int stat(const char *pathname, struct stat *buf);

struct stat {
   dev_t st_dev; /* device containing file */
   ino_t st_ino; /* inode number */
   mode_t st_mode; /* protection */
   nlink_t st_nlink; /* number of hard links */
```

```
    uid_t st_uid;  /* user ID of owner */
    gid_t st_gid;  /* group ID of owner */
    dev_t st_rdev; /* device ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize;
      /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;
      /* number of 512B blocks allocated */
    struct timespec st_atim; /* last access */
    struct timespec st_mtim; /* last modification */
    struct timespec st_ctim; /* last status change */
};
```

3.16 Autres

Autres appels système qui peuvent servir (et ne sont pas le sujet de prochains cours)

- `time` : renvoie le temps Unix, i.e. le nombre de secondes depuis le 1er Janvier 1970.
- `exit` : termine le programme
- `nanosleep` : endort le processus pour un temps déterminé (void aussi `sleep` et `usleep`)

Pour trouver l'appel système si on a la commande shell : regarder la fin du man. (Notamment : gestion des droits et fichiers spéciaux...)

4 La mémoire

4.1 Les différentes mémoires vives

Une machine possède différent type de mémoire vive :

- La "mémoire principale" (RAM). Taille de l'ordre de Giga-octet (ordinateur/smartphone actuel) au Tera-octet (grosses machines de calcul).
- Les registres. Il s'agit de mémoires directement implantées dans l'unité de calcul du processeur. Il s'agit de la mémoire la plus rapide disponible, mais très limitée *quelques mots 64 bits*
  On les utilise couramment quand on programme en assembleur. On s'en occupe rarement dans un langage de haut niveau : les compilateurs savent optimiser assez bien l'utilisation des registres. Le mot clef `register` en C est une relique du temps où les compilateurs n'étaient pas aussi performants...
- Les mémoires caches. Mémoire temporaire, plus rapide que la mémoire principale, généralement implantée dans le processeur. Il y en a plusieurs niveaux (L1i, L1d, L2, L3), de taille et de rapidité décroissante. Leur utilisation est gérée au niveau du matériel, et est transparente pour les processus. Pour voir la taille des caches : `lscpu`. On n'en parlera pas plus ici.
- pour voir la taille des caches : `lscpu`
- Le "swap".

Il ne s'agit pas vraiment d'une mémoire vive. C'est une zone sur le disque dur où l'OS peut mettre des parties de mémoire allouées quand il n'a plus de "mémoire principale" disponible. Ceci est transparent pour le processus, mais quand l'OS doit utiliser le swap, les performances chutent....

4.2 Mémoire principale

— La mémoire principale est un tableau d'octets (=8 bits).
— Une adresse mémoire est un index (un "numéro") de case mémoire.
— Un pointeur : une variable qui contient une adresse mémoire.

Sur une machine 64 bits :
— Une adresse mémoire est un entier de 64 bits
— Théoriquement, 2^{64} octets accessibles = 17179869184 Go...

4.3 Adressage sans abstraction

Dans les "vieux" ordinateurs (-286, DOS) (et dans certains modes des ordinateurs actuels) :
— Si processus accède à la donnée à l'adresse i, il accède à la donnée à l'adresse i dans la RAM :
Le processus "voit" directement la mémoire physique.

Deux processus ne peuvent pas utiliser la même zone mémoire, sans interférer.

Problèmes :
— Un processus voit la mémoire des autres processus (problèmes de sécurité et de droits)
— les processus peuvent empiéter les uns sur les autres.
— La mémoire d'un processus doit correspondre à une zone mémoire physique (ex : utilisation de "swap" impossible)

4.4 Virtualisation de la mémoire

Mémoire virtuelle : il n'y a pas une correspondance directe entre l'espace d'adressage d'un processus et la mémoire physique.
— La mémoire vue par un processus est formée d'un ensemble de pages mémoires.
Pour voir la taille des pages mémoires : `getconf PAGESIZE`
— La RAM est découpée en zones de même taille.
— Une translation (au niveau du processeur) a lieu pour convertir les adresses virtuelles en adresse physique, via la table des pages (Pour voir comment cela se passe sur x86 : registre "CR3".)

Avantages :
— Le processus peut organiser la mémoire comme il le veut (chaque processus a sa table)
— Des zones mémoires peuvent être partagées entre différents processus

- Déplacement possible de zones mémoires (swap...)
  - une interruption a lieu si le processus veut accéder à une zone qui ne correspond à rien dans la table des pages.

4.5 Le mode noyau et mode utilisateur

Sous Unix, il y a deux modes de fonctionnement :
- Le mode "utilisateur" : mémoire virtualisée, accès matériel impossible (autrement que via les syscalls).
  Tous vos processus seront dans ce mode.
- Le mode "noyau"
  - Le noyau voit (et peut gérer) toute la mémoire physique. Il peut modifier les tables des pages.
  - + de privilèges (accès au matériel...)

4.6 Appel système

- Un appel système : passage du mode utilisateur au mode noyau
- Via une sorte d'interruption : le processus fait basculer le processeur du mode utilisateur en mode système.
- Chaque syscall a un numéro.
- On ne peut donc pas appeler n'importe quelle fonction du noyau, seulement celles qui ont été prévues...

4.7 Différentes zones mémoire d'un processus :

- les instructions :
  - le code du programme (en langage machine)
  - les librairies qu'il utilise (libc...)
- les données :
  - segment de donnée statique
  - pile (stack)
  - tas (heap)
- non allouées : si on essaye d'y lire ou d'y écrire, il y aura une erreur de segmentation (ou segfault)

- les zones ont également des droits (lecture seule, exécution autorisée...)
- certaines zones peuvent être partagées entre différents processus (c'est un moyen de communiquer inter-processus).

4.8 Pile (Call stack)

- Sont stockés dans la pile : les variables locales aux fonctions, les paramètres des fonctions, les adresses de retour.
- Attention au dépassement !
  (On peut augmenter la taille de la pile avec `setrlimit`)

4.9 Tas (Heap)

Pour les allocation dynamiques.

En C, le tas est géré traditionnellement par la libc via `malloc`/`free`. La structure en interne peut être vue comme une liste chaînée.

Désavantages des malloc/free :
— (des)allocation un peu lent
— une structure allouée prend un peu plus de place en mémoire
— fragmentation
— Il ne faut pas oublier à libérer la mémoire qui ne sert plus (`free`) sinon on aura des fuites mémoires !

On peut changer la taille du tas avec `brk` ou `sbrk`. Néanmoins, comme cette zone mémoire est normalement gérée par la STD, on ne devrait pas l'utiliser.

Mais `sbrk(0)` permet de connaître l'adresse de fin du tas.

4.10 Gérer différemment la mémoire dynamique

Il existe des mécanismes de ramasse miettes (garbage collector), pour éviter d'avoir à désallouer la mémoire. C'est le cas en Caml.

On peut faire ses propres allocateur de mémoire

Exemple : "memory pool", si on alloue beaucoup d'objets de la même taille t
— on alloue un grand tableau de n cases de taille t
— une "allocation" renvoie l'adresse d'une nouvelle case
— les zones libres sont gérées par une liste chainée.

4.11 Exemple : variables dans différentes zones

```
#include <stdio.h>
#include <stdlib.h>
int a ; /* dans le segment statique donn'ees*/
int main() { /* dans le segment statique instructions */
  int b; /*dans la pile */
  int *ptr_c=(int*)malloc(sizeof(int)); /* l'entier point
      'e par ptr_c est dans le tas */
  printf("statique(data):%p statique(instruction):%p pile
     :%p tas:%p\n",&a,main,&b,ptr_c);
  free(ptr_c);
  return 0;
}
```

```
statique(data):0x6009fc statique(instruction):0x400576
   pile:0x7ffd21051b94 tas:0x1310010
```

4.12 Demander des nouvelles zones mémoires

`mmap` permet de mapper de nouvelles zones mémoires
- On peut mapper soit un fichier (via un descripteur), soit une zone vierge
- Deux modes possible : "shared" ou "private"
  - private : on a notre propre copie en mémoire
  - shared : la copie est partagée
- On doit spécifier les droits (read, write, exec)
- On peut spécifier l'adresse.

On peut libérer une zone avec `munmap`.

```
#include <sys/mman.h>

void *mmap(void *addr, size_t length, int prot, int flags
    , int fd, off_t offset);
```

Demande de mapper `length` octets depuis le fichier `fd`, à partir de l'offset (position) `offset`.

`prot` est la protection choisie : un sous ensemble de PROT_READ, PROT_WRITE, PROT_EXEC.

`flags` donne les options. Il doit contenir MAP_SHARED ou (exclusif) MAP_PRIVATE, plus éventuellement d'autres options :
- MAP_SHARED : la modification de la zone mémoire se répercutera sur les autres copies (c'est à dire, le fichier et les autres mappages du fichier). C'est un moyen de communication inter-processus.
- MAP_PRIVATE : on aura notre propre copie de l'objet en mémoire. (C'est un moyen de demander une allocation de mémoire au système, surtout avec l'option MAP_ANONYMOUS).
- Si addr est NULL, le système choisi l'emplacement (conseillé en général), Sinon, il essaye de trouver un emplacement proche
- MAP_FIXED : l'adresse est impérative (soit le mappage se fait à `addr`, soit la fonction échoue)
- MAP_ANONYMOUS : on ne mappe pas un fichier (`fd` est ignoré), mais une zone vierge (initialisée à 0).

4.13 Format et chargement des binaires

Le format des exécutables sous la plupart des Unix est ELF (Executable and Linkable Format)
- Un fichier ELF est composé de plusieurs sections, qui vont correspondre à des zones mémoires ("text" pour les instructions, "data"...)
- Pour voir les différentes sections : `objdump`
- Ces sections seront "chargées" en mémoire via mmap.
- Les librairies (glibc...) seront chargées à l'exécution, par la librairie "ld".

— "ld" cherche les librairies dans les répertoires listés dans LD_LIBRARY_PATH, /lib et /usr/lib
— Il est possible de forcer "ld" à choisir une autre librairie (LD_PRELOAD)

5 Processus

5.1 Les processus : Rappels (?)

Un processus :
— a un numéro (le PID = Process IDentifier)
— a un père.
— a un propriétaire (réel et effectif)
— a un état : en exécution, en sommeil, stoppé ou zombie
— a un niveau de priorité
— un répertoire courant
— (certains processus) peuvent être “légers” (“threads”)
(on y reviendra dans quelques cours)
— une table de descripteurs de fichiers
— une table de pages
— renvoie un code retour (un entier entre 0 et 255)

5.2 Voir les processus

— Pour voir tous les processus tournant, avec leur lien de parenté : `ps faux`
— Pour voir en temps réel (utilisation CPU, mémoire...) : `top`
— Toutes les informations dans le système procfs (`/proc/`)

5.3 État d’un processus

Un processus peut être :
— actif (i.e. en exécution)
— prêt (en attente d’exécution)
— suspendu (par exemple avec ctrl+z)
— en sommeil : il attend un évènement
— sleep, pause...
— attente sur une I/O
— zombi

5.4 Ordonnancement

Deux processus ne peuvent pas s’exécuter en même temps sur un même coeur.

L’OS découpe le temps en petits bouts, et fait tourner les processus les uns après les autres.

L'OS choisi l'ordre, en essayant de respecter la priorité des processus (voir `nice`)

Passage d'un processus à un autre : commutation de contexte (context switch).

— assez lent...
— transparent pour nous

Note : On peut changer la priorité des processus avec `nice` ou `renice`

5.5 Gestion de processus : syscalls

— `getpid()` renvoie le PID du processus courant
— `getppid()` renvoie le PID du père
— `getuid()` renvoie l'UID de l'utilisateur processus courant
— `geteuid()` renvoie l'UID de l'utilisateur effectif du processus courant
— Ces UIDs peuvent être différents si le binaire est en "setuid"
— `setuid()` et `seteuid()` permettent de changer les utilisateurs, si on a les droits !

5.6

— Obtenir/changer le répertoire courant : `getcwd()` / `chdir()`
— Obtenir le temps CPU consommé (en mode utilisateur et système) : `times()`
— Modifier le masque de création de fichiers : `umask()`
— Pour voir/changer les "limites" d'un processus : `ulimit`, `getrlimit`, `setrlimit`

5.7 Priorité d'un processus

— Chaque processus a une priorité :
  — généralement, un nombre entre -20 et 19, et par défaut : 0.
  — plus le nombre est élevé, moins le processus aura du temps de calcul.
— Il est possible de lancer un processus avec une priorité plus basse avec `nice`
— Il est possible de diminuer la priorité d'un de ses processus en exécution :
  — Commande : `renice`
  — Appel système : `nice`
— Seul `root` a le droit d'augmenter une priorité.

5.8 Comment lancer un processus ?

Il faut différencier le fait de :

— créer un nouveau processus : le processus appelant continue de vivre, et un nouveau processus naît

— exécuter un binaire spécifié : il n'y a pas de nouveau processus, l'ancien processus est "écrasé" par le nouveau

Créer un nouveau processus (création) se fait avec `fork`

Exécuter un binaire (recouvrement) se fait avec `exec...`

Créer un nouveau processus qui est l'exécution d'un binaire se fait avec la combinaison de `fork` et `exec...`.

5.9 `int system(const char *command)`

(Pas un appel système. Donné à titre informatif.)

Un moyen simple de lancer un processus depuis un programme est d'utiliser `system` (dans <stdlib.h>).

`system` lance un shell (`/bin/sh`) qui exécutera `command`. Une fois la commande terminée, la fonction retournera le code retour de la commande.

Exemple : `system("ls");`

5.10 Les `exec*`

```
#include <unistd.h>
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[]);
int execvpe(const char *file, char *const argv[], char *const
    envp[]);
```

— Elles ne créent pas un nouveau processus : elles remplacent (recouvrent) le processus courant par l'exécution du fichier en argument.
— En particulier, le PID, L'UID, les fichiers ouverts sont conservés (sauf si option O_CLOEXEC)
— Si l'exécution se fait normalement, elles ne retournent jamais!
— execl* : les arguments sont donnés les un après les autres en argument de la fonction
— execv* : les arguments sont donnés dans un tableau
— execlp/execvp/execvpe : le fichier est cherché dans le PATH
— execlpe execvpe : on spécifie en plus le nouvel environnement

Note : `exec existe aussi dans le shell...`

5.11 `pid_t fork(void)`

`fork()` (dans <unistd.h>) est la commande pour lancer un nouveau processus.

Elle duplique le processus courant, pour créer un processus fils.
- Dans le père, elle renvoie le PID du fils (et rien ne change)
- Dans le fils (le nouveau processus) :
  - elle retourne 0
  - le fils aura un nouveau PID
  - son père sera le processus père
- `fork` retourne donc deux fois, une fois dans le père, une fois dans le fils
- Tout se passe comme si toute la mémoire du processus appelant fork est copiée.
- (En pratique, le système copie seulement si c'est nécessaire.)
- Les deux processus sont concurrents. On ne peut pas dire lequel des deux retournera en premier.

`vfork` est une version allégée de `fork`, mais qui n'est plus trop d'actualité.

Sous Linux, `clone` est similaire à `fork`, mais les deux processus gardent le même espace mémoire. On peut y voir un moyen de faire des processus plus légers (sans recopie de la mémoire).

On verra par la suite un moyen de faire des processus légers proprement avec `pthread`.

5.12 Exemple : fork

```
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    /*...*/
    return 0; /* fin du fils */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

5.13 fork et descripteurs de fichiers

- Lors d'un fork, la table des descripteurs du processus est copiée.
- Les descripteurs des deux processus (père et fils) référencient les mêmes fichiers ouverts par le système (comme après un `dup`)

— En particulier, si un des deux processus modifie le curseur d'un descripteur (read/write/lseek...), cela affectera le curseur du même descripteur de l'autre processus

```
int main()
{
  int fd=open("sortie.txt",O_CREAT | O_RDWR,0644);
  if(fork()==0) {
    write(fd,"A",1);
    close(fd);
    return 0;
  }
  write(fd,"B",1);
  close(fd);
  return 0;
}
```

sortie.txt : `AB` ou `BA`

5.14 Exemple : fork + exec

```
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'execlp a
        echoue */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

5.15 `pid_t wait(int *ptr)`

`wait` : attend jusqu'à ce qu'un processus fils termine.

Plus précisément :
— Si un processus fils termine avant l'appel de `wait` de son père, il devient zombi.
— Si un processus n'a pas de fils : `wait` renvoie -1.
— Si un processus a un fils zombi : `wait` renvoie le PID du fils zombi, et efface ce processus de la liste des processus.

— Si `ptr` n'est pas NULL, `wait` copie le "statut" dans l'entier pointé par `ptr` (voir man). Le status permet (entre autres) de retrouver le code de retour du processus.

— Si un processus a des fils, mais pas pas de fils zombi : `wait` attend jusqu'à ce qu'un fils devienne zombi, puis idem.

Attente d'un processus particulier :

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

5.16 Exemple : fork + exec + wait

```
#include <unistd.h>

int main()
{
  int status;
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'execlp a
        echoue */
  }
  /* si on est ici, on est le pere */
  wait(&status);
  return 0; /* fin du pere */
}
```

5.17 `pid_t setsid(void)`

Un processus peut se détacher de son père en appelant `setsid()`.

Plus précisément, cette fonction sert à créer une nouvelle session. Le processus appelant devient leader de cette session.

5.18 Exemple : `nohup &` (un premier essai...)

```
#include <unistd.h>

int main(int argc, char *argv[])
{
  argc -= 1;
  argv += 1;

  if(fork()) {
    /* si on est ici, on est le pere */
    return 0;
```

```
  }
  /* si on est ici, on est le fils */
  setsid();
  execvp(*argv, argv);
  return 1; /* si on se trouve ici, c'est qu'execvp a
      echoue */
}
```

5.19 Communication inter processus : avant goût

IPC = Inter Processus Communucation

Comment faire communiquer des processus ?
— via les entrées sorties : pas dynamique
— fichier standards : archaïque, lent, problèmes de synchronisation
— fichiers tubes
— Signaux : information très limitée (mais ça sert à plein de niveau)
— partage de mémoire
— par un canal réseau ...

6 [C : Pointeurs sur fonctions]

6.1

— Une fonction dispose également d'une adresse mémoire
— C'est son point d'entrée , i.e. l'adresse mémoire où commence la liste des instructions en langage machine
— Il est possible de manipuler les adresses des fonctions en C, et d'avoir des pointeurs sur des fonctions
— Il est obligatoire de savoir manipuler les pointeurs sur fonctions pour gérer les signaux et les threads...

6.2 Syntaxe en C

Le type d'un pointeur sur fonction doit contenir les types des paramètres de la fonction, et le type de retour.
— les paramètres n'ont pas besoin d'avoir de nom :
— le compilateur doit juste savoir quel type empiler sur la pile

Pour déclarer un pointeur sur une fonction :
`type_retour (*nompointeur) (liste_arguments...);`
Exemple :

```
int (*fct) (int);
```

déclare fct comme étant un pointeur sur une fonction prenant en argument un entier, et revoyant un entier

Appeler une fonction pointée se fait de la même manière que pour une fonction normale.

6.3 Exemple

```
int carre(int x) {return x*x;}

int cube(int x) {return x*x*x;}

void iter(int (*fct)(int)) {
  int i;
  for(i=1;i<=10;i++)
    printf("%d : %d\n",i,fct(i));
}

void main() {
  int (*x)(int);
  x=carre;
  iter(x);
  iter(cube);
}
```

6.4 avec `typedef`

On peut simplifier les choses avec `typedef` :

— `typedef int (*typefctintint) (int);`

Définit `typefctintint` comme étant le type pointeur sur une fonction int → int ;

— `typefctintint fct=carre;`

6.5 Exemple 1 : atexit

`atexit` enregistre une fonction qui sera appelée à la fin (normale) du processus (après un `exit`, ou au retour du `main`)

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Exemple :

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

void fin(void)
{
  printf("Je suis toujours l'a !\n");
}

int main()
```

```
{
  atexit(fin);
  printf("Je me termine\n");
  return 0;
}
```

6.6 Exemple 2 : qsort

`qsort` est une fonction de la libc effectuant un *quick sort*.

```
#include <stdlib.h>
void qsort(
  void *base,
  size_t nmemb,
  size_t size,
  int (*compar)(const void *, const void *)
  );
```

On doit passer en argument l'adresse de la fonction de comparaison (`compar`) que `qsort` doit utiliser.

Exemple (trie les lignes de l'entrée standard) :

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

int cmp(const void *a, const void *b)
{
  return strcmp(*(const char**)a,*(const char **)b);
}

int main()
{
   char bf[1000];
   char *tab[1000];
   int k=0,i;
   while(fgets(bf,1000,stdin))
     tab[k++]=strdup(bf);
   qsort(tab,k,sizeof(tab[0]),cmp);
   for(i=0;i<k;i++)
     printf("%s",tab[i]);
   return 0;
}
```

7 Les signaux

7.1 Les signaux : introduction

— Les signaux permettent de notifier des évènements à un processus.
— Il s'agit d'un moyen de communication limité :
  — ponctuel
  — unique information : le numéro du signal, un entier entre 1 et (généralement) 64.
— Mais très important sous Unix.

Beaucoup de mécanismes sous Unix utilisent des signaux. Par exemple :
— `ctrl + c` (arrêt d'une tache)
— `ctrl + z` (mise en pause d'une tache)
— Tuer un processus par `kill`
— Erreur de segmentation
— Division par 0...

7.2 Signaux standard

— SIGTERM (15) : Signal de fin (signal par défaut de `kill`)
— SIGINT (2) : Terminaison depuis le clavier (ctrl + c)
— SIGKILL (9) : Tuer un processus (on ne peut pas le contourner)
— SIGSTOP (19) : Arrêt (pause) du processus (ctrl + z)
— SIGCONT (18) : Continuer si en pause
— SIGALRM (14) : Temporisation `alarm` écoulée.
— SIGUSR1 (10) : Signal utilisateur 1.
— SIGUSR2 (12) : Signal utilisateur 2.
— SIGCHLD (17) : Fils arrêté ou terminé

Les erreurs :
— SIGFPE (8) : Erreur mathématique virgule flottante.
— SIGPIPE (13) : Écriture dans un tube sans lecteur.
— SIGSEGV (11) : Référence mémoire invalide.
— SIGILL (4) : Instruction illégale.
— ...

7.3 Signaux générés

Un signal est généré par un évènement :
— Envoi d'un signal par un autre processus
— Action sur le terminal (ctrl+c, ctrl+z...)
— Erreur (arithmétique, de segmentation ...)
— Minuterie
— Arrêt ou terminaison d'un fils...

Lorsque le signal est délivré à un processus, une action se produit :
— action par défaut
— signal ignoré

— effectuer une action choisie : handler
Un signal généré, mais pas (encore) délivré, est pendant

Si le même signal est généré plusieurs fois, on est pas sûr qu'il sera délivré le même nombre de fois

7.4 Envoyer un signal

Depuis le shell : `kill -sig pid`, où :
— *sig* est le signal : le nom (KILL, STOP,CONT...) ou numérique
— *pid* est le PID du processus à qui on lance le signal

Appels systèmes :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
int raise(int sig);
unsigned int alarm(unsigned int s)
```

`kill` envoie le signal `sig` au processus `pid`.
`raise` envoie le signal `sig` au processus courant.
`alarm` envoie le signal SIGALRM s secondes plus tard. (Si $s = 0$, annule l'alarme)

7.5 Réception : comportement par défaut

À la réception d'un signal, un comportement par défaut est définit. Celui-ci peut être :
— Terminer le processus
  — KILL, TERM, ALARM, INT, FPE, PIPE, USR1, USR2...
— Terminer le processus avec fichier `core`
  — ILL, SEGV
— Signal ignoré
  — CHLD
— Suspension du processus
  — STOP
— Continuation du processus
  — CONT

Il est possible d'ignorer ce comportement par défaut, ou d'en définir un autre, pour tous les signaux, sauf SIGSTOP et SIGKILL

7.6 Ensemble de signaux

`sigset_t` est un type pour un ensemble de signaux. Une variable de ce type peut être manipulé par les fonctions suivantes.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

7.7 Masquer des signaux

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set, sigset_t *
    oldset);
```

- `how` :
  - SIG_SETMASK : nouveau masque = set
  - SIG_BLOCK : nouveau masque = ancien masque ∪ set
  - SIG_UNBLOCK : nouveau masque = ancien masque \ set
- Masquer un signal ne veut pas dire l'ignorer.
- Si un signal masqué est généré, il reste pendant, sauf si le comportement par défaut est de l'ignorer.

7.8 Lister les signaux pendants

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Copie dans `set` la liste des signaux pendants.

C'est particulièrement utile si des signaux sont masqués (et non ignorés).

7.9 Changer le comportement : `signal`

On peut demander à exécuter une fonction (handler) en cas de réception d'un signal.

L'ancienne interface Unix (non POSIX) est la suivante. Donnée à titre indicatif (car plus simple à comprendre). Ne pas utiliser.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
```

`handler` est soit :
- SIG_IGN : ignore le signal
- SIG_DFL : action par défaut
- l'adresse d'une fonction prenant un entier
  - à la réception d'un signal, la fonction sera appelée, avec comme argument le numéro du signal.

```
void handler(int i)
{
    printf("signal recu : %d\n",i);
}

int main()
{
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    sleep(10000);
    return 0;
}
```

7.10 sigaction

L'interface à utiliser pour manipuler les handlers est `sigaction`

```
struct sigaction {
 void      (*sa_handler)(int);
 void      (*sa_sigaction)(int, siginfo_t *, void *);
 sigset_t    sa_mask;
 int         sa_flags;
};

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

- sa_handler le handler (comme `signal`)
- sa_flags : options (void man)
- sa_mask : liste des signaux à masquer en plus, le temps que le handler s'exécute
- Si `act` n'est pas NULL : nouveau handler à installer
- Si `oldact` n'est pas NULL : l'ancien handler est copié dans la structure pointée
- On peut utiliser `sa_sigaction` à la place de `sa_handler` pour avoir un comportement plus fin (voir man).

7.11 Attente de signaux

```
#include <unistd.h>
int pause(void);

#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

— `pause` met le processus en pause, jusqu'à ce qu'un signal (n'importe lequel) arrive.
— Problème : un signal non masqué peut arriver avant l'appel à pause(), et être "perdu"...
— `sigsuspend` change temporairement le masque des signaux masqués, et attend jusqu'à ce qu'un signal arrive.

7.12 Signaux et appels systèmes

— Certains appels systèmes peuvent être interrompus par un signal.
  — Dans ce cas, l'appel système échoue, et le code retour (errno) sera EINTR
— Lors d'un fork, les signaux pendants ne sont pas hérités (le masque et les handlers, si)
— Lors d'un exec, les handlers ne sont pas hérités (le masque et les signaux pendants, si)

8 Communication Inter Processus (IPC) : Tubes

8.1 Principe

— À partir de maintenant, on veut faire communiquer plusieurs processus.
— Un tube est un moyen de le faire.

Note :

— Faire communiquer des processus sur une même machine par tubes peut sembler archaïque, et pas très efficace (comparé à la mémoire partagée).
— Mais : les communications réseau (par sockets) se feront de manière similaire
— les principes/fonctions expliqués dans ce chapitre seront toujours valables.
— Tube : canal de communication FIFO (First In First Out)
— Utilise 2 descripteurs de fichiers : un pour l'écriture (l'entrée), et un pour la lecture (la sortie)
— L'écriture dans l'entrée sera mise en attente dans un tampon
— La lecture dans la sortie lira les données du tampon, dans l'ordre (FIFO).
— La lecture et l'écriture se font comme pour les fichiers réguliers : read et write
— Il n'y a pas de curseur : lseek est impossible !

Par exemple, lorsque l'on exécute :

```
cat fichier.txt | grep password
```

— le shell lance deux nouveaux processus : un pour `cat` et un pour `grep`
— le shell crée un tube
— la sortie standard de `cat` sera le côté "écriture" du tube
— l'entrée standard de `grep` sera le côté "lecture" du tube

Plus précisement :
`cat fichier.txt | grep password`

- le shell crée un tube
- le shell lance deux nouveaux processus (deux `fork()`) : un pour `cat` et un pour `grep`
- la sortie standard de `cat` est ecrasée par le côté "écriture" du tube (via par exemple `dup2`)
- l'entrée standard de `grep` est écrasée par le côté "lecture" du tube
- les fils se recouvent (`exec...`) en cat et grep.

```
cat file.txt | grep passwd | sed 's/.*passwd=\(\w*\).*/\1/'
```

```
$ lsof
...
cat   5223  mrao  0u  CHR  136,1  0t0  4      /dev/pts/1
cat   5223  mrao  1w FIFO   0,10  0t0  27854  pipe
cat   5223  mrao  2u  CHR  136,1  0t0  4      /dev/pts/1
...
grep  5224  mrao  0r FIFO   0,10  0t0  27854  pipe
grep  5224  mrao  1w FIFO   0,10  0t0  27856  pipe
grep  5224  mrao  2u  CHR  136,1  0t0  4      /dev/pts/1
...
sed   5225  mrao  0r FIFO   0,10  0t0  27856  pipe
sed   5225  mrao  1u  CHR  136,1  0t0  4      /dev/pts/1
sed   5225  mrao  2u  CHR  136,1  0t0  4      /dev/pts/1
```

8.2 Créer un tube (par syscall)

```
#include <unistd.h>

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

- Ouvre les 2 descripteurs de fichier associés a un nouveau tube
- Prend en argument un tableau de deux entiers :
- Renvoie dans `pipefd[0]` la sortie du tube (le descripteur en lecture)
- Renvoie dans `pipefd[1]` l'entrée du tube (le descripteur en écriture)

8.3 Exemple : pipe

```
main() {
  int fd[2],r;
  char buffer[10];

  pipe(fd);

  r=write(fd[1],"hello",5);
  assert(r==5);

  r=read(fd[0],buffer,10);
```

```
  assert(r==5);

  buffer[r]=0;
  printf("recu: %s\n",buffer);
}
```

8.4 Exemple : pipe + fork

```
#include <unistd.h>
#include <stdio.h>
#include <string.h>
#include <assert.h>

int main() {
  char buffer[11];
  int fd[2];
  pipe(fd);

  if(fork()==0) {
    close(fd[1]); // on ferme les descripteurs qui ne
        servent plus
    while(1) {
      int r=read(fd[0],buffer,10);
      if(r<0) perror("Erreur lecture");
      else {
        buffer[r]=0;
        printf("fils recoit: '%s'\n",buffer);
      }
    }
    return 0;
  }
  close(fd[0]); // on ferme les descripteurs qui ne
      servent plus

  while(1) {
    snprintf(buffer,11,"%d",rand()%1000);
    printf("pere envoie '%s'\n",buffer);
    int r=write(fd[1],buffer,strlen(buffer));
    if(r<0) perror("Erreur ecriture");
    usleep(1000*(rand()%1000));
  }
  return 0;
}
```

8.5 Créer un tube nommé ("fichier tube")

— Un autre moyen de créer un tube est de créer et ouvrir un "tube nommé"

— Il s'agit d'un fichier spécial (non "régulier")
— Commande shell pour créer un tube nommé : `mkfifo`.
— Appels systèmes : `mkfifo` ou `mknod`.

Quand un fichier tube est ouvert en lecture, et ouvert par un autre processus en ecriture, le comportement sera le même qu'un tube crée par `pipe`

8.6 Mode "flot" (stream)

Mode flot : les envois successifs d'informations s'additionnent.
Il n'y a pas de "séparations" entre elles.

Exemple :
— `write(in,"ABC",3)`
  — le tube contient "ABC"
— `write(in,"123",3)`
  — le tube contient "ABC123"
— `read(out,bf,4)`
  — renvoie 4, et bf contient "ABC1"
  — le tube contient "23"
— `read(out,bf,4)`
  — renvoie 2, et bf contient "23"
  — le tube est vide
— `read(out,bf,4)`
  — bloque jusqu'à ce qu'un processus écrive dans le fifo...

8.7 Nombre de lecteur et écrivains

— Un tube peut avoir un nombre de lecteur (ou d'écrivain) différent de un.
— Si un tube a 0 lecteur : l'écriture échouera (signal SIGPIPE)
— Si plus d'un lecteur : premier arrivé, premier servi
— Si un tube a 0 écrivain (et le tube est vide), la lecture renverra 0 (i.e. comme pour un fin de fichier)

— Comme toujours, on ferme les descripteurs qui ne servent plus.

8.8 Caractère bloquant

— Par défaut, la lecture dans un tube vide sera bloquant
— Il est possible de rendre la lecture non bloquante, en changeant l'option O_NONBLOCK du descripteur de fichier
— Dans ce cas, la lecture dans un tube vide échouera (retour -1), avec errno = EAGAIN

— Attention, un tube a aussi une capacité limitée (PIPE_BUF=4096). Quand un tube est plein, une écriture sera également bloquante.

8.9 Manipuler un descripteur de fichier : fcntl

`fcntl` permet de manipuler les descripteurs de fichiers.

Elle permet (entre autres) de changer les options (modes) des descripteurs de fichiers. En particulier :
— O_NONBLOCK : caractère non bloquant d'un descripteur

Pour passer un descripteur en mode non bloquant :

```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

8.10 Attente sur plusieurs descripteurs de fichiers : select

`select` permet d'attendre (avec un temps limite) sur un ensemble de descripteurs de fichiers en un seul appel.

Pour utiliser `select`, il faut au préalable manipuler une structure qui représente un ensemble de descripteurs de fichiers. Cela se fait via les primitives suivantes :

```
#include <sys/select.h>

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);

#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *writefds,
    fd_set *exceptfds, struct timeval *timeout);
```

— nfds : le plus grand descripteur de fichiers à vérifier +1
— readfds : l'ensemble des descripteurs à vérifier en lecture
— writefds : l'ensemble des descripteurs à vérifier en ecriture
— execptfds : l'ensemble des descripteurs à vérifier en exception
— timeout : temps maximal à attendre.

À sa sortie, `select` modifie les ensembles de telle façon qu'il ne reste que les descripteurs de fichiers sur lesquels il y a quelque chose à lire ou écrire.

8.11 Attente sur plusieurs descripteurs de fichiers : poll

`poll` permet également d'attendre sur un ensemble de descripteurs de fichiers, mais plus finement.

```c
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
 int   fd;         /* file descriptor */
 short events;     /* requested events */
 short revents;  /* returned events */
};
```

- fds : un table de pollfd à surveiller,
- nfds : taille de fds
- timeout : temps maximum (en millisecondes)
- cmdevents et `revents` sont des conjonctions de :
  - POLLIN : il y a quelque chose à lire
  - POLLOUT : il est possible d'y écrire
  - POLLERR : il y a une erreur
  - POLLHUP : pipe ou socket fermé de l'autre côté

8.12 Un premier pas vers les communications réseau

Une socket est un point de communication où il est possible d'envoyer et de recevoir des informations.
On en reparlera longuement au moment de la programmation réseau

Les sockets communiquent par pair. Il y a plusieurs moyen de les faire communiquer (différents protocoles réseau, ou en local).

On peut créer une paire de socket en communication locale, qui fonctionnera similairement deux tubes :

- l'entrée de la 1ere socket sera l'entrée du 1er tube et la sortie de la 2eme socket sera la sortie du 1er tube
- l'entrée de la 2eme socket sera l'entrée du 2eme tube et la sortie de la 1ere socket sera la sortie du 2eme tube

```c
#include <sys/types.h>
#include <sys/socket.h>

int socketpair(int domain, int type, int protocol, int sv
    [2]);
```

Crée 2 sockets associées.
Pour le faire via une communication locale :

- domain = AF_UNIX
- protocol = 0
- type :
  - SOCK_STREAM : communication par flot (comme pour les tubes)
  - SOCK_DGRAM : communication en mode paquet
- sv : un tableau de 2 entiers, pour le renvoi des 2 descripteurs de fichiers (les 2 sockets)

8.13 mode paquet (DGRAM)

Au contraire du mode flot (STREAM), chaque information envoyée constitue une entité indivisible.

Exemple :
- `write(in,"ABC",3)`
  - la file de messages contient "ABC"
- `write(in,"123",3)`
  - la file contient "ABC","123"
- `read(out,bf,10)`
  - renvoie 3, et bf contient "ABC"
  - la file contient "123"
- `read(out,bf,10)`
  - renvoie 3, et bf contient "123"
  - la file vide
- `read(out,bf,4)`
  - bloque jusqu'à ce qu'un processus écrive dans le socket...

8.14 Autres IPC

D'autres moyens de communication inter-processus existent (POSIX et SysV).

Nous n'ont parlerons pas, car les mécanismes sont similaires à des mécanismes déjà vus (pipe/socket), ou que l'on verra plus tard (threads)

Ce sont :
- Les files de messages (POSIX : `man mq_overview`)
  - Similaire aux sockets en mode paquet (DGRAM)
- La mémoire partagée (POSIX : `man shm_overview`)
  - Un segment mémoire est partagé entre plusieurs processus. C'est un moyen de communication très rapide (au sein d'une même machine), mais il faut faire attention aux synchronisations.
- Les sémaphores (POSIX : `man sem_overview`)
  - Il s'agit de mécanisme de synchronisation (exclusion mutuelle). On parlera de sémaphores et mutex en même temps que les threads.

Pour voir les mécanismes System V : `man svipc`

9 Bonnes pratiques, débogage et optimisation

9.1 On va voir :

Quelques bonnes et mauvaises pratiques de programmation

Outils de débogage :
- gdb
- valgrind

Outils de "profilage" :
— gprof
— gcov

Options utiles de `gcc`

9.2 Les "bugs"

Des bugs (cachés ou non) peuvent avoir de conséquences fâcheuses :
— plantages (aléatoires), pertes de données
— "exploitations" : porte d'entrée aux problèmes de sécurités

Lorsqu'un "bug" arrive :
— c'est (généralement) votre faute !

S'il un programme s'exécute sans "bug" :
— cela n'implique pas que vous avez bien programmé !
— les bugs peuvent être "non déterministes" ("Heisenbug"...)

9.3 Les bugs dans un code

Mieux vaut prévenir que guérir :
— adopter de bonnes pratiques de programmation
— tester régulièrement son code
— détecter les problèmes le plus tôt possible dans le processus de programmation

Mais quand il faut guérir :
— utilisation d'outils de débogage

9.4 Bonnes pratiques

Servent à éviter la confusion, et améliorer la compréhension entre les différents programmeurs. Donc en conséquence, à limiter le risque d'erreurs.
— commenter le code
— avoir indentation correcte
— utiliser des noms de variables/fonction explicites
— "garder le code simple" (KIS) :
— préférer des fonctions courtes
— éviter la redondance de code
— lors de la première version préférez un algorithme simple (et plus lent) à un algorithme complexe (et plus rapide)

Pour les projets conséquents, ou à plusieurs :
— code "modulaire"
— documentez vos fonctions
— respectez une convention de nommage
— utilisez un utilitaire de versionnage

Note : on peut faire un code correct sans ces pratiques, mais c'est périlleux (e.g : `ioccc`)

9.5 Pratiques mauvaises/dangereuses/interdites

Ne pas initialiser les variables
— l'erreur pourra passer inaperçue, car souvent elle sera initialisée à 0 la première fois, mais après ce sera plus aléatoire...
Ne pas tester les codes retours
— lire les manuels des fonctions que vous utilisez
L'utilisation de fonctions réputées dangereuses
— sprintf(), strcpy(), strcat(), vsprintf(), gets() ne vérifient pas si il y a assez de place
— fonctions non ré-entrantes dans un code multithread
Ne pas désallouer/fermer ce qui ne sert plus (mémoire, descripteurs de fichiers...)

Note : avec ces pratiques, un code ne sera pas "correct".

9.6 Tester

En cas de projet conséquent, il faut régulièrement :
— tester si le code compile
— tester s'il donne les résultats attendus

Séparer le processus de développement en petites parties. Par exemple, on peut dégager deux processus indépendants :
— le "refactoring" : on ne change pas les fonctionnalités, on ne fait que réorganiser/clarifier/simplifier/optimiser le code.
— l'ajout de fonctionnalités.

Ne pas les faire en même temps, et vérifier après chaque étape.

9.7 Tests

— "Test unitaire" : vérifier le bon fonctionnement d'une partie (unité, module) du logiciel.

— Créez et intégrez une batterie de tests qui teste automatiquement chaque partie les unes après les autres

— Les tests doivent être "méchants" : testez sur beaucoup d'entrées, et essayez de couvrir tous les cas

9.8 Programmation par contrats

Assertion : expression qui doit être évaluée à vraie à un moment donné

Dans le paradigme "Programmation par contrats", 3 types d'assertions :

— pré-conditions
— post-conditions
— invariants

En C/C++ : on peut tester une assertion avec `assert`

9.9 `assert`

`assert(expr)`
— dans assert.h
— se déactive avec l'option -DNDEBUG
— attention : pas pour tester les codes retours dans un vrai programme !

```
#define mon_assert(expr) {\
  if(!(expr)) {\
   fprintf(stderr,"assert␣%s␣fail␣%s:%s:%d\n",\
            __STRING(expr), __FILE__,\
            __ASSERT_FUNCTION,__LINE__);\
   abort();\
  }\
}
```

9.10 Outils d'aide au développement

Utilisation d'environnement de développement

Outils de gestion de version
— subversion (SVN), Git, mercurial...
— possible de faires des branches stable / développement
— il est possible de reprendre une ancienne version pour tracker l'apparition d'un bug.

9.11 Tracker les bugs : outils à disposition

voir les choses "suspectes", même sur un code qui semble marcher correctement :
— gcc -Wall
  — un code devrait toujours compiler sans warning !
  — on peut raffiner les tests de warning. ex : "-Wno-sign-compare"
  — on peut (des)activer un test dans le code :

```
#pragma GCC diagnostic ignored "-Wsign-compare"
...
#pragma GCC diagnostic warning "-Wsign-compare"
```

— gcc -fstack-protector-all
— en C++ : g++ -D_GLIBCXX_DEBUG pour des tests sur les conteneurs de la STL

— Valgrind : passer un coup de valgrind de temps en temps, même sur un code sans suspicion, ne fait pas me mal...

En cas de bug avéré :
— compiler avec les infos de débogage : gcc -g
  — attention, des fois cela fait des choses bizarres avec "-Ox"
— gdb
— valgrind

9.12 Valgrind

valgrind détecte (des fois) :
— les variables non initialisées
— les fuites mémoires
— les dépassements de tableaux

Il y a (rarement) des faux positifs (dans certaines librairies). Mais en général : si il y a un warning, c'est qu'un truc n'est pas bon dans votre code. Càd, un truc à corriger au plus tôt !

Principe (idée) : exécute le code dans un processeur virtuel. Exécution 10 à 30x plus lente...

9.13 Valgrind : utilisation

— Compiler avec l'option -g (rajout des symboles de débogage dans le fichier binaire)
— Exécuter la commande, précédée de `valgrind`
— Les avertissement seront envoyés sur la sortie erreur :

```
==21068== Invalid write of size 8
==21068==    at 0x400A6F: add(char const*, elm_t*) (vector.cpp:28)
==21068==    by 0x400AF7: main (vector.cpp:39)
==21068==  Address 0x5a81c88 is 8 bytes inside a block of size 16 free'd
==21068==    at 0x4C2A30B: operator delete(void*) (vg_replace_malloc.c
    :575)
...
```

9.14 Autres outils de la suite Valgrind

`valgrind -tool=<toolname>`
— memcheck (par défaut) : reporte les problèmes d'accès mémoire (non alloué, non initialisé, inaccessible), les fuites mémoires, double-free...
— massif : profilage de tas
— cachegrind, callgrind : profilage de cache
— helgrind , DRD : déboguer programmes multithreadés

9.15 `gdb`

`gdb` est le déboguer par défaut de la suite GNU
Permet, entre autres :
— d'exécuter jusqu'à un ou des points d'arrêts
— d'exécuter pas à pas
— de regarder l'état des variables, pile, registres...

Comment ça marche (idée, sur x86) :
— gdb a accès à tout l'espace mémoire du processus qu'il débogue
— quand on met un point d'arrêt sur une ligne, gdb remplace la première instruction machine correspondante à la ligne par une instruction "INT 3" (opcode : 0xCC)
— l'exécution de "INT 3" provoque une interruption, qui rend la main à gdb (qui peut remettre l'instruction initiale à la place de INT 3)

9.16 `gdb` : lancement

Compiler le programme à déboguer avec l'option "-g"
— attention, ça fait souvent des choses bizarres avec -Ox
Lancer le gdb :
— `gdb ./executable`
— si arguments : `gdb --args ./executable arguments...`
Dans l'interface de gdb :
— `run` : lance l'exécution

Attacher un programme en cours d'exécution :
— lancer `gdb`
— `attach pid`

— `gdb -tui` : avec une interface textuelle

9.17 `gdb` : lister le code

— `run` : lance l'exécution
— `ctrl+c` : stoppe l'exécution
— `cont` : continue l'exécution
— `list` : lister le code (à la position courante)
— `list` *`fct`* : lister le code depuis le début de la fonction *fct*
— `list` *`fichier:fct`* : lister le code depuis le début de la fonction fct dans le fichier *fichier*
— `list +`, `list -` : avancer (reculer) dans le fichier
— `step` : avance d'un pas
— `next` : avance d'un pas (sans entrer dans les fonctions)
— `finish` : avance jusqu'à la fin de la fonction courante

9.18 `gdb` : points d'arrêts

Point d'arrêt (breakpoint) : arrête le processus quand il atteint une ligne
— `break` *`fct`* : rajoute un point d'arrêt au début de la fonction *fct*
— `break` *`n`* : rajoute un point d'arrêt à la ligne *n*
— `break` *`fichier:ligne`* ou `break fichier:fct`
— possibilités de point d'arrêts conditionnels
— `info breakpoints` : lister les points d'arrêts
Retirer un point d'arrêt :
— `clear` *`fct`*
— `delete` *`nb`*

9.19 `gdb` : variables et "watchpoints"

— `print` *`var`* : affiche la valeur d'une variable (ou expression)
— `display` *`var`* : affiche à chaque pas

Modifier une variable :
— `set` *`var = x`*

watchpoint : arrête le programme quand une variable est modifiée
— `watch` *`var`*

9.20 `gdb` : pile et threads

— `backtrace` : affiche la pile d'appels
— `up` / `down` : monter ou descendre dans les *frames*
— `frame` *`num`* : changer de *frame*

Multithread :
— `info threads` : liste les threads
— `thread` *`num`* : change le thread courant

9.21 `gdb` : registres et assembleur

— `info registers` : affiche les registres
— `layout asm` : affiche le code assembleur
— `layout src` : affiche le code source

Il existe des interfaces graphiques à gdb...

9.22 `gdb` : raccourcis

— entrée : précédente commande
— r : run
— l : list
— c : continue
— s : step

— n : next
— bt : backtrace
— i : info
— b : breakpoints
— i b : info breakpoints
— ...

9.23 Déboguer : aller plus loin

Il est possible d'intégrer des outils de débogage dans son code. Exemple : `backtrace`

```
void sigsegv(int)
{
  void *bt[DEBUGMEM_MAXBT];
  int sizebt = backtrace (bt,DEBUGMEM_MAXBT);
  char **strings = backtrace_symbols (bt, sizebt);
  for(int i=0;i<sizebt;i++)
    fprintf(stderr,"  %s\n",strings[i]);
  exit(1);
}

...
signal(SIGSEGV, (sighandler_t) sigsegv);
signal(SIGBUS, (sighandler_t) sigsegv);
...
```

9.24 Optimiser son code

Là, on suppose que notre code marche bien. On veut l'optimiser :

Options de gcc :
— -Ox
  — -O0 : pas d'optimisation
  — -O1 : optimisations modérées
  — -O2 : pleines optimisations
  — -O3 : comme -O2, en encore plus agressif
  — -Os : optimisation en mémoire (taille de d'exécutable)
— -march=native : compile pour le processeur de la machine
— -ffastmath : active certaines optimisations sur les flottants (ne respecte plus la norme IEEE 754)
— ...

Optimisations de gcc : passer des variables en registres, rendre des fonctions *inline*, dérécursiver, déboucler, réorganisation des instructions...

9.25 À savoir :

— les malloc/free (new/delete), c'est plutôt lent. Préférer d'autres méthodes d'allocations en cas de grosse demande
— les realloc peuvent être très lents (déplacement en mémoire)
— les appels systèmes, c'est très lent

En C++ : Certains conteneurs sont plus lents que d'autres :
— utiliser le conteneur le plus adapté
— array, c'est bcp plus rapide que vector
— remplir un vecteur avec un push_back, c'est lent
Certaines choses rendent l'inlinisation impossible :
— les accesseurs séparés dans un autre fichier source
— les fonctions membres `virtual`...

9.26 Outils de profilage

Profilage : analyse dynamique de l'exécution d'un code.

Outils :
— gprof
— gcov
— C++ : g++ -D_GLIBCXX_PROFILE
`https://gcc.gnu.org/onlinedocs/libstdc++/manual/profile_mode.html`

9.27 `gprof`

— Calcule le temps passé dans chaque fonction, et le graphe d'appel.
— Le compilateur rajoute du code, qui va générer un fichier `gmon.out` contenant les informations de profilage.
— Inconvénient : le code ne doit pas être optimisé (-Ox) , sinon cela peut faire des choses bizarres
  — cela ne dit pas vraiment le temps passé dans chaque fonction quand ce sera optimisé, mais cela donne néanmoins de bonnes approximations
— Note : le code devient notablement plus lent

9.28 `gprof` : utilisation

— Compiler avec l'option -pg
  — attention, souvent cela fait des choses bizarres avec -Ox !
— Lancer le programme normalement. Il va générer le fichier `gmon.out`
— Une fois terminé, lancer `gprof executable`
— L'affichage est en 2 parties
  — Le temps passé dans chaque fonction
  — le graphe d'appel

9.29 `gcov`

— Teste la "couverture". Pour chaque ligne, affiche le nombre de fois que la ligne a été exécutée
— Compiler avec `-fprofile-arcs -ftest-coverage`
— Exécuter le code.
— Exécuter `gcov` *`fichier_source`*
— Il va générer un fichier texte *fichier_source.gcov*

10 Threads POSIX (1) Création et gestion

10.1 Introduction

— Loi de Moore plus trop d'actualité
— La puissance de calcul augmente maintenant (majoritairement) avec la multiplication des coeurs, des processeurs et des machines
— Pour tirer parti des machines multicoeurs : utilisation d'algorithmes parallèles et programmes multithreadés

Il est possible d'implémenter des algorithmes parallèles avec ce qu'on a vu jusque là, mais c'est lourd :
— `fork` : appel système lourd
— chaque processus a son espace mémoire (perte de mémoire)
— chaque processus a ses structures systèmes (table des pages, fichiers ouverts...)
— *context switch* lent
— communication inter-processus généralement lente

Solution : les processus légers ("threads")

10.2 Processus légers (*threads*)

— Plusieurs visions et implémentations possibles
— Introduction dans la norme POSIX en 1995 (POSIX 1.c)

Différence entre un thread et un processus normal :
— un thread est une "partie" d'un processus
— un processus est l'exécution d'un ensemble (>=1) de threads

Différence entre un ensemble de threads et un ensemble de processus :
— les threads partagent pratiquement tout (mémoire, pid, fichiers ouverts...)
— la synchronisation n'est plus gérée au niveau du système, mais est laissée à l'utilisateur

10.3 Threads : avantages et inconvénients

Avantages et inconvénients par rapport à des processus communiquants avec les "anciens" mécanismes

— partage de la mémoire : mécanisme rapide de communication inter-thread
— plus léger : moins de données système à recopier
— plus rapide : le context-switch est plus facile

Les inconvénients sont (uniquement) des "difficultés" de programmation :

— Les threads utilisent les mêmes copies des librairies : les librairies doivent être "MT-safe"
— Il faut gérer la synchronisation (mutex, sémaphores...)

En cas de mauvaise synchronisation :

— Comportement "aléatoire" : bugs, segfaults, exploitations...
— Interblocage

10.4

Ordonnancement des threads (et processus) :

— Coopératif :
Chaque thread doit explicitement rendre la main.
Problème : si un thread plante ou ne rend pas la main, les autres threads ne s'exécutent plus.
— Préemptif :
Le système peut arrêter n'importe quel thread, pour switcher à un autre thread (via un mécanisme de temporisation et d'interruption matérielle)

L'ordonnancement des systèmes d'exploitation "modernes" se fait préemptivement (Unix, windows...). Mais l'ordonnancement coopératif peut toujours exister au niveau utilisateur.

10.5 Modèle 1 :1 (threads système ou threads noyau)

— Chaque thread correspond à une entité ordonnancée par le noyau.
— L'ordonnancement est alors préemptive.
— Le comportement va être proche d'un ensemble de processus qui partagent leur mémoire et leurs données système.

Avantage : permet à un processus d'utiliser plusieurs coeurs

Implémentations de threads 1 :1 : LinuxThread, NPTL

10.6 Modèle N :1 (threads utilisateurs)

— Tous les threads du processus correspondent à une entité ordonnancée par le noyau.

— L'ordonnancement et le *switch* entre les threads se fait au niveau utilisateur

Avantage :
— le *switch* est très rapide (pas d'appel système)
— possibilité d'avoir énormément de threads
— possible même sur des systèmes légers, sans ordonanceur (systèmes embarqués)

Implémentations de threads N :1 : GNU Pth, State Threads
— Certains langages/paradigmes de programmation utilisent nativement le multi-threading
— C'est le cas notamment de ceux qui utilisent les *coroutines*.
— Une implémentation naïve donnerait trop de threads, et trop de "context switchs", pour être efficacement traités par le système.
— Ces threads utilisateurs légers, exécutions de coroutines, sont appelés des *fibres*

processus / thread système / thread utilisateur / fibre

10.7 Modèle M :N ("hybride")

— Tire les avantages des modèles 1 :1 et N :1
— Idéalement, N = nombre de coeurs

Exemples : GHC (Glasgow Haskell compiler), threads NetBSD...

10.8 Threads POSIX

Une norme POSIX pour créer des threads, et de les synchroniser

```
#include <pthread.h>
```

Compiler avec l'option `-pthread`

Intègre :
— Fonctions pour créer, attendre et tuer un thread
— Des mécanismes de synchronisation : mutex et variables de condition

10.9 L'implémentation Linux de pthread

— L'implémentation actuelle des threads POSIX sous Linux est NPTL (Native POSIX Threads Library). Elle respecte la norme POSIX, et rajoute quelques fonctions non POSIX.
— En interne, il s'agit de threads systèmes (préemptifs et 1 :1).
— NPTL utilise des appels système à `clone()` et `futex()`, et des signaux temps réels.

10.10 Threads POSIX : partage

Les pthreads d'un processus partagent :
— le PID, le PPID
— l'espace mémoire
— les descripteurs de fichiers ouverts
— les utilisateurs propriétaires
— les handlers des signaux
— le répertoire courant, le masque de fichiers...
Ne partagent pas :
— les identifiants des threads
— la pile
— le masque des signaux
— `errno` (exercice : comment cela est possible ?)

10.11 Créer un thread

Au départ, le processus est constitué d'un unique thread : celui qui exécute la fonction `main`

On peut créer d'autres threads, avec la fonction suivante :

```
int pthread_create(
    pthread_t *thread,
    const pthread_attr_t *attr,
    void *(*start_routine) (void *),
    void *arg
    );
```

— `thread` : l'adresse mémoire où sera copié l'identifiant du nouveau thread
— `attr` : des attributs (NULL = defaut)
— `start_routine` : la fonction d'entrée du thread
— `arg` : l'argument de la fonction
Exemple :

```
void *fonction(void *a)
{
  ...
  return NULL;
}

...
pthread_t id;
phtread_create(&id,NULL,fonction,NULL)
...
```

10.12 Identifiant d'un pthread

— Un pthread n'a pas de PID propre (ils partagent tous le même PID, celui du processus contenant les threads)
— L'identifiant d'un pthread est un objet du type `pthread_t`.
— La norme ne dit pas ce qu'il y a dans `pthread_t` (objet opaque).
— `phtread_self()` renvoie le pthread_t du thread courant.
— `phtread_equal(pthread_t a,pthread_t b)` permet de comparer deux identifiants

10.13 Argument et retour d'un pthread

— Il est possible de passer un argument à la fonction qu'on appelle : un pointeur, qu'on peut faire pointer vers la structure de son choix
— Quand start_routine termine, le thread se termine.
— Un thread peut également terminer avec pthread_exit()

— Le thread `main` est spécial, sa terminaison termine le processus, même si d'autres threads sont encore en exécution.
— exit() dans n'importe quel thread termine le processus (i.e. tous les threads)

10.14 Fin et attente d'un thread

Similairement à un processus, un thread renvoie un code retour : un pointeur.

Similairement à un fils qui devient zombi, le code retour du thread est gardée en mémoire jusqu'à ce qu'un autre thread le "rejoigne"

Il n'y a pas de notion de père/fils :
— n'importe quel thread peut joindre n'importe quel thread
— si plusieurs attentes du même thread : comportement indéfini

```
int pthread_join(pthread_t thread, void **retval);
```

`retval` : un pointeur sur une variable (contenant un pointeur), où le code retour sera copié.
— NULL : ignoré
— thread "annulé" : `PTHREAD_CANCELED`

Il existe des versions non bloquantes dans NPTL (non POSIX !) : `pthread_tryjoin_np`, `pthread_timedjoin_np`

10.15 Détacher un thread

Si on ne veut pas avoir à gérer la fin d'un thread, on peut le "détacher".

```
int pthread_detach(pthread_t thread);
```

— La structure sera détruite à la terminaison du thread
— Il ne sera pas possible de retrouver son pointeur retour avec `pthread_join`

10.16 Annuler un thread

```
int pthread_cancel(pthread_t thread);
```

Permet de d'"annuler" (de terminer) un thread

Attention : dans beaucoup de cas, on ne peut pas simplement terminer un thread sans risquer des fuites mémoires, ou des interblocages.

Il ne s'agit pas de "tuer" un thread : le thread doit préparer son annulation.

Il faut faire attention :
— aux fuites mémoires (mémoire dynamique allouée par le thread)
— aux sections critiques (par exemple, si le thread bloque un mutex)
Pour cela, on peut rajouter des "handlers" qui seront exécutés à l'annulation d'un thread

```
void pthread_cleanup_push(void (*routine)(void*), void *
    arg);
void pthread_cleanup_pop(int execute);
```

— push : rajoute dans une pile une nouvelle fonction à executer en cas d'annulation
— pop : enlève le dernier élement de la pile (et l'execute si `execute` n'est pas 0)
Le thread peut (doit) aussi dire quand il peut être annulé

```
int pthread_setcancelstate(int state, int *oldstate);
int pthread_setcanceltype(int type, int *oldtype);
void pthread_testcancel(void);
```

— `pthread_setcancelstate` (des)active la possibilité d'annulation du thread courant
— `pthread_setcanceltype` spécifie si l'annulation se fait immédiatement ou à un point d'annulation
— `pthread_testcancel` spécifie un point d'annulation

10.17 Attributs

`pthread_attr_t` : objet (opaque) spécifiant les attributs d'un thread.
— `pthread_attr_init` / `pthread_attr_destroy` : initialise (détruit) un attribut

— `pthread_attr_setstacksize` (`pthread_attr_setstackaddr`) permet de spécifier la taille (l'emplacement) de la pile
— `pthread_attr_setdetachstate` : spécifie l'état détaché
— Il est également possible de spécifier des paramètre d'ordonnancement

10.18 Autres choses de pthread

Des parties importantes de pthread seront vus par la suite :
— les *mutex*
— variables de condition
Ces mécanismes permettent de synchroniser les threads lors d'accès concurrents.

10.19 Comportement avec les signaux

— Les handlers des signaux sont partagés entre tous les threads.
— Mais chaque thread a son propre masque de signaux !
— On peut modifier le masque d'un thread via `pthread_sigmask` (mêmes arguments que `sigprocmask`).
— On peut envoyer un signal à un thread spécifique via `pthread_kill(pthread_t id,int sig)`
— On peut attendre un signal avec `sigwait`

10.20 Comportement avec `exec` et `fork`

— Comportement avec `exec` :
Si un thread fait un appel à `exec` (qui n'échoue pas), tous les autres threads sont tués.
— Comportement avec `fork` :
Seul le thread appelant fork est dupliqué.
Problème : comme les autres threads n'existent plus dans le fils, il se peut qu'un mutex ne soit jamais libéré.
Une solution est d'utiliser `pthread_atfork` qui rajoute des handlers en cas de `fork`.

10.21 Le début des problèmes...

— Les threads partagent leur mémoire. Y compris le segment des données des librairies.
— Les (fonctions des) librairies doivent donc être prévues pour un usage multi-thread
— Lors de la communication par mémoire partagée, il faut aussi faire attention à ce que deux threads ne travaillent pas sur la même zone mémoire en même temps
— Sinon : bug, plantage, exploitation, heisenbug...

10.22 Libraires "MT-safe"

— Quand on utilise une librairie (ou une fonction d'une librairie) dans un programme multi-threadé, il faut vérifier si elle a été prévue pour une utilisation multi-threadée (MT-safe)
— Certaines fonctions de la libc sont MT-safe, d'autres non
— Il faut voir le manuel !

Ré-entrance :
— Une fonction est appelée "re-entrante" si elle peut être interrompue, et rappelée (de façon sûre).

10.23 Exemple : `strtok`

```
#include <string.h>
char *strtok(char *str, const char *delim);
```

`strtok` coupe `str` aux caractères présents dans `delim`

Si `str=NULL`, elle continue de découper la chaîne précédente

```
char w[]="2:5:6:1:2:6:3";
char *p=strtok(w,":");
while (p) {
 printf("%s ",p);
 p=strtok(NULL,":");
}
```

Sortie : 2 5 6 1 2 6 3

Problème (avec les programmes multithreads) : strtok garde en interne un pointeur sur la position courante dans le chaîne.

Version ré-entrante : `strtok_r`

```
char *strtok_r(char *str, const char *delim, char **
    saveptr)
```

`strtok_r` ne garde pas un pointeur interne : il faut lui en fournir un

```
char *tmp;
char w[]="2:5:6:1:2:6:3";
char *p=strtok_r(w,":",&tmp);
while (p) {
 printf("%s ",p);
 p=strtok_r(NULL,":",&tmp);
}
```

10.24 Problème d'optimisation du compilateur

```
int i;

void *thread(void *a) {
  sleep(1);
  i=1;
}
...
pthread_create(id,NULL,thread,NULL);
i=0;
while(i==0) { usleep(1000); }
...
```

Avec trop d'optimisations, le compilateur peut considérer que `i` reste à 0, car jamais affectée à autre chose que 0 dans main.

Modificateur du C : `volatile`. Indique au compilateur qu'une variable peut changer entre ses différents accès.

10.25 Problèmes de synchronisation

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :

— y compris au milieu d'une ligne

— y compris au milieu d'une instruction basique en C (ex : `i++`)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.

— Comportement imprévisible !

```
long long z=0;

void* th(void *r) {
  for(long long a=0;a<1000000;a++)
    z++;
}

int main() {
  pthread_t id1,id2;
  pthread_create(&id1,NULL,th,NULL);
  pthread_create(&id2,NULL,th,NULL);
  pthread_join(id1,NULL);
  pthread_join(id2,NULL);
  printf("%Ld\n",z);
}
```

sortie : 948249

10.26 Atomicité

Code assembleur de `z++` :

```
movq    z(%rip), %rax
addq    $1, %rax
movq    %rax, z(%rip)
```

On aimerait que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.
Mais : il n'est pas possible de bloquer les interruptions dans le mode utilisateur.

Pour rendre atomique (vis à vis des autres threads) une accès sur une zone mémoire : mécanisme d'exclusion mutuelle (mutex)
C'est le sujet du prochain cours !

11 Threads (2) Synchronisation des processus concurrents : mutex

11.1 Introduction

Les threads partagent leur mémoire.

Le partage de mémoire est généralement voulu et avantageux :
— cela évite de gaspiller de la mémoire
— c'est un mécanisme de communication inter-thread (et inter-processus) très rapide
L'important est de bien savoir gérer l'accès concurrent à la mémoire

(Rappel : il faut faire attention aux fonctions/librairies non réentrantes, ou non "MT-safe")

11.2 Un exemple pour commencer...

Un thread peut être arrêté n'importe quand pour laisser sa place à un autre thread :
— y compris au milieu d'une ligne
— y compris au milieu d'une instruction basique en C (ex : `i++`)

Problème : si un thread A travaille sur une zone mémoire M, et est arrêté alors que M est inconsistante, le thread B trouvera M en état inconsistant.
— Comportement imprévisible ! (bug, plantage, exploitation, mauvaise note)

```
long long z=0;

void* th(void *r) {
  for(long long a=0;a<1000000;a++)
    z++;
```

```
}

int main() {
  pthread_t id1,id2;
  pthread_create(&id1,NULL,th,NULL);
  pthread_create(&id2,NULL,th,NULL);
  pthread_join(id1,NULL);
  pthread_join(id2,NULL);
  printf("%Ld\n",z);
}
```

sortie : 1020102 ou 1271305 ou 948249...

11.3 Atomicité

Code assembleur de `z++` :

```
movq    z(%rip), %rax
addq    $1, %rax
movq    %rax, z(%rip)
```

On aimerait que ces 3 instructions ne puissent être interrompues.

Instruction(s) atomique : instruction(s) ne pouvant être interrompues.

Pour que l'exemple précédent soit correct, il faudrait rentre l'instruction `z++` atomique.

11.4 Atomicité d'une instruction

L'atomicité peut se faire au niveau du processeur.

Il faut distinguer 2 choses :
- Atomicité vis à vis d'une interruption
  Une instruction processeur est atomique vis à vis d'une interruption.
- Atomicité vis à vis des autres coeurs
  Dans les ordinateurs multiprocesseurs et/ou multicoeurs, du fait de la pipeline, une variable peut changer entre sa lecture et son écriture. Elle n'est donc pas (par défaut) atomique vis à vis des autres coeurs .
  Sur x86 : on peut rendre atomique une instruction machine avec le préfixe `lock`.

11.5 Atomicité d'une instruction : exemple

Pour que le programme précédent fonctionne comme souhaité

— On peut incrémenter `z` avec l'instruction assembleur `incq`
⇒ l'opération sera atomique vis a vis des interruptions.
⇒ le programme fonctionnera correctement si la machine a un unique coeur.
— Pour que le programme soit "correct" dans le cas général :
Il faut rendre atomique l'instruction : `lock incq`
⇒ le programme fonctionne comme souhaité.

Problème : Du fait que la pipeline ne sert presque plus, c'est beaucoup plus lent...

11.6 Atomicité d'un ensemble d'instructions

En pratique, les opérations sur une zone mémoire prennent généralement plusieurs instructions

Problèmes :
— Il n'est pas possible (ni raisonnable) de bloquer les interruptions / les autres coeurs dans le mode utilisateur.
— Pourquoi ? Un processus malveillant/planté/bogué pourrait bloquer tous les autres processus...
— Il n'est pas possible d'utiliser un mécanisme du type `lock` sur un ensemble d'instructions
De toutes façons : on ne veut pas l'atomicité d'un ensemble d'instructions...
Cela bloquerait tous les coeurs pour accéder à une zone mémoire, alors que les autres ne travaillent pas forcement sur cette zone...

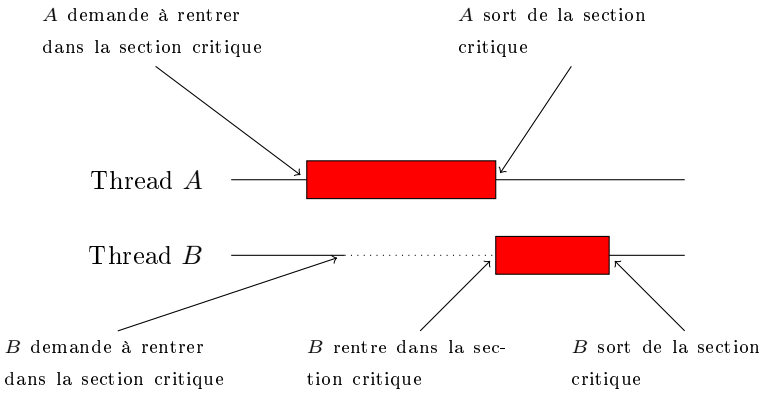
La bonne solution n'est pas d'avoir des sections atomiques, mais des sections où on a l'exclusivité sur une partie de la mémoire.

Cela s'appelle une section critique .

11.7 Sections critiques

Une section critique est une section du code où il n'y a au maximum qu'un thread à la fois

Si un thread B veut rentrer dans une section critique, et qu'un autre thread A est déjà dans la section critique, on doit faire attendre le thread B jusqu'à ce que le thread A sorte de la section critique.



Avantage :
— on ne bloque pas les autres threads qui ne travaillent pas sur la section critique
— il peut y avoir plusieurs sections critiques différentes, qui n'interfèrent pas entre elles.

Il faut un mécanisme pour entrer et sortir des sections critiques.

Ce qu'on attend d'un mécanisme de gestion des sections critiques :
— l'exclusion mutuelle : deux threads ne sont pas en même temps dans la section critique
— la progression : le processus continue de progresser dans tous les cas possibles d'exécution
— l'attente bornée : un thread qui demande à rentrer dans une section critique ne va pas attendre indéfiniment

Ce n'est pas un problème trivial. Plusieurs solutions fausses ont été publiées

Les problèmes en cas de mauvaise gestion des sections critiques :
— Si l'exclusion mutuelle est pas respectée :
Situation de compétition (race condition) : deux threads sont dans une section critique en même temps : non déterminisme (bug, plantage, exploitation...)
— Si la progression n'est pas respectée :
Interblocage (deadlock) : le processus est bloqué.
— Si l'attente bornée n'est pas respectée :
Famine (starvation) : un thread ne voit jamais sa demande aboutir

11.8 Exemple simple : 2 threads

Solution 1 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 while(intB) /*wait*/;
 intA=1;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 while(intA) /*wait*/;
 intB=1;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent être dans la section critique en même temps (situation de compétition)

Solution 2 (?)

```
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 intA=1;
 while(intB) /*wait*/;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 intB=1;
 while(intA) /*wait*/;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?

Non ! Les 2 threads peuvent se bloquer mutuellement (interblocage)

Solution 3 (?)

```
int rnd=0;
```

Thread A :

```
while(1) {
 while(rnd!=0) /*wait*/;
 //sect. critique
 rnd=1;
 //sect. normale
}
```

Thread B :

```
while(1) {
 while(rnd!=1) /*wait*/;
 //sect. critique
 rnd=0;
 //sect. normale
}
```

Correct ?

Non ! Quand le thread A termine, B est bloqué indéfiniment (famine)

Solution 4 (?)

```
int rnd=0;
int intA=0,intB=0;
```

Thread A :

```
while(1) {
 intA=1;
 rnd=1;
 while(intB && rnd==1)
     /*wait*/;
 //sect. critique
 intA=0;
 //sect. normale
}
```

Thread B :

```
while(1) {
 intB=1;
 rnd=0;
 while(intA && rnd==0)
     /*wait*/;
 //sect. critique
 intB=0;
 //sect. normale
}
```

Correct ?
Oui ! (Solution de Peterson)

11.9 Attente active et passive

L'attente avant d'entrer dans une section critique peut être :
— active (*spinlock*) : le thread continue de tourner jusqu'à ce qu'il a le droit d'entrer dans la section critique
Dans l'exemple précédent (Peterson), l'attente est active
— passive : le thread est mis en pause jusqu'à ce qu'il a la possibilité de rentrer dans la section critique
Dans ce cas, il faut interférer avec l'ordonnanceur
Avantage : on laisse le temps CPU aux autres threads qui peuvent faire des choses plus constructives

En général, on préfère l'attente passive. (Mais dans certains cas très particuliers, l'attente active peut être plus avantageuse.)

11.10 Les primitives

En général, on ne reprogramme pas soi même les tests d'entrée dans une section critique.
— C'est fastidieux
— le risque d'erreur est très important
— on ne tire pas parti des possibilités de l'OS.
On passe par des primitives (du langage, de librairies et/ou du système) : des verrous .
Il existe différents types de verrous :
— sémaphores (POSIX 1.b)
— mutex (pthreads)
— rwlocks (pthreads)
— barrières (pthreads)
— variables de condition / moniteurs (pthreads)

Attention ! Les verrous proposés par les langages/systèmes ne sont que des primitives (pour simplifier la vie).

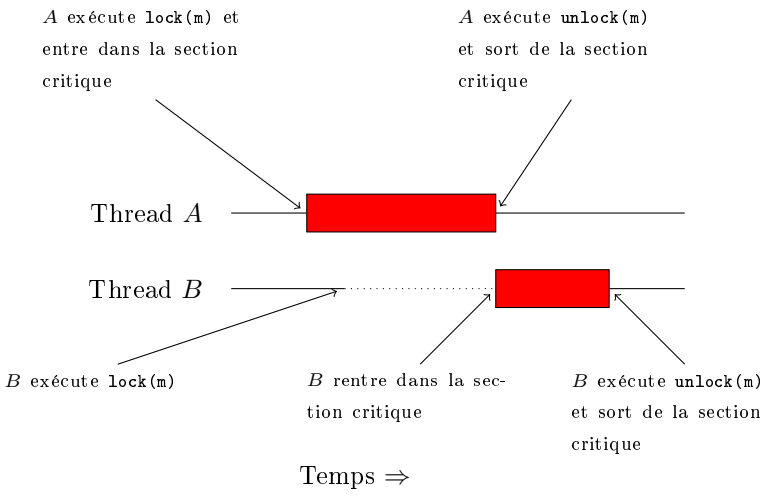
Une mauvaise utilisation peut toujours mener à des problèmes : situation de competition, interblocage ou famine...

11.11 Verrou le plus simple : le mutex

Mutex : assure qu'au plus un thread est dans la section critique à un moment donné.

Pseudo-code :

```
mutex m;

//section non critique
lock(m);
//section critique;
unlock(m);
//section non critique
..
```



11.12 Les mutex de `pthreads`

```
#include <pthread.h>

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

int pthread_mutex_init(pthread_mutex_t *mutex, const
    pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

```
int pthread_mutex_lock(pthread_mutex_t *mutex);
int pthread_mutex_trylock(pthread_mutex_t *mutex);

int pthread_mutex_unlock(pthread_mutex_t *mutex);
```

Exemple :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;

pthread_mutex_lock(&mutex);
//section critique
pthread_mutex_unlock(&mutex);
```

Note :

— Deux façons d'initialiser un mutex :

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

ou

```
pthread_mutex_t mutex;
pthread_mutex_init(&mutex,NULL);
```

— `mutexattr` permet d'initialiser un mutex avec d'autres attributs que ceux par défaut (récursif, partagé...).
NULL = défaut

— `trylock` essaye de bloquer le mutex. Si il échoue, il renvoie directement la main avec une erreur (retour $\neq 0$)

11.13 Implémentation d'un mutex (?)

Ici, un mutex est un entier.

```
int a=1;

lock(int &a) {
 while(a==0) /*wait*/;
 a=0;
}

unlock(int &a) {
 a=1;
}
```

Problème : pour que ce soit correct, le test et l'affectation doivent se faire atomiquement

11.14 Implémentation possible d'un mutex

```
int a=0;

lock(int &a) {
 int tmp=0;
 while(1) {
  xchg(a,tmp);
  if(tmp==1) break;
 }
}

unlock(int &a) {
 a=1;
}
```

`xchg(a,b)` échange (atomiquement) a et b. (instruction x86)
— Problème : attente active et possible famine

11.15 Pseudo-implémentation idéale d'un mutex

```
int libre=1;
queue liste_attente;

lock() { //atomiquement
 while(1) {
  if(libre) {
   libre=0;
   return;
  }
  liste_attente.push(thread_courant());
  wait();
 }
}

unlock() { //atomiquement
 if(liste_attente.non_vide())
   signal(liste_attente.pop());
 else
   libre=1;
}
```

11.16 Où sont implémentés les verrous ?

Problème des verrous en mode utilisateur :
— pour éviter l'attente active, il faut jouer avec l'ordonnanceur
— pour éviter les famines, il faut une file d'attente
Ces taches sont souvent laissées aux OS

Problème des verrous en mode noyau : les appels systèmes sont très lents !

Bon compromis : combiner les deux

11.17 Implémentation dans NPTL

— `lock()` sur un verrou libre : opération atomique
— `lock()` sur un verrou non libre : opération atomique +
appel système (`futex()`) qui met le thread en pause, et rajoute à une liste d'attente
— `unlock()` : opération atomique +
(si la liste d'attente est non vide) appel système à `futex` pour libérer le thread suivant.

— coté utilisateur : un boléen (libre ou non) et le nombre de threads en attente
— coté système : une liste d'attente

11.18 Mutex récursif

NPTL (et d'autres implémetations) introduisent d'autres possibilités (non POSIX, "non portables"). Par exemple :

— mutex récursif : l'action de bloquer un mutex déjà bloqué par le thread courant ne bloque pas. Exemple :

```
foo(int i)
{
  lock(mutex);
  // section critique
  if(i>0) foo(i-1);
  // section critique
  unlock(mutex);
}
```

11.19 Read-write locks

On pourrait permettre à plusieurs threads qui ne modifient pas la mémoire, de travailler (en lecture seule) sur une zone mémoire.

Une solution : read-write locks (rwlocks)

Deux types de sections critiques
— les sections critiques en lecture seule (celles des lecteurs)
— les sections critiques en lecture/écriture (celles des écrivains)
Garantie des rwlocks :
— si un thread est dans une section critique en lecture/écriture, il n'y a aucun autre thread dans une section critique (ni en lecture seule, ni en lecture/écriture)
— (si aucun thread est dans une section critique en lecture/écriture, il n'y a pas de limite sur le nombre de threads dans une section critique en lecture seule)

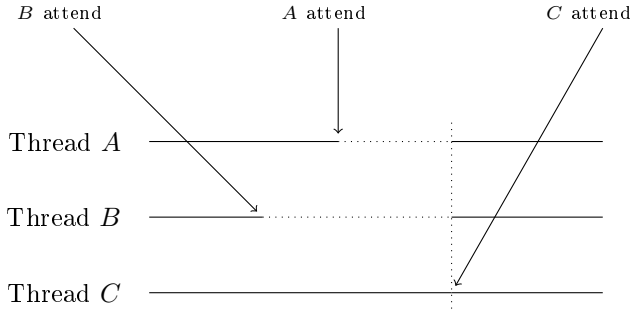
Attention : on peut facilement arriver à des famines
— préférer les lecteurs : il peut y avoir famine des écrivains
— préférer les écrivains : il peut y avoir famine des lecteurs

11.20 Les rwlocks de `pthreads`

```
#include <pthread.h>

pthread_rwlock_t lock = PTHREAD_RWLOCK_INITIALIZER;

int pthread_rwlock_init(pthread_rwlock_t * restrict lock, const
    pthread_rwlockattr_t * restrict attr);
int pthread_rwlock_destroy(pthread_rwlock_t *lock);

int pthread_rwlock_rdlock(pthread_rwlock_t *lock);
int pthread_rwlock_tryrdlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedrdlock(pthread_rwlock_t * restrict lock,
    const struct timespec * restrict abstime);

int pthread_rwlock_wrlock(pthread_rwlock_t *lock);
int pthread_rwlock_trywrlock(pthread_rwlock_t *lock);
int pthread_rwlock_timedwrlock(pthread_rwlock_t * restrict lock,
    const struct timespec * restrict abstime);

int pthread_rwlock_unlock(pthread_rwlock_t *lock);
```

11.21 Les barrières

Les barrières permettent de synchroniser les threads



11.22 Les barrières POSIX

```
int pthread_barrier_init(pthread_barrier_t *restrict barrier, const
     pthread_barrierattr_t *restrict attr, unsigned count);
int pthread_barrier_destroy(pthread_barrier_t *barrier);
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

— `count` : nombre de threads qui doivent attendre à la barrière

Exemple :

```
pthread_barrier_t barrier;
pthread_barrier_init(&barrier,NULL,3);
```

Puis dans chaque thread (A, B et C) :

```
// section non synchronisee
pthread_barrier_wait(&barrier);
// section synchronisee
```

11.23 Problèmes : vitesse

Le but des programmes parallèles est de gagner en vitesse.

En utilisant les mécanismes précédents (mutex...) un processus peut perdre du temps :

— dans les attentes qu'un verrou se libère (attente active ou passive)
— dans les instructions atomiques (plus lentes à cause de problèmes de cache)
— dans les appels systèmes relatifs aux (dé)blocage de verrous

Solutions :

— limiter la taille des sections critiques
— séparer les zones mémoires partagés
(idéalement : 1 zone mémoire = 1 verrou)
— mais sans faire trop d'entrées/sorties de sections critiques

Un problème d'échelle (de granularité) peut parfois se poser
— trouver le bon compromis
Exemple (bidon)

$$\sigma(n) = \sum_{1 \leq i \leq n : i | n} i$$

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;

void* th(void *r) {
  for(long long i=(long long)r;i<=n;i+=2)
    if(i%n==0) {
      pthread_mutex_lock(&m);
      s+=i;
      pthread_mutex_unlock(&m);
    }
  return NULL;
}
```

Mieux :

```
long long n,s=0;
pthread_mutex_t m=PTHREAD_MUTEX_INITIALIZER;
```

```
void* th(void *r) {
  long long tmp=0;
  for(long long i=(long long)r;i<=n;i+=2)
     if(i%n==0)
       tmp+=i;
  pthread_mutex_lock(&m);
  s+=tmp;
  pthread_mutex_unlock(&m);
  return NULL;
}
```

— plus d'opérations de calcul (ici, 2 additions en plus)
— mais beaucoup moins de sections critiques

11.24 Gros problèmes

En cas de mauvaise gestion des sections critiques :
— situation de compétition : bugs, plantages, morts (Therac-25)
— interblocage

Si on protège chaque section critique par un verrou adéquat, l'exclusion mutuelle devrait être respectée. Le problème sera généralement l'interblocage

11.25

Suite :
— Les 5 philosophes...
— Sémaphores
— Variables de condition (Moniteurs)
— Gérer les interblocages
— Concurrence en C++11

12 Threads (3) : Sémaphores et variables de condition

12.1 Introduction

— Les threads partagent leur mémoire
— Il faut protéger les accès mémoires concurrents
— Les mutex sont des verrous qui permettent de délimiter des sections critiques
— Mais des fois, les mutex ne suffisent pas...

On va voir :
— Les sémaphores
— Les variables de conditions

12.2 Problème classique : 5 philosophes

— 5 philosophes sont autour d'une table ronde
— Il y a une baguette entre chaque philosophe (i.e. : une baguette pour deux philosophes)
— Un plat de sushis pour tout le monde est au centre
— La vie d'un philosophe se résume à deux actions : penser et manger
— Pour manger, il doit prendre les 2 baguettes (à sa gauche et à sa droite), puis il peut commencer à manger
— Quand il a fini, il repose les baguettes et peut commencer à penser
— Chaque baguette : une ressource (un mutex)

But :
— Tout le monde doit pouvoir manger (pas de famine)
— Limiter les attentes

```
mutex baguette[5];

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct ?

Non : interblocage. Si tout le monde a la baguette à gauche, tout le monde est bloqué

12.3 5 philosophes : solution ?

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    printf("%d mange\n",i);
    unlock(&baguette[i]);
```

```
    unlock(&baguette[(i+1)%5]);
    unlock(&general);
  }
}
```

Correct... mais qu'un seul philosophe mange à la fois. Les mutex baguette[i] ne servent à rien → une seule section critique

```
mutex baguette[5],general;

void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    lock(&general);
    lock(&baguette[i]);
    lock(&baguette[(i+1)%5]);
    unlock(&general);
    printf("%d mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Mieux... mais un philosophe doit des fois attendre inutilement pour manger.

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    while(1) {
      lock(&general);
      if(baguette[i]==1 && baguette[(i+1)%5]==1) {
       baguette[i]=baguette[(i+1)%5]=0;
       unlock(&general);
       break;
      }
      unlock(&general);
    }
    printf("%d mange\n",i);
    baguette[i]=baguette[(i+1)%5]=1;
 }
}
```

Attente active et famine...

12.4 Parenthèse : Famine ?

On peut distinguer deux types de famines :

— la famine "avérée" : un thread est indéfiniment bloqué (quelle que soit le déroulement de la suite)

— la famine "probabiliste" : il y a une probabilité non nulle qu'un thread reste bloqué indéfiniment longtemps.

Si on s'autorise la famine "avérée", il existe une solution simple aux 5 philosophes sans interblocage, ni situation de compétition :

— On désigne un philosophe qui n'a pas le droit de prendre de baguettes (donc ni de manger et penser)

Si on s'autorise la famine "probabiliste" (mais pas "avérée"), la solution précédente est bonne (modulo le fait qu'elle soit en attente active)

Note :

— La famine est, des fois, pas très grave (e.g. si les threads font le même travail).

— Beaucoup considèrent que la famine probabiliste n'est pas un vrai problème.

Ordre d'importance :

famine "probabiliste" / famine "avérée" / interblocage / situation de compétition

12.5 5 philosophes : solution ?

Plusieurs (idées) de solutions

— Contre l'attente active : rajouter des temporisations aléatoires
  — bricolage : pas optimal (et toujours possible famine)

— Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer.
  — attente active (et possible famine)

— Prendre une baguette (lock), essayer de prendre l'autre (trylock). Si la deuxième n'est pas disponible, reposer la première (unlock) et recommencer dans le sens contraire .
  — (possible famine)

Note : si les philosophes sont sur une table linéaire (5 philosophes, 6 baguettes), la solution triviale marche.

Ce qui pose problème : les cycles

Solution : casser les cycles

```
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    if(i==0) {
     lock(&baguette[(i+1)%5]);
```

```
      lock(&baguette[i]);
    } else {
      lock(&baguette[i]);
      lock(&baguette[(i+1)%5]);
    }
    printf("%d mange\n",i);
    unlock(&baguette[i]);
    unlock(&baguette[(i+1)%5]);
  }
}
```

Correct. Mais pas très joli, et l'asymétrie introduit des biais (des philosophes attendent plus que d'autres en moyenne)

12.6 5 philosophes : Solution de Dijkstra

On limite à 4 le nombre de philosophes qui peuvent rentrer dans la phase "prendre des baguettes"

— au plus 4 arcs dans le graphe ⇒ pas de cycle.

Principe des sémaphores .

(Exercice : est il possible que le 5ème philosophe attende inutilement ?)

(Exercice : combien, au minimum, faut-il autoriser de philosophes dans la phase "prendre des baguettes" pour qu'il n'y ait pas d'attente inutile ?)

12.7 Les sémaphores

Un autre type de verrou est le sémaphore

— Introduit par Dijkstra ($\sim$ 1963)
— Premier verrou à apparaître dans un vrai système
— Plus général qu'un mutex
— (Mais il a plutôt un intérêt historique et, ici, didactique)

— 3 opérations :
  — init(N) : initialise le sémaphore à N "ressources"
  — P() (ou wait(), ou down()) : demande une ressource. Si il n'y a plus de ressource libre, attend jusqu'à ce qu'une ressource se libère
  — V() (ou signal(), ou post(), ou up()) : libère une ressource
— garantie : au plus N threads possèdent une "ressource"

12.8 Sémaphore : différence avec un mutex

— Un sémaphore avec N=1 simule (un peu près) un mutex
— (Un sémaphore avec N=1 est un sémaphore binaire)
— Mais : le mutex est un booléen, le sémaphore est un entier
  — unlock();unlock(); n'aura pas le même comportement que V();V();

— Mais : un mutex doit être débloqué par le thread qui l'a bloqué.
  — On peut se servir d'un sémaphore comme d'un "signal" pour débloquer un autre thread

```
init(sem,0);
```

Thread A :

```
sleep(2);
// pas syncro
V(sem);
//synchro
```

Thread B :

```
sleep(1);
// pas syncro
P(sem);
//synchro
```

12.9 Sémaphores POSIX

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value)
    ;
int sem_destroy(sem_t *sem);

int sem_wait(sem_t *sem);
int sem_trywait(sem_t *sem);
int sem_timedwait(sem_t *sem, const struct timespec *
    abs_timeout);

int sem_post(sem_t *sem);
```

— `pshared` = 0 : le sémaphore n'est pas partagé avec un autre processus (uniquement entre les threads de ce processus)
— `pshared` = 1 : le sémaphore est partagé. `sem` doit être dans une zone mémoire partagé entre les différents processus

12.10 5 philosophes avec sémaphore

```
mutex baguette[5];
semaphore sem;
init(sem,4);

void *philosophe(void *a) {
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);
    wait(sem);
    lock(baguette[i]);
    lock(baguette[(i+1)%5]);
    post(sem);
    printf("%d mange\n",i);
```

```
      unlock(baguette[i]);
      unlock(baguette[(i+1)%5]);
    }
}
```

Parfait ! pas de famine (même "probabiliste"), pas d'attente inutile.

12.11 Sémaphore : Implémentation ?

La première solution (Dijkstra) utilisait des blocages d'interruptions.

Cela n'est plus valable sur des machines multiprocesseurs, mais cela peut être simulé avec un mutex

```
init(int &s,int N) {
  lock(mutex);
  s=N;
  unlock(mutex);
}

V(int &s) {
  lock(mutex);
  s++;
  unlock(mutex);
}
```

```
P(int &s) {
  while(1) {
    lock(mutex);
    if(s>0) {
      s--;
      unlock(mutex);
      return;
    }
    unlock(mutex);
  }
}
```

— Attente active et famine

Comment implémenter un sémaphore avec une attente passive ?

C'est les mêmes problèmes qu'on avait déjà vu avec les mutex :

— il faut une file (éviter la famine)

— il faut communiquer avec l'ordonnanceur (pour éviter l'attente active)

(Les sémaphores sont disponibles dans POSIX, mais imaginons que ce ne soit pas le cas.)

Comment peut on faire pour les reprogrammer correctement ?

On ne peut pas simuler un sémaphore avec des mutex seuls

Mais on peut le faire avec un mutex + une variable de condition

12.12 Variable de condition

Plus généralement, supposons qu'on veut qu'un thread bloque jusqu'à ce qu'une condition (qui peut être compliquée) soit vérifiée

```
//...
while(1) {
  lock();
  if(condition()==1) {
    // operations d'entree de section critique
    unlock();
    break;
  }
```

```
  unlock();
}
//section critique
lock();
// operations de sortie de section critique
unlock();
//...
```

On voudrait avoir le même comportement que le code ci-dessus, mais sans les famines et dans l'attente active. Comment faire ?

Variable de condition : primitive qui permet de mettre en attente (dans une queue) un thread en débloquant (atomiquement) un mutex

```
mutex m;
cond c;

//...
lock(m);
while(condition()==0)
  wait(c,m);
// operations d'entree de section critique
unlock(m);
//section critique
lock(m);
// operations de sortie de section critique
signal(c);
unlock(m);
//...
```

Une variable de condition fonctionne toujours de pair avec un mutex

3 primitives :

- wait(c,m) : (atomiquement) débloquer m, mettre le thread en pause, et le rajouter dans la queue de c
  - au moment de l'appel, m doit être bloqué !
- signal(c) : débloque le premier thread de la queue de c
- broadcast(c) : débloque tous les threads de la queue de c

12.13 signal ou broadcast ?

Si tous les threads en attente attendent sur la même condition : signal()

Si les threads ont des conditions différentes : broadcast()

Sinon : un thread est débloqué, mais si sa condition n'est pas validée, il va devoir re-entrer en sommeil. Si un autre thread a sa condition validée, il ne sera pas réveillé par défaut.

12.14 Variables de condition dans pthread

```c
#include <pthread.h>

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;

int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t
     *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *
    mutex);
int pthread_cond_timedwait(pthread_cond_t *cond,
    pthread_mutex_t *mutex, const struct timespec *abstime);

int pthread_cond_signal(pthread_cond_t *cond);

int pthread_cond_broadcast(pthread_cond_t *cond);
```

12.15 5 philosophes avec mutex+cond

```c
void *philosophe(void *a)
{
  int i=(long long) a;
  while(1) {
    printf("%d pense\n",i);

    pthread_mutex_lock(&mutex);
    while(baguette[i]==0 || baguette[(i+1)%5]==0)
      pthread_cond_wait(&cond,&mutex);
    baguette[i]=baguette[(i+1)%5]=0;
    pthread_mutex_unlock(&mutex);

    printf("%d mange\n",i);

    pthread_mutex_lock(&mutex);
    baguette[i]=baguette[(i+1)%5]=1;
    pthread_cond_broadcast(&cond);
    pthread_mutex_unlock(&mutex);
  }
}
```

12.16 Sémaphores avec mutex+cond

```
typedef struct {
  int n;
  pthread_mutex_t m;
  pthread_cond_t c;
} sem_t;

void sem_init(sem_t *s,
              int n)
{
  s->n=n;
  pthread_mutex_init(
        &s->m,NULL);
  pthread_cond_init(
        &s->c,NULL);
}
```

```
void sem_post(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 s->n++;
 pthread_cond_signal(&s->c);
 pthread_mutex_unlock(&s->m);
}

void sem_wait(sem_t *s)
{
 pthread_mutex_lock(&s->m);
 while(s->n<=0)
  pthread_cond_wait(&s->c,
                    &s->m);
 s->n--;
 pthread_mutex_unlock(&s->m);
}
```

12.17 Pour finir...

— Moniteurs = mutex + variable de condition associée au mutex

— Généralise les autres primitives

— Par exemple, en C++11 : les seules primitives introduites dans la STD sont `<mutex>` et `<condition_variable>`

13 Interlude : Concurrence C++11

13.1

Le C++11 introduit plusieurs classes pour gérer les threads et la concurrence :

— threads
— mutex
— variables de condition

13.2 <thread>

— `std::thread` représente un thread (similaire à `pthread_t`)
— le thread est lancé à la construction :

```
void fct(int a, double b) {
 //...
}

std::thread th(fct,42,3.14);
```

— grâce à la "magie" des templates, pas besoin de jouer avec un argument `void*`

— `std::thread` est remplaçable (mais pas copiable). `operator=` déplace le thread.

```
std::thread th[42];

for(int i=0;i<42;i++)
  th[i]=std::thread(foo,i);
```

— Fonctions membres : `join()`, `detach()`, `swap()`
— si on détruit un std : :thread alors qu'il correspond à un thread en exécution, non détaché : exception
— pas de code retour (voir <future>)

13.3 <mutex>

— `std::mutex` et `std::recursive_mutex`
— Fonctions membres : `lock()`, `try_lock`, `unlock`
— `std::lock_guard` est un conteneur pour un mutex (il le bloque à sa construction, et le débloque à sa destruction)

```
std::mutex m;
//..
{
 std::lock_guard<std::mutex> l(m);
 //section critique
}
// section normale
```

— particulièrement utile pour que le mutex soit débloqué automatiquement en cas d'exception
— `std::unique_lock` est un conteneur plus évolué.
— Il a notamment les mêmes fonctions membres qu'un mutex, ce qui permet de s'en servir comme un mutex (avec l'assurance qu'il sera débloqué en cas de destruction)

```
std::mutex m;
std::uniquelock<std::mutex> l(m,std::defer_lock);

l.lock();
//section critique
l.unlock();
```

13.4 `std::lock()`

— `std::lock(m1,m2,...)` permet de bloquer plusieurs mutex à la fois
— Algorithme :
  — bloque le premier mutex m_1,
  — puis essaye de bloquer les autres avec `try_lock()`.
  — Si un mutex $m_i, i > 1$ échoue, il débloque tous les autres : $m_1, \ldots m_{i-1}$,

— puis recommence en commençant par m_i.

— Sans deadlock, et sans attente active.

13.5 5 philosophes en C++11

```
#include <thread>
#include <mutex>
std::mutex baguette[5];

void philosophe(int i) {
  while(1) {
    printf("%d pense\n",i);
    lock(baguette[i],baguette[(i+1)%5]);
    printf("%d mange\n",i);
    baguette[i].unlock();
    baguette[(i+1)%5].unlock();
  }
}

int main() {
  std::thread id[5];
  for(int i=0;i<4;i++)
    id[i]=std::thread(philosophe,i);
  id[0].join();
}
```

— Famine? (voir l'algo de `lock()`...)

13.6 <condition_variable>

— `std::condition_variable` est une variable de condition

— constructeur sans argument

Fonctions membres :

— `wait(l)` (`l` doit être un `unique_lock<mutex>`)

— `notify_one()` = signal

— `notify_all()` = broadcast

13.7 <future>

Permet d'accéder au résultat de procédures asynchrones

```
int carre(int x) {
  for(long long i=0;i<1000000000;i++);
  return x*x;
}

int main()
{
  std::future<int> res=std::async(carre,42);
  printf("resultat=%d\n",res.get());
```

```
  return 0;
}
```

Ou bien avec des promesses :

```
void carre(int x,std::promise<int> pr) {
  for(long long i=0;i<1000000000;i++);
  pr.set_value(x*x);
}

int main()
{
  std::promise<int> pr;
  std::future<int> res=pr.get_future();
  std::thread th(carre,42,std::move(pr));
  printf("resultat=%d\n",res.get());
  th.join();
  return 0;
}
```

13.8 <atomic>

Permet de faire des opérations atomiques sur une variable

```
std::atomic<int> atint;

void th() {
  for(int i=0;i<10000000;i++)
    atint++;
}

int main() {
  atint.store(0);
  std::thread th1(th);
  std::thread th2(th);
  th1.join();
  th2.join();
  printf("%d\n",atint.load());
  return 0;
}
```

13.9 Variables `thread_local`

En C/C++ : il y a deux types de variables :
— les automatiques (ou locales) : dans la pile (défaut, mot clef `auto`)
— les statiques (ou globales) : dans le segment de données (mot clef `static`)

Le C++11 rajoute le type `thread_local`
— la variable sera locale au thread

14 Gestion des interblocages

14.1 Conditions pour avoir un interblocage

Un interblocage arrive quand les 4 conditions suivantes sont vérifiées en même temps : [Coffman 1971]

— exclusion mutuelle : la/les ressource(s) n'est pas partageable
— "hold and wait" : le(s) thread(s) a déjà bloqué une ressource, et en demande une autre
— non-préemption : c'est le thread qui libère par lui même les ressources
— attente circulaire : il existe une chaîne de processus $P_1 \dots P_k$ telle que chaque processus P_i bloque une ressource R_i et chaque processus P_i demande la ressource $R_{(i+1)\%k}$.

Exemple : les 5 philosophes ont chacun la fourchette de gauche, et attendent la fourchette de droite.

14.2 Prévenir et éviter les interblocages

Briser une des 4 conditions :

— Enlever l'exclusion mutuelle : par exemple
  — remplacer par des opérations atomiques.
  — algorithmes "sans blocages" : utilisation d'opérations atomiques "lecture-modification-ecriture"
— Enlever "hold and wait" : Exemple :
  — s'imposer de demander au plus 1 ressource à la fois.
  — bloquer plusieurs mutex atomiquement en même temps
  — si on demande plusieurs mutex, on fait attention à ne pas bloquer si on tient déjà un mutex
— Préemption : Difficile à faire... faudrait que cela soit prévu par les primitives et le programme

Contre l'attente circulaire : ne pas créer de cycles dans le graphe

— Par exemple : Solution de Dijkstra pour les 5 philosophes
— Solution générale possible : mettre un ordre total sur toutes les ressources, et demander les ressources dans l'ordre
— Le graphe de dépendance sera toujours un sous graphe d'un graphe acyclique
— Exemple : 5 philosophes asymétriques : un des philosophes prend les fourchettes dans l'autre sens.
— Une solution simple dans le cas général : adresse mémoire du mutex

Si on a (en plus) les informations de quelles ressources, ou combinaisons de ressources, peuvent être demandés, il est possible de prévenir dynamiquement les cycles.

Idée : On connaît le graphe des dépendances possibles, et on connaît le sous graphe des dépendances actuelles.

Il suffit de bloquer l'accès à une ressource qui pourrait créer un cycle. (Rester dans des états "sains")

Exemple : algorithme du Banquier de Dijkstra

Exemple (5 philosophes)

— P_0 prend f_0, P_1 prend f_1, P_2 prend f_2, P_3 prend f_3
— P_4 demande f_4. Lui donner ? Non !
— P_0 possède f_0 : l'arc $f_0 \to f_1$ est possible.
— Etc. $f_0 \to f_1 \to f_2 \to f_3 \to f_4$ est possible
— Donner f_4 à P_4 créerait un cycle : un interblocage est possible.
— La demande de P_4 pour f_4 bloque jusqu'à ce que cette possibilité soit écartée

14.3 Détecter les interblocages

Il est théoriquement possible (pour l'OS) de détecter les interblocages.

Par exemple, si on a que des mutex (exclusion mutuelle, et pas de préemption), il suffit de construire le graphe de dépendance :

— Pour tout thread t bloqué dans un lock(R_t), tous les arcs entre $x \to R_t$, pour tout les x bloqués par t.

Et de tester ce graphe contient un cycle. Si oui : il y a un interblocage...

Mais que peut-on faire si on détecte un interblocage ?

14.4 Sous Linux ?

Le noyau Linux :

— S'occupe qu'il n'y ait pas d'interblocage dans le noyau
— Mais ne détecte/résout pas les interblocages des processus.

Pourquoi ?

— Quand on est dans une situation d'interblocage, on ne peut pas débloquer en respectant l'exclusion mutuelle
  — à part "tuer" quelque chose...
— On ne peut pas savoir à l'avance si on va arriver à une situation d'interblocage :
  Il faudrait analyser le code pour savoir les dépendances possibles...
  On frise avec des problèmes indécidables

Il faudrait des primitives de verrouillage beaucoup plus compliquées. Cela en vaut il la peine ?

14.5 Livelock

Un autre type d'interblocage : le livelock

Aucun thread n'est bloqué, mais aucun thread n'avance (le code exécuté ne sert qu'à la gestion de concurrence)

Exemple : excès de politesse. Un thread veut laisser sa place pour une ressource à un autre thread si il le demande. Chaque thread se passe la main.

15 Concurrence : Ordonnancement

15.1 Ordonnancement : rappels

(Ici thread = thread ou processus non multi-threadé)

L'ordonnancement est préemptif.

Travail de l'ordonnanceur :
— Choisir à quel thread (prêt) il doit donner la main, et
— (pour les ordonanceurs préemptifs) combien de temps il lui donne.

15.2 État des threads

États des threads considérés par l'ordonnanceur :
— exécution : le thread est en exécution (sur un des coeurs)
— prêt : le thread attend que l'ordonnanceur lui donne la main
— en attente : le thread ne peut/doit pas être exécuté pour le moment (en attente d'une IO ou d'un mutex, sleep...)
— terminé

15.3 Changement d'état

Les threads "vivants" changent constamment d'état (exécution, prêt, attente)
— exécution → attente : appel système sur une ressource bloquante
— attente → prêt : la ressource se libère
— prêt → exécution : l'ordonnanceur donne la main au thread (dispatch)
— exécution → prêt :
  — le thread décide par lui même de rendre la main (POSIX : `sched_yield()`),ou
  — (ordo préemptif) le temps accordé au thread est dépassé (interruption)

15.4 Les différents temps

— temps total ("réel") : temps total d'un processus (de son création à sa terminaison)
— temps user : temps passé en mode utilisateur (temps passé en "exécution")
— temps système : temps passé en mode système (dans les appels systèmes)
— temps d'attente : temps passé en état "prêt"
(Note : pour voir les temps réels, user et sys : `time` *`commande`*).

15.5 Objectifs d'un ordonnanceur

Un ordonnanceur doit avoir plusieurs objectifs :
— Utiliser le(s) CPU à 100%
— Respecter l'équité
— Respecter les priorités ("nice")
— Minimiser le temps total d'une tache courte. (Réactivité. Par exemple : une commande.)
— Minimiser le temps total pour un long travail
— Éviter de faire trop de context-switch (par exemple, accorder des temps trop courts)
— Éviter de passer trop de temps dans l'ordonnanceur
— ...

15.6 Algorithmes l'ordonnancement

— Problème difficile
— Il y en a plusieurs possibles, en fonctions des objectifs principaux
— Cela pourrait être le sujet de tout un cours...
— Les algorithmes simples ont souvent des problèmes.

Stratégies "typiques" :
— First-Come, First-Served (FCFS) : (coopératif). Algo "minimal", on ne réfléchit pas. Peu de context switch. Problème : le temps de réponse peut être long (infini). Pas de gestion des priorités.
— Shortest-Job-First (SJF) : résout le problème de la réactivité.
Problème : il faut connaître (ou prédire) le temps d'un processus a priori.

15.7 Algorithmes d'ordonnancement : Round-Robin

Round-Robin (RR) : (ordo préemptif)
— L'ordonnanceur a une liste (ou queue) L de threads "prêt".
— L'ordonnanceur prend (et retire) le premier thread (T) de la liste
— L'ordonnanceur exécute T, avec une limite de temps déterminé (ex : 0.1 seconde).
— Quand T rend la main (ex : IO bloquant), ou a épuisé son temps autorisé, l'ordonnanceur le rajoute à la fin de L.

Problèmes :
— Pas de gestion de la priorité.
— Si T fait souvent des appels à des I/O bloquantes, il est laissé.

15.8 Avec les priorités ?

Solutions :
— Avoir plusieurs listes (une par priorité)
Si beacoup de priorités, un peu lourd...
Décisions à prendre : quelle liste dispatcher ?

— Utilisation de files de priorités (plutôt que des listes/queues)
(permet aussi de dépénaliser les threads qui font beaucoup d'I/O bloquantes)

15.9 Et avec plusieurs processeurs/coeurs ?

Un thread est généralement associé à un coeur
Pourquoi ?
— histoire de cache
— histoire de mémoire...

Mais si un coeur est moins utilisé qu'un autre, l'ordonnanceur peut décider de déplacer un thread.

L'ordonnanceur doit choisir quel coeur associer à un thread
Et décider quand le changer de coeur (load balance)

15.10 Parenthèse : UMA / NUMA

Architecture UMA (uniform memory access)

— une mémoire centrale partagée par plusieurs processeurs/coeurs
Exemples : vos machines (Intel Core ix, smartphones...)

Problème :
— si il y a beaucoup de coeurs, goulot d'étranglement pour l'accès mémoire
— les contrôleurs de mémoire sont (maintenant) souvent intégrés aux processeurs, et il difficile de concevoir des processeurs avec beaucoup de coeurs (plus de pertes, problème de dissipation thermique)
Architecture NUMA (non-uniform memory access)
— Chaque processeur (ou groupe de coeurs) dispose de sa mémoire (et forme un noeud)
— Mais chaque noeud peut accéder à toute la mémoire de la machine.
— Si c'est une mémoire d'un autre noeud, il faut passer par un bus qui inter-connecte les différents noeud (bus très rapide, mais quand l'accès est quand même plus lent)
Exemples : Systèmes multiprocesseurs courants (Intel Xeon, AMD Opterons...)

— Optimalement, il faudrait qu'un thread soit exécute dans le noeud où est sa mémoire

Contrainte supplémentaire pour l'ordonnanceur...

— Il faut interférer avec l'ordonnanceur mémoire
— déplacer un thread d'un noeud à un autre est plus contraignant

15.11 Ordonnanceur(s) de Linux

Plusieurs ordonanceurs au cours de l'histoire de Linux

— O(n) scheduler (Linux 2.4 : 2001-2011)
  — le temps est divisé en "epoch". À chaque epoch, chaque thread a droit à un certain nombre de temps CPU (basé sur la priorité)
  — Si un thread n'a pas épuise tout son temps CPU au cours d'une epoch, il rajoute la moitié du temps restant à son temps à l'epoch suivante.
  — Problème : temps linéaire en le nombre de processus entre chaque epoch
— O(1) scheduler (Linux 2.6, avant 2.6.23 : 2003-2007)
  — 140 niveaux de priorités (0-99 pour le système et temps réel, 100-139 pour les utilisateurs)
  — 2 queues par priorité : actif/inactif
  — pénalité si temps écoulé, récompense si I/O bloquante

Actuellement : Completely Fair Scheduler

— une file de priorité : arbre rouge-noir, indexé par le temps utilisé par le processus
— temps accordé : le temps que le thread a attendu $\times$ le ratio du temps qu'il aurait utilisé sur un processeur "idéal"
— l'ordonnanceur donne la main au thread qui a moins utilisé son temps (le plus petit dans la liste)
— quand le thread rend la main, l'ordonnanceur réinsère le thread dans l'arbre rouge-noir, avec son nouveau temps
— les processus avec beaucoup d'attente sont naturellement "récompensés"

15.12 Politiques et priorités POSIX

(Voir `man sched`)

On peut interférer avec l'ordonnanceur pour changer les politiques ou les priorités

(Généralement, plutôt utile pour les applications temps réel.)

Déjà vu : `nice()` pour un processus

Il y a aussi `getpriority()` / `setpriority()`. (Priorité d'un processus, d'un groupe de processus, et d'un utilisateur.)

15.13 Différentes politiques POSIX

Différentes politiques d'ordonnancement sont prévues dans POSIX (pour les processus et threads) :

— SCHED_OTHER : politique "standard"

Des politiques "temps réel" :

— SCHED_FIFO : (First-in-First-Out)

— SCHED_RR : (Round-Robin)

Dans ce cas, il y a une priorité supplémentaire (généralement entre 0 et 99) pour le thread

(Et quelques autres politiques, apparues plus récemment)

Pour changer la politique d'un processus : `sched_setscheduler()`

Pour changer la politique d'un thread : `pthread_setschedparam()`

15.14 Affinités

On peur vouloir contrôler sur quel noeud un processus tourne, ou répartir ses threads sur différents noeuds, pour avoir de meilleurs performances.

Il est possible de choisir (sous Linux) sur quel(s) coeur(s) un thread peut tourner, avec `sched_setaffinity`.

Il est aussi possible de choisir des affinités mémoires. Voir `man numa`

15.15 Problème : Inversion de priorité

— A de forte priorité

— B de priorité moyenne

— C de faible priorité

Scénario :

— C bloque une ressource R

— A demande (et bloque) sur la ressource R

— B arrive

— B prend 100% CPU, car il a la priorité sur C

— C ne relâche pas R

— A ne peut pas être exécuté

B bloque A (par l'intermédiaire de C).

(problème réel : Mars pathfinder)

Solutions :

— ne pas trop prioriser les threads de priorité supérieure. (Pas valable en temps réel)

— aléatoirement, donner du temps CPU à des taches de priorité basse (bricolage...)

— (priority ceiling) donner une priorité à un mutex. Si un thread bloque le mutex, il hérite (temporairement) de la priorité du mutex (si elle est plus haute)
— (priority inherance) si un thread C bloque le mutex, et un thread A de plus haute priorité demande le mutex, C hérite (temporairement) de la priorité de A.

Dans pthread :

```
int pthread_mutexattr_getprotocol(
    const pthread_mutexattr_t *attr,
    int *protocol);
int pthread_mutexattr_setprotocol(
    pthread_mutexattr_t *attr,
    int protocol);
int pthread_mutex_setprioceiling(
    pthread_mutex_t* mutex,
    int prioceiling, int* old_ceiling );
```

`protocol` :
— PTHREAD_PRIO_NONE : le fait de bloquer le mutex n'affecte pas la priorité
— PTHREAD_PRIO_PROTECT : (priority ceiling) bloquer le mutex donne au thread la priorité du mutex (s'il est supérieur)
— PTHREAD_PRIO_INHERIT : (priority inherance) bloquer le mutex donne au thread le maximum des priorités des threads demandant le même mutex

16 Partage de mémoire entre processus (POSIX)

16.1 Mémoire partagée inter-processus

Plusieurs processus peuvent partager leur mémoire : c'est un IPC très rapide

Attention : comme pour les threads, il faut gérer la concurrence, soit :
— avec des sémaphores (pshared=1)
— avec des mutex partagés (attribut pshared)

16.2 Mémoire partagée inter-processus : cas simple

Entre père et fils :

```
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_SHARED|
    MAP_ANONYMOUS,0,0);
if(fork()==0) {
  /*fils*/
  strcpy(x,"COUCOU");
  return 0;
```

```
}
/* pere */
sleep(1);
printf("%s\n",x);
```

Affiche : COUCOU

16.3

Sans lien de parenté :

```
int fd=open("partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)
    ;
```

Processus 1 :

```
while(1) {
 snprintf(x,999,"COUCOU_%d",rand());
 sleep(1);
}
```

Processus 2 :

```
while(1) {
 sleep(1);
 printf("%s\n",x);
}
```

— Cela marche, mais on utilise le disque (plus lent que la RAM)
— On voudrait mimer ce comportement, sans passer par le disque

16.4 Mémoire partagée inter-processus : `shm_open`

Solution : objets mémoire partagé (`man shm_overview`)

```
int fd=shm_open("/partage",O_RDWR|O_CREAT,0644);
ftruncate(fd,65536);
char *x=mmap(NULL,65536,PROT_READ|PROT_WRITE,MAP_SHARED,fd,0)
    ;
```

(Processus 1 et 2 comme précédemment)
— OK !
— En fait `/partage` sera dans un système de fichier en RAM, dans `/dev/shm/`

16.5 Objet mémoire POSIX

```
#include <sys/mman.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
int shm_open(const char *name, int oflag, mode_t mode);
int shm_unlink(const char *name);
```

(compiler avec `-lrt`)

Renvoie un descripteur de fichier, sur lequel on peut faire les opérations qu'on a déjà vues :
`ftruncate`, `mmap`, `munmap`, `close`...

16.6 Sémaphore partagé entre processus

(Voir `man shm_overview`)

Deux façons de créer un sémaphore partagé :
— Sémaphore anonyme
— Sémaphore nommé

Sémaphore anonyme : le sémaphore doit être crée avec `sem_init`, avec `pshared=1`, dans une zone mémoire partagée (via `mmap`)
Sémaphores nommés (mécanisme similaire à `shm_open`) :

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

sem_t *sem_open(const char *name, int oflag);
sem_t *sem_open(const char *name, int oflag, mode_t mode,
    unsigned int value);
```

(compiler avec `-pthread`)

16.7 Mutex partagé entre processus

```
pthread_mutexattr_t mutexattr;
pthread_mutexattr_init(&mutexattr);
pthread_mutexattr_setpshared(&mutexattr,
   PTHREAD_PROCESS_SHARED);

pthread_mutex_init(pmutex,&mutexattr);
//...
```

`pmutex` doit être dans une zone mémoire partagée

Similairement, les variables de condition, rwlocks, barrières... peuvent aussi être partagés entre processus.

17 Réseau : introduction

17.1 Introduction

Réseau :
- Ensemble de noeuds
- Interconnecté (mais pas forcément tous connectés 2 à 2)
- Pour faire circuler des éléments ou des flux, même entre 2 noeuds non voisins

Ici : Réseaux informatiques, échange de données (séquence de bits ou d'octets, en paquets ou en flux)

Applications :
- Partage de ressources (données, imprimante, machine de calcul...)
- communication (mail, voix, visioconf...)

17.2 Exemples de réseaux :

- Réseau local (LAN : local area network)
- Réseau au étendu (WAN)
- "Internet"
- Téléphone cellulaire
- ...

Ce cours : principalement LAN et Internet

17.3 Internet

Désigne à la fois :
- Le réseau Internet (interconnexion de réseaux)
- Le nom d'un ensemble de protocoles ("TCP/IP")

Histoire rapide d'Internet
- Fin années 60 : ARPANET : réseau (militaire) censé être résistant aux attaques.
  - inter-universités (en contrat avec le Departement of Defense)
  - Commutation de paquets
  - Invention de TCP/IP pour la communication entre réseaux différents

Années 80 - 90 :
- Ouverture aux universités (CSNET, NSFNET), puis au privé

1989 : WWW (World Wide Web), hyperliens

17.4 Problématiques

- Acheminer les données
  - router les données
  - trouver et mettre à jour les routes
  - limiter les congestions

— Assurer que les données arrivent en l'état
  — vérifier s'il y a des erreurs
  — corriger les erreurs
  — transmettre les données dans l'ordre,
  — sans doublons

Objectifs :
— Vitesse de transmission (débit)
— Temps de latence : temps entre l'émission et la réception

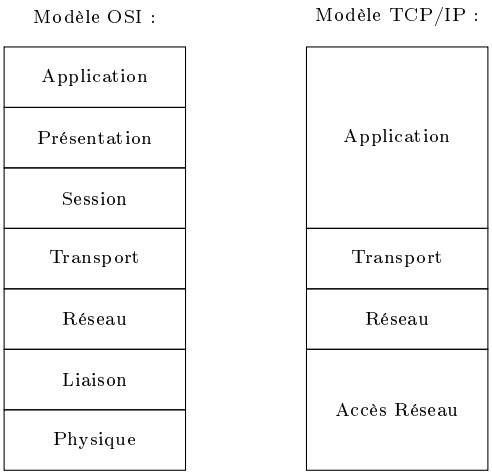
17.5 Topologie

Réseau ⇔ graphe connexe
Topologie (type de graphe) :
— étoile, arbre (Ethernet)
— cycle (anneau)
— graphe quelconque (Internet...)
— graphe complet
— grille, hypercubes (HPC)...

— homogène (LAN Ethernet)
— hétérogène (Internet)

Modèles d'organisation :
— Modèle Client / serveur (ex : HTTP)
— Modèle pair à pair (P2P)

17.6 Modèle(s) par couches



OSI = Open Systems Interconnection
TCP/IP = Transmission Control Protocol / Internet Protocol
Ce cours : principalement TCP/IP

17.7 Protocoles de communication

Protocole : Ensemble de règles (convention) pour que deux (ou +) entités puissent communiquer

Cela inclus :

— Format des données
— Signification des données (adresses, somme de contrôle, numéro dans une séquence...)
— "Algorithmes"

Chaque couche a son/ses protocole(s)

— Accès réseau : Ethernet...
— Réseau : IP (Internet Protocol)
— Transport : TCP, UDP
— Application : HTTP, DNS, IMAP, FTP...

17.8 Unités de communication

PDU ("Protocol Data Unit") : l'unité de base manipulé par le protocole

Unité typique :

En-tête	Message

L'en-tête dépend du protocole, et est spécifié par le protocole

Contient généralement un sous ensemble de :

— Type de "paquet"
— Taille du message
— Somme de contrôle
— Émetteur / Destinataire
— Port de destination
— Information sur le chiffrement
— Numéro de séquence...

17.9 Encapsulation

Chaque protocole d'une couche N communique via le/un protocole de la couche $N-1$.

Théoriquement, le protocole $N-1$ ne sait pas ce qu'il transporte

Par exemple le protocole TCP (couche transport) communique via le protocole IP (couche réseau)

Dans ce cas, le message TCP sera encapsulé dans le paquet IP :

En-tête IP	En-tête TCP	Message TCP

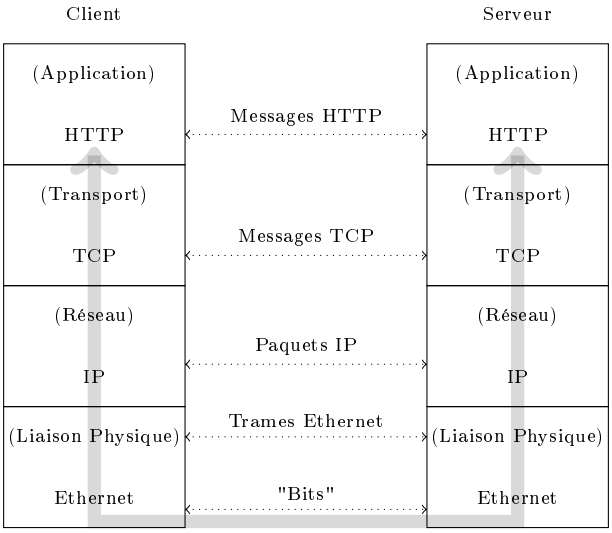
Le protocole IP communique via un protocole Ethernet :

En-tête Ethernet	En-tête IP	En-tête TCP	Message TCP

Fragmentation possible :

— Si un message est trop grand pour être transporté dans une unité du protocole $N-1$, il peut être découpé (si cela est prévu) en plusieurs messages.

17.10



17.11 Couche "Physique" et "Liaison" ("accès réseau")

Couche "Physique" :
Le médium de transport : câble en cuivre, fibre optique, ondes...
PDU : bits
— Comment sont encodés les bits ?
— Perturbations possibles

Couche "Liaison" :
— S'occupe des liaisons point à point.
— Éventuellement, transmet une somme de contrôle pour vérifier l'intégrité (Ethernet : CRC)
PDU : "Trames"
Protocoles : Ethernet, Wifi...

17.12 Couche "Réseau"

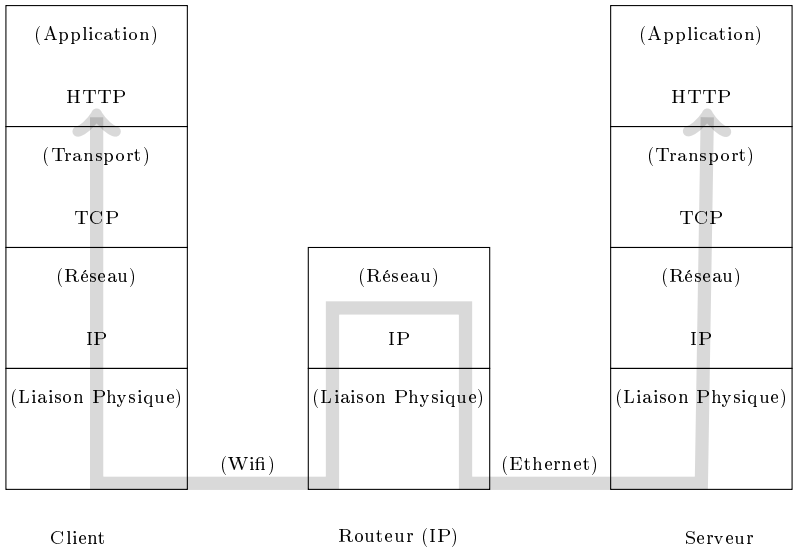
S'occupe de trouver les routes dans le réseau, de les mettre à jour, et router les paquets

Sur Internet : IP (Internet Protocol).
PDU : "Paquets IP"

Deux "variantes" d'IP :

— IPv4 : la plus utilisée actuellement, mais un nombre limité d'adresse possible 2^{32}. Arrive à saturation...
  — exemple d'adresse IPv4 : `192.168.1.79`
— IPv6 : la nouvelle norme. Partiellement en place. 2^{128} adresses
  — adresse IPv6 : `fe80::e6f8:9cff:fe67:ee22`

17.13



17.14 Couche Transport

Établir et maintenir les connexions, assurer l'arrivée, dans l'ordre des paquets...

Sur Internet : Principalement TCP et UDP.
— TCP : "Transmission Control Protocol"
  — Mode connecté
  — Fiable : détecte les erreurs, les données perdues, assure l'ordre
(Socket TCP : comme les sockets à la fin du cours sur les tubes!)

— UDP : "User Datagram Protocol"
  — Non connecté (plus léger : pas de confirmation de réception)
  — Mais : ne garantie pas la bonne livraison, ni l'ordre
Ces deux protocoles ajoutent un numéro de port : entre 1 et 65535
Généralement un numéro de port $\leftrightarrow$ une application.
HTTP : 80, SSH : 22, DNS : 53...

PDU : Segment TCP, Datagramme UDP...

17.15 Couche Application

— Web : HTTP, FTP
— mail : IMAP, POP, SMTP
— session : (Telnet), SSH
— DNS, DHCP
— ...

— Vos programmes !

17.16 Suite :

— Les couches en détail
— Programmation IP en POSIX

18 Couches Accès Réseau

18.1 Couche "Physique" : Médium

Support de transmission "guidés" :
— paire de cuivre torsadée
  — Ethernet (actuel) : 4 paires (câble de catégorie 5 "RJ45"...)
  — RTC/xDSL : 1 paire torsadée
— câble coaxial
— fibre optique
— ...
Sans fil :
— Ondes radio (wifi, bluetooth, satellites...)
— Ondes lumineuses
Sujet à des perturbations (erreurs)
— atténuation (résistance, impuretés), auto-perturbations
— perturbations extérieures

Plus que c'est long/loin, plus qu'il y a de possibilités d'erreur
— débit théorique maximum diminue avec la longueur de la liaison

Pourquoi torsadées ?
— les perturbations électromagnétiques s'annulent
Éventuellement : un blindage en plus.

18.2 Couche "Physique" : Encodage des bits

"Modulation numérique" : convertir des bits en signaux analogiques

— Transmission en bande de base

— Modulation d'un signal porteur
  — modulation de fréquence
  — modulation d'amplitude

— modulation de phase

18.3 Transmission en bande de base

Le plus simple (NRZ) :
Paire torsadée :
— Bit = 0 → tension positive
— Bit = 1 → tension négative
Fibre optique
— Bit = 0 → pas de lumière
— Bit = 1 → lumière

Problème : si trop de 0 de suite, on s'y perd.
— Codage de Manchester 0 = -+ et 1 =+- (Ethernet "classique")
— Interdire les suites trop longues de 0 ou 1 (réencoder...)

18.4 Modulation d'un signal porteur

Pourquoi :
— Multiplexage de fréquences
— Ondes électromagnétiques
— Médium fonctionnant sur une plage de fréquences
Comment :
— Modulation de fréquence (0 : f, 1 : f')
— Modulation d'amplitude
— Modulation de phase

— Possible de coder plus qu'un bit à la fois
— Possible d'associer modulation d'amplitude et de phase

18.5 Half / Full Duplex

— (Full-)Duplex : communication dans les deux sens possible en même temps
— Half-Duplex : communication dans les deux sens, mais une à la fois
— Simplex : communication dans un sens

18.6 Câble catégorie 5 (Ethernet "RJ45")

— 4 Paires torsadés
— Sert pour Ethernet
— Attention : Ethernet = une famille de normes (10BASE2, 10BASE-T, 100BASE-T, 1000BASE-T...)
(Ethernet existe aussi sur coaxial et fibre optique).
— L'utilisation des paires diffère selon la norme.
— Améliorations : Cat 5e, Cat 6...
— spécification plus strictes (résistance, capacité, inductance...)

— RJ45 : nom du connecteur (8P8C)

18.7 Couche Liaison

But :
— Interface à la couche réseau
— Contrôler et traiter les erreurs de transmission
— Réguler les flux

Deux types de liaisons :
— point à point : communication entre 2 machines
— diffusion : canal partagé entre plusieurs machines
Problèmes :

Comment découper le flux de bits en trames ?
→ fanions de signalisation de début de trame

Détecter les erreurs
— dues au perturbations extérieures
— dues aux collisions de paquets
→ utilisation de sommes de contrôle (CRC : Cyclic Redundancy Check)

Contrôle de flux → retour d'information

18.8 Liaison en mode diffusion

Et quand le médium est partagé ? (Wifi, câble commun à plus de 2 machines)

Au début d'Ethernet : un câble coaxial pour plusieurs machines.
Problèmes :
— Plus de collisions à gérer
— Destinataire de la trame

⇒ Sous-couche MAC (Medium Acces Control)

18.9 MAC Ethernet classique

Paquet Ethernet :

Préambule	MAC dest.	MAC source	Type/Longueur	Données	CRC
8	6	6	2	<=1500	4

Adresse MAC :
— 6 octets : 3 premiers pour le constructeur, les 3 derniers pour les cartes construites par le constructeur.
— Théoriquement, chaque carte réseau a une adresse MAC différente.
CRC : somme de contrôle

Trame MAC Wifi (802.11)

Contrôle	Durée	Dest.	Source	Adresse 3	Séquence	Données	CRC
2	2	6	6	6	2	<=2312	4

18.10 Ethernet commuté

Ethernet sur un câble coaxial :
— Difficile à rajouter une nouvelle machine
— Une carte réseau défaillante peut bloquer tout le réseau
— Vite encombré.

Évolutions d'Ethernet :
— Topologie en étoile : toutes les machines sont reliées à un *hub*, par une paire torsadée

Puis, pour augmenter la capacité : Inutile d'envoyer les paquets aux machines non destinatrices de la trame !
— Remplacement du *hub* par un *switch* (commutateur) : Ethernet commuté
Le switch doit garder une table adresse MAC / port.

Comment les trouver la bonne bijection ?

Apprentissage a posteriori :

Quand il reçoit une trame de source s, destinataire d par le port p :
— Il associe l'adresse MAC s au port p
— Si le port de l'adresse d n'est pas p, il diffuse la trame sur le port de d
— Si le port de l'adresse d est p, il rejette la trame
— Si il ne sait pas le port de l'adresse d, il diffuse à tout le monde

Marche aussi pour une topologie en arbre !
Les switch jouent un rôle similaire aux routeurs IP.

Pourquoi rajouter une couche "Réseau" ?

Suite :
— Couche Réseau et Transport : TCP/IP

19 Couche réseau & Protocole IP

19.1 Introduction

La couche "liaison" s'occupe des communications point à point.

La couche "réseau" s'occupe de :
— router les informations dans le réseau
— trouver et mettre à jour les routes
— détecter nouveaux liens
— détecter les pertes de liens et problèmes de congestion

Sur internet : protocoles IP
— IPv4 : le plus courant, mais on arrive à bout des adresses disponibles
— IPv6 : le nouveau, partiellement en place

19.2 Commutation de paquets vs de circuits

Il existe deux grand types de réseaux :
— Réseaux à commutation de paquets (store-and-forward)
  — Unité de base, un "paquet" : une suite d'octets
  — Le routeur reçoit un paquet, analyse à qui il est destiné, et le renvoie dans la bonne direction
— Réseaux à commutation de circuit
  — Chaque routeur prépare la route en connectant le port d'entrée au port de sortie. Puis l'information passe en flux jusqu'à la fin de la communication
  — Exemple : réseau téléphonique traditionnel

Il est aussi possible d'avoir des réseaux à paquets, où les routes sont prédéfinies à l'avance pour chaque connexion.

Internet : commutation de paquets.
Unité de base du protocole : "paquet IP"

19.3 Routage de paquets

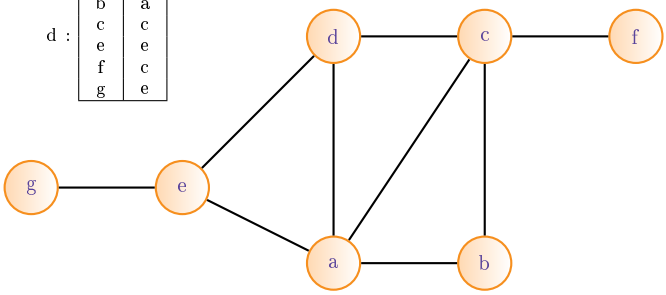
Chaque noeud interne du réseau (noeud avec au moins 2 voisins) doit choisir, quand il reçoit un paquet, à qui il doit le transférer. ("Router un paquet").

Un noeud interne est souvent appelé un routeur (ordinateur ou matériel spécialisé)

Il possède pour cela une table de routage :
— une table de paires (adresse, voisin)

d :

a	a
b	a
c	c
e	e
f	c
g	e

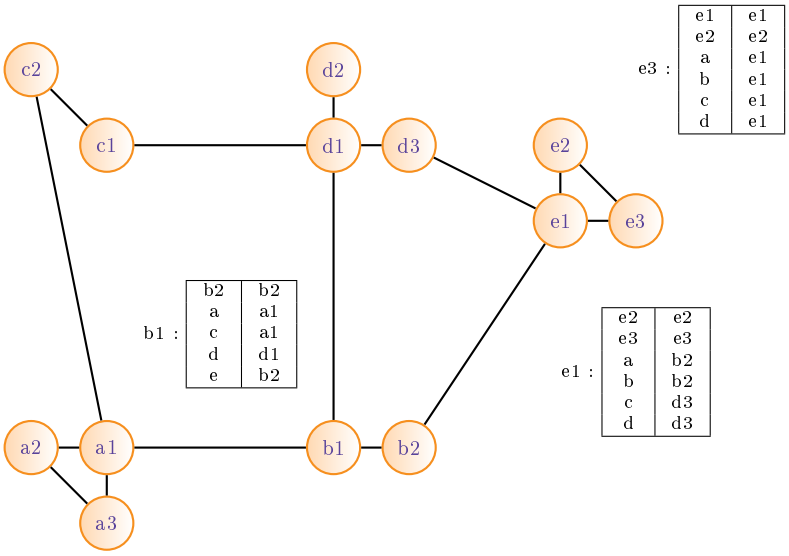


19.4 Routage hiérarchique

Problème : la table a autant d'entrées que de noeuds dans le graphe.
Internet IPv4 : $\sim 2^{32} = \sim 4$ milliard...

Solution : Routage hiérarchique
Chaque le réseau est divisé en sous réseaux :
— les adresses sont hiérarchiques, du type "sous-reseau.machine"

— chaque sous réseau S sait comment router ses paquets
— en dehors du sous-réseau S, tous les paquets vers une machine de S est routé vers le même routeur, un point d'entrée de S.



19.5 Trouver les tables

Les tables de routage peuvent être définies statiquement (routage statique), ou dynamiquement

— Statique : L'administrateur de la machine/routeur définit les entrées de la table.
Cas habituel pour vos machines.
Problème : ingérable pour les routeurs internes
— Dynamique : Le routeur construit lui même sa table de routage, en partageant des informations avec ses voisins

Comment un routeur peut faire pour remplir/compléter/mettre à jour sa table de routage ?

19.6 Routage par inondation

Découverte de routes par inondation :
Si on ne connaît pas la direction pour un noeud, on peut faire une diffusion générale ("broadcast") d'un paquet demandant où se trouve le noeud
Chaque routeur qui reçoit le paquet de demande : soit
— répond si il sait, ou
— renvoie le paquet à tous ses autres voisins
Problème : lourd

Pour éviter de trop encombrer le réseau :

— Chaque requête possède un numéro. On garde mémoire des requêtes déjà renvoyées, pour ne pas le faire une deuxième fois
— Pour chaque demande non satisfaite, on attend un petit moment avant de la renvoyer. Si on reçoit la même demande entre temps d'un autre voisin, on ne la lui retransmettra pas.

19.7 Plus court chemin

Si un routeur connaît la topologie du réseau, il peut effectuer un algorithme du plus court chemin (comme l'algorithme de Dijkstra)

⇒ plus court chemin dans un graphe

Avantage : poids sur les liens (distance, prix...)

Problème : il faut connaître toute la topologie du réseau

19.8 Vecteur de distance (Bellman-Ford)

— Chaque routeur connaît la distance vers ses voisins. (d_i)
— Distance : nombre de sauts, délai de propagation...
— Chaque routeur maintient (en plus de sa table de routage) une liste de distance estimée à tous les noeuds du réseau ("vecteur")
— Chaque routeur envoie régulièrement à tous ses voisins ce vecteur
— Le routeur met à jour sa table de routage : pour chaque noeud x dont il a connaissance, il route le paquet vers le voisin i qui minimise $d_i + V_i[x]$. (Et met à jour son vecteur de distance en même temps)

Problème : si une route disparaît, l'information mettra du temps à être corrigée... (problème de la valeur infinie)

19.9 État de lien

Routage par informations d'état de lien :
— Chaque routeur construit un paquet avec l'ensemble de ses voisins, et la distance
— Le paquet est broadcasté sur tout le réseau (avec un numéro de séquence)
— Chaque routeur connaît donc l'ensemble des noeuds et leurs voisins, et peut reconstruire le graphe
— Les routes sont décidées grâce à un algorithme comme Dijkstra

19.10 Contrôle de la congestion

En cas de congestion, il peut y avoir l'effondrement du débit du réseau :
— les émetteurs retransmettent les paquets perdus (ou trop retardés), qui seront à nouveau perdus...

Lors d'une congestion, que faire ?

— augmenter la capacité du lien
— rerouter sur une route moins encombrée
— avertir les sources
— contrôle d'admission
— éliminer des paquets...

19.11 Qualité de service (QoS)

Toutes les applications utilisant le réseau n'ont pas les mêmes besoins. Par exemple :

— Visio-conf : demande haute en délai, faible en fiabilité, haute en bande passante
— Mail : demande faible en délai, haute en fiabilité, faible en bande passante

Les algorithmes de routage peuvent considérer plusieurs classes de paquets
Par exemple : on préfère perdre des paquets plutôt que les faire attendre lors des congestions.

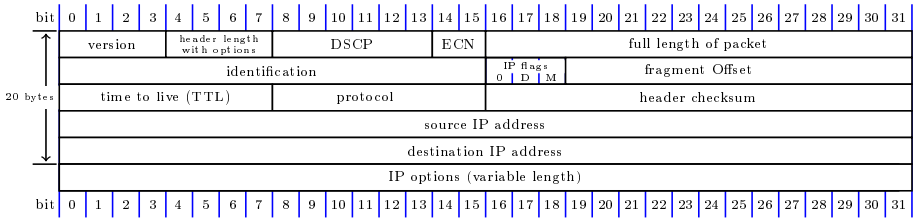
19.12 Internet Protocol

Protocole réseau utilisé sur Internet : IP (Internet Protocol). Deux versions :

— IPv4 : le plus utilisé actuellement, mais un nombre d'adresses limité ($< 2^{32}$).
— IPv6 : la nouvelle norme, partiellement en place

Routage de paquets, store-and-forward

19.13 Entête IPv4



— Version = 4
— DSCP (Differentiated Services Code Point) : Classe du paquet (QoS)
— ECN (Explicit Congestion Notification)
— Identification / D / M / Framgent Offset : Quand un paquet est fragmenté, tous les fragments qu'un paquet contiennent la même identification. Fragment Offset continent la position du fragment. (D : Don't fragment. M : More fragment)
— TTL : décrémenté à chaque saut (retransmission par un routeur). Quand il atteint 0, le paquet est éliminé (et un message ICMP est envoyé à la source)
— Protocol : TCP=6, UDP=17, ICMP=1...
— Options : Routage strict, enregistrement de la route...

Attention : Big endian !

19.14 Adresses IPv4

Une adresse IPv4 = 32 bits

— c-à-d 4 octets, ou
— 4 entiers de 0 à 255.
Notation a.b.c.d. Ex : `192.168.1.1`

ICANN (Internet Corporation for Assigned Names and Numbers) fournit les adresses

Organisées hiérarchiquement en sous-réseaux

19.15 Sous réseaux IPv4

Un sous réseau possède une adresse IP i, et un masque m de sous-réseau

Toutes les adresses IP j telles que $j \& m = i$ font parties du sous-réseau.

Masque (ou netmask) : en binaire : suite de k '1', puis de $32-k$ '0'
Notation : ip/k :
Exemple : `192.168.1.0/24` signifie que :
— le masque est `255.255.255.0`
— le sous réseau va de `192.168.1.0` à `192.168.1.255`

19.16 Sous réseaux IPv4 : Exemple

— Machine (hôte) d'adresse IP : `147.222.23.42`
— Sur le réseau : `147.222.16.0/20`
— ⇒ Masque réseau : 255.255.240.0

Adresse hôte	147								222								23								42							
Adresse hôte	1	0	0	1	0	0	1	1	1	1	0	1	1	1	1	0	0	0	0	1	0	1	1	1	0	0	1	0	1	0	1	0
Masque	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	0	0	0	0	0	0	0	0	0	0	0	0
Adresse réseau	1	0	0	1	0	0	1	1	1	1	0	1	1	1	1	0	0	0	0	1	0	0	0	0	0	0	0	0	0	0	0	0
Adresse réseau	147								222								16								0							
	Champ sous-réseau																								Champ hôte							

19.17 Routage simple

Pour voir/configurer les interfaces réseaux et les adresses IP/masque : `ifconfig`.
Routage statique via une passerelle (cas simple de routage)
— Tous les paquets pour les adresses IP dans le réseau sont envoyé directement au destinataire via Ethernet (ou Wifi, ou la couche liaison du réseau)
— Tous les autres paquets sont envoyés à la passerelle (l'adresse IP de l'interface du routeur qui est connecté au reste d'Internet)

Pour voir/configurer les routes sur Linux : `route`

```
Table de routage IP du noyau
Destination   Passerelle   Genmask          ... Iface
0.0.0.0.      140.77.12.1  0.0.0.0              eth0
140.77.12.0   0.0.0.0      255.255.254.0        eth0
169.254.0.0   0.0.0.0      255.255.0.0          eth0
```

(Règle du "plus grand préfixe commun")

19.18 Adresses réservées

Certaines plages d'adresses sont réservées :
— 127.0.0.0/8 : Bouclage interne
— 255.255.255.255/32 : Broadcast local
— 10.0.0.0/8, 172.16.0.0/12, 196.168.0.0/16 : Adresses privées (NAT)
— 169.254.0.0/16 : lien local
— 224.0.0.0/4 : multicast
— 240.0.0.0/4 : réservé pour une utilisation future...
— ...

19.19 NAT (Network Address Translation)

Pour éviter de gaspiller les adresses et d'avoir à demander à l'ICANN des adresses pour les machines personnelles ou qui ne nécessitent pas une adresse IP visible de l'extérieur (pas de serveur)

— Assigner à un sous-réseau local une seule adresse IP (`addr`) pour internet

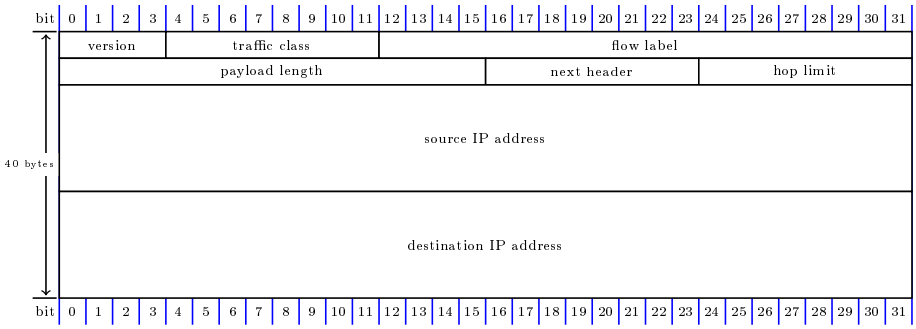
Principe de NAT :
— Les machines à l'intérieur du réseau local ont une adresse IP en `192.168.0.0/16`, `10.0.0.0/8` ou `172.16.0.0/12`.
— Quand elles veulent envoyer un paquet à Internet, elles passent par une passerelle, qui est connectée à Internet via l'adresse `addr`
— La passerelle remplace dans le paquet IP l'adresse du réseau local en `addr` avant d'envoyer le paquet sur Internet
— Quand la passerelle reçoit un paquet depuis internet, elle analyse les entêtes des paquets TCP et UDP, et regarde le port utilisé pour retrouver l'adresse du destinataire dans le réseau.

19.20 IPv6

Problème d'IPv4 :
— Seulement 2^{32} adresses
— Tables de routage trop longues, dû à la fragmentation en sous-réseaux trop petits
Solution : IPv6

Changements entre IPv4 et IPv6 :
— Adresses sur 128 bits
— Simplification de l'en-tête
— Suppression de la somme de contrôle
— Suppression de la fragmentation des paquets en cours de route
— IPsec (Internet Protocol Security)
— ...

19.21 Entête IPv6



— version = 6
— traffic class = DSCP/ECN de IPv4
— flow label : identification du flux (réservation / QoS)
— hop limit : le nouveau nom du "time to live"
— next header : soit le contenu du prochain entête facultatif (option), soit le protocole
— options : informations sur la fragmentation, routage, chiffrement...

19.22 Adresse IPv6

8 blocs de 16 octets.
Les blocs sont écrits en hexadécimal, séparés par `":"`
— On peut omettre les 0 de début de blocs
— Un ou plusieurs blocs consécutifs à 0 peuvent être remplacés par `"::"`

Exemple :
`fe80:0000:0000:0000:028d:99ff:fec1:0078`=
`fe80::28d:99ff:fec1:78`

19.23 Protocoles de gestion

La couche IP contient d'autres protocoles (de gestion) :
— ICMP : messages de contrôle IPv4
— ICMPv6 : messages de contrôle IPv6
— ARP/RARP : résolution d'adresse
— DHCP : configuration dynamique
— IGMP
— ...

19.24 ICMP

ICMP = Internet Control Message Protocol
Paquets ICMP :
— destination inaccessible
— délai expiré

— demande/envoi d'écho (`ping`)
— problème d'en-tête
— ...

Commandes utilisant les messages ICMP : `ping`, `traceroute`

19.25 ARP

ARP = Address Resolution Protocol

Permet, dans un sous-réseau, de trouver la correspondance entre l'adresse IP d'une interface et son adresse MAC. Principe :
— Quand un noeud ne connaît pas l'adresse MAC associée à une adresse IP, il envoie un paquet "broadcast" (à tout le monde) demandant :
— "Je suis (adresse IP)/(adresse MAC). À qui appartient (adresse IP) ?"
— L'ordinateur possédant l'adresse IP répond.

— RARP (Reverse ARP) : demande l'adresse IP à partir de l'adresse MAC

Commande : `arp`

19.26 DHCP

DHCP = Dynamic Host Configuration Protocol

Un noeud sur un réseau peut demander, via ce protocole, une adresse IP et la configuration du réseau (Masque, adresse IP de la passerelle, serveurs DNS...)

— Il envoie un message broadcast
— Un serveur DHCP (normalement, un serveur pour tout le sous-réseau) se charge d'assigner les adresses, et de répondre.

Commandes : `dhcpd` / `dhclient`

19.27 Algorithmes de routage sur Internet

Internet est un réseau de réseaux. Il n'y a pas un unique algorithme de routage. Chaque sous-réseau peut utiliser le sien.
Plusieurs algorithmes existent.

Pour les "Système Autonome" (FAI, RENATER...)
— RIP (Routing Information Protocol) : vecteurs de distance
— IGRP (Interior Gateway Routing Protocol) : vecteurs de distance
— OSPF (Open Shortest Path First) : état de liens
— IS-IS (Intermediate system to intermediate system) : état de liens

Entre système autonomes : des considérations économiques et politiques s'ajoutent. Par exemple : certains liens sont payants.
BRG (Border Gateway Protocol) : Vecteurs de chemins