Antisèche :

NAME open -open a file
SYNOPSIS int open(const char *pathname, int flags);
DESCRIPTION
The open() system call opens the file specified by pathname. The return value of open() is a file descriptor, a small, nonnegative integer that is used in subsequent system calls (read(), write(), etc.) to refer to the open file. The argument flags must include one of the following access modes: O_RDONLY, O_WRONLY, or O_RDWR. These request opening the file read-only, write-only, or read/write, respectively.
RETURN VALUE
open() returns the new file descriptor, or -1 if an error occurred (in which case, errno is set appropriately).

==========

NAME read -read from a file descriptor
SYNOPSIS ssize_t read(int fd, void *buf, size_t count);
DESCRIPTION
read() attempts to read up to count bytes from file descriptor fd into the buffer starting at buf.
RETURN VALUE
On success, the number of bytes read is returned (zero indicates end of file), and the file position is advanced by this number. It is not an error if this number is smaller than the number of bytes requested; this may happen for example because fewer bytes are actually available right now (maybe because we were close to end-of-file, or because we are reading from a pipe, or from a terminal), or because read() was interrupted by a signal. On error, -1 is returned, and errno is set appropriately.

==========

NAME write -write to a file descriptor
SYNOPSIS ssize_t write(int fd, const void *buf, size_t count);
DESCRIPTION
write() writes up to count bytes from the buffer starting at buf to the file referred to by the file descriptor fd.
RETURN VALUE
On success, the number of bytes written is returned (zero indicates nothing was written). It is not an error if this number is smaller than the number of bytes requested; this may happen for example because the disk device was filled. On error, -1 is returned, and errno is set appropriately.

==========

NAME fork -create a child process
SYNOPSIS pid_t fork(void);
DESCRIPTION
fork() creates a new process by duplicating the calling process. The new process is referred to as the child process. The calling process is referred to as the parent process.
RETURN VALUE
On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, no child process is created, and errno is set appropriately.

==========

NAME waitpid -wait for process to change state
SYNOPSIS pid_t waitpid(pid_t pid, int *wstatus, int options);
DESCRIPTION
waitpid() is used to wait for state changes in a child of the calling process. A state change is considered to be: the child terminated; the child was stopped by a signal; or the child was resumed by a signal. In the case of a terminated child, performing a wait allows the system to release the resources associated with the child; if a wait is not performed, then the terminated child remains in a "zombie" state.

If pid $>$ 0, then the call will wait for the children whose PID equals pid.

If wstatus is not NULL, then waitpid() stores status in informations in the int it points to. If wstatus is NULL, then this parameter is ignored. The value of options is an OR of zero or more of the following constants : WNOHANG, WUNTRACED, WCONTINUED.
RETURN VALUE
On success, waitpid() returns the process ID of the child whose state has changed; if WNOHANG was specified and one or more child(ren) specified by pid exist, but have not yet changed state, then 0 is returned. On error, -1 is returned.

==========

NAME getpid, getppid -get process identification
SYNOPSIS pid_t getpid(void);
pid_t getppid(void);
DESCRIPTION
getpid() returns the process ID (PID) of the calling process. getppid() returns the process ID of the parent of the calling process. This will be either the ID of the process that created this process using fork(), or, if that process has already terminated, the ID of the process to which this process has been reparented.
ERRORS
These functions are always successful.

==========

NAME pipe -create pipe
SYNOPSIS int pipe(int pipefd[2]);
DESCRIPTION
pipe() creates a pipe, a unidirectional data channel that can be used for interprocess communication. The array pipefd is used to return two file descriptors referring to the ends of the pipe. pipefd[0] refers to the read end of the pipe. pipefd[1] refers to the write end of the pipe. Data written to the write end of the pipe is buffered by the kernel until it is read from the read end of the pipe.

If all file descriptors referring to the write end of a pipe have been closed, then an attempt to read from the pipe will see end-of- file and will return 0. If all file descriptors referring to the read end of a pipe have been closed, then a write will cause a SIGPIPE signal to be generated for the calling process. If the calling process is ignoring this signal, then write fails with the error EPIPE. An application that uses pipe and fork should use suitable close calls to close unnecessary duplicate file descriptors; this ensures that end-of- file and SIGPIPE/EPIPE are delivered when appropriate.
RETURN VALUE
On success, zero is returned. On error, -1 is returned, and errno is set appropriately.

==========

NAME sigaction -examine and change a signal action
SYNOPSIS int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact);
DESCRIPTION
The sigaction() system call is used to change the action taken by a process on receipt of a specific signal. signum specifies the signal and can be any valid signal except SIGKILL and SIGSTOP. If act is non-NULL, the new action for signal signum is installed from act. If oldact is non-NULL, the previous action is saved in oldact. The sigaction structure is defined as something like:

```
struct sigaction {
  void     (*sa_handler)(int);
  ... ;
};
```

sa_handler specifies the action to be associated with signum and may be SIG_DFL for the default action, SIG_IGN to ignore this signal, or a pointer to a signal handling function. This function receives the signal number as its only argument.
RETURN VALUE
sigaction() returns 0 on success; on error, -1 is returned, and errno is set to indicate the error.

ASR5 - Système d'Exploitation

Introduction

Nicolas Louvet

Univ. Claude Bernard Lyon 1

Séance 1

Les responsables de l'UE : Nicolas Louvet et Laure Gonnord.

Ce support de cours est largement repompé de celui de Fabien Rico.

Plan

Introduction

Qu'est-ce qu'un système d'exploitation ?

Littéralement, *ce qui permet d'utiliser la machine.* Quatre grands rôles :

- Interface entre applications et matériel (*e.g.*, gestion des periphériques)
- Organisation (*e.g.*, des disques, de la mémoire, et des processus)
- Sécurité (*e.g.*, des données, du matériel)
- Interaction avec le ou les utilisateurs (*e.g.*, comptes, droits, installation)

Différents type de systèmes d'exploitations, pour différents usages :

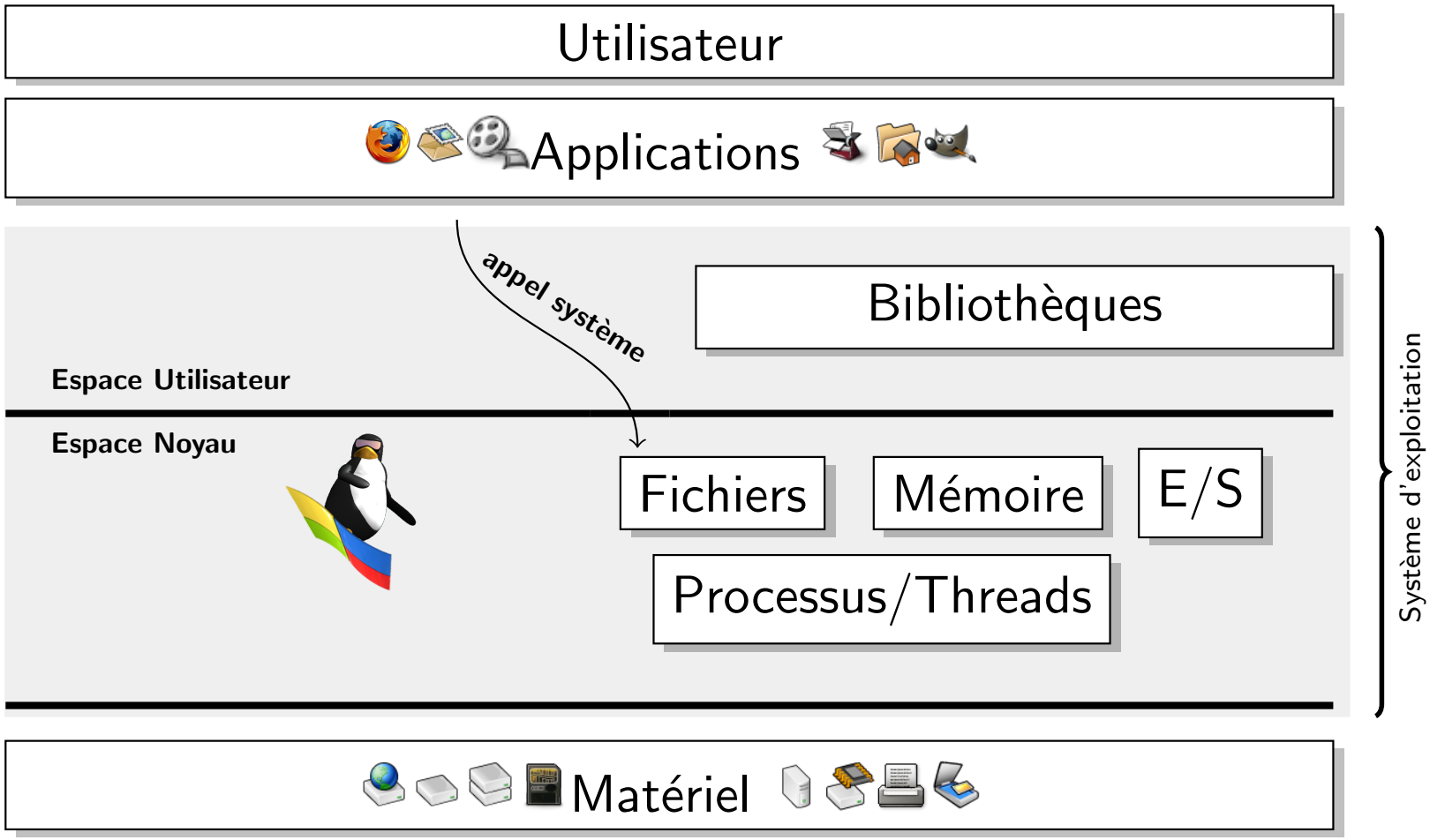
- Système « généralistes », multi-utilisateurs et multi-tâches : GNU-Linux (Debian, Ubuntu, Redhat...), Windows, Mac OS X...
- Pour l'embarqué (contraintes sur l'utilisation des ressources) : Windows Embedded Compact, Android...
- Pour le temps-réel (contraintes d'échéance) : RTLinux, RTAI.

Interface avec le matériel

Que se passe-t-il lorsque l'on branche une clé USB ?

- La clé est traitée comme un disque amovible ;
- on peut le formater, lire et écrire des fichiers ;
- pourtant une clé USB n'est pas vraiment un disque dur, ni un périphérique SCSI (*Small Computer System Interface*) !

Le seul élément courant que l'utilisateur manipule :
possible installation d'un driver (en tant qu'administrateur)


Utilisateur
Applications
appel système
Bibliothèques
Espace Utilisateur
Espace Noyau
Fichiers
Mémoire
E/S
Processus/Threads
Système d'exploitation
Matériel

Fonctionnalités

En tant qu'interface, un système d'exploitation doit fournir :

- à l'utilisateur/programmeur une « machine virtuelle », avec
  - une vue unifiée du matériel (mémoire, disque, carte réseau, ...)
  - des objets abstraits (fichiers, répertoires, processus, threads, ...)
- au matériel
  - gestion des ressources (conflit d'accès, ordonnancement),
  - protection contre la mauvaise utilisation,
  - une gestion des évènements (interruptions, exceptions).

Cela impose des vérifications, des attentes et des accès indirects :

- Au moins 2 modes de fonctionnement :
  - *mode utilisateur* (exécution par défaut, sans accès aux ressources)
  - *mode noyau* (accès direct aux ressources, exécution protégée)
- Un moyen de passer de l'un à l'autre : les *appels systèmes*.
- Un *mécanisme d'interruption* et de mise en attente (files de priorités) des évènements asynchrones

Un appel système est :

- une fonction fournie par le système,
- que tout programme peut utiliser,
- qui est exécutée en mode noyau.

Ce sont des ponts entre le mode utilisateur et le mode noyau.

Par exemple, pour lire, la fonction C `scanf` utilise l'appel système `read`.
En C++, l'opérateur << utilise l'appel système `write` pour écrire.

Les appels système :

- font des vérifications,
- sont toujours susceptibles de générer une erreur,
- prennent du temps !

Organisation

Les ressources proposées par la machine :

- au moins un processeurs,
- de la mémoire vive,
- de la mémoire de masse,
- des périphériques d'entrées/sorties.

Les besoins des utilisateurs ou des programmes :

- Accès aux ressources : nécessite d'un arbitrage.
  ex : si plusieurs programmes veulent écrire sur le disque ?
- Organisation des données.
  ex : comment retrouver les données rapidement ?
- Gestion des évènements.
  ex : que faire si l'utilisateur insére une clé usb ?

Gérer les ressources demandées par les programmes :

- Que se passe-t-il lorsque deux programmes / *processus* demandent la même chose en même temps ?
- Il faut arbitrer, et se rappeler quel processus à obtenu quoi, maintenir une liste de demandes, pour assurer un accès équitable aux ressources.

Ressources les plus importantes : le(s) processeur(s) et la mémoire

Le noyau doit décider

- quelle tâche devient active : c'est l'*ordonnancement* ;
- quelle données/programmes sont présents en mémoire vive : c'est le *swapping* (ou *va-et-vient*).

Pour gérer l'ordonnancement, le noyau doit reprendre la main : quand ?

Cela a généralement lieu à chaque passage en mode noyau :

- Lors de la gestion des exceptions ; par exemple :
  - division par zéro,
  - accès mémoire non autorisé (*segmentation fault*),
  - instruction interdite (*illegal instruction*).
- Lors d'une interruption matérielle (IRQ, *interrupt request*) :
  - en provenance d'un périphérique
  - du timer (quantum de temps)
- Lors des interruptions logicielles via les appels système
  - à chaque fois que le programmeur fait des entrées/sorties par exemple,
  - cela explique aussi le coût des appels systèmes !

↪ Quand vous écrivez du texte, le système en profite pour travailler !

Sécurité

Que se passe-t-il si :

1. on fait tellement de calculs que le processeur dépasse 100 degrés ?
2. on interrompt une écriture de disque brutalement ?
3. un code se met à écrire n'importe-où en mémoire ?
4. Vous essayez de lire le répertoire d'un autre utilisateur ?

Puisque le système est une interface entre les applications et le matériel, il a aussi un rôle de protection :

- du matériel
  - monitoring *e.g.*, `cat /sys/class/thermal/thermal_zone*/temp`
  - actions automatiques *e.g.*, gestion de l'énergie
  - zones critiques
- des données
  - systèmes de fichiers journalisés (protection contre l'arrêt brutal)
  - utilisateur, droits, authentification
- des programmes
  - séparation des tâches
  - virtualisation
  - communication

Interface avec l'utilisateur

Que voit l'utilisateur ?

- Le logo au démarrage, et quelques bizarreries :

- Un système de configuration et d'installation.

Pour les utilisateurs, le système doit :

- imposer des limites (*e.g.*, droits d'accès) ;
- permettre la programmation « avancée » (*e.g.*, processus, client/serveur) ;
- fournir des outils pour administrer la machine ;

Le système doit (suite) :

- différencier les utilisateurs avec
  - une base de données des utilisateurs,
  - un système d'authentification ;
- être configurable :
  - base de données des configurations,
  - interface de configuration ;
- avoir un système d'installation de programmes :
  - comment installer ?
  - notion de packages.

Résumé

Le système d'exploitation peut être abordé selon 3 points de vues :

1. **Utilisation et programmation :** les outils fournis pour mieux utiliser les possibilités des ordinateurs (programmation multiprocessus, multithread, réseau par exemple)
2. **Conception et théorie :** les problèmes posés par les systèmes et les moyens de les résoudre.
3. **Administration :** comment configurer et gérer le système ?

On va essayer dans l'UE d'aborder chacun de ces points de vue.

À retenir dans cette section

- rôles du système d'exploitation,
- notion de mode d'exécution (utilisateur ou noyau),
- les appels systèmes et leur rôle.

Plan

Point sur les ≪ prérequis ≫ de l'UE

- Avoir des notions de base : architecture des ordinateurs, et réseaux.
- ≪ Se débrouiller ≫ sur un système GNU/Linux
  - ↪ ne pas avoir peur de la ligne de commande !
- ≪ Se débrouiller ≫ pour programmer en C/C++
  - ↪ ne pas hésiter à se documenter, via des sources fiables,
  - ↪ bien comprendre un code que l'on veut réutiliser.
- Compilation d'un code
  - ↪ savoir compiler en ligne de commande (`g++` ou `clang++`),
  - ↪ comprendre les étapes de la compilation,
  - ↪ savoir écrire un `Makefile` simple.

C'est une UE technique : il faut comprend des concepts et aussi pratiquer !

Remarques sur la programmation C/C++

- Vous avez des déjà appris beaucoup de choses, en LIFAP1, LIFAP3, peut-être en LIFAP4 : il faut continuer !
- En 2019, on peut se permettre d'utiliser certains apports du C++11 : développé à partir de 2003, standard ISO/CEI 14882 publié en 2011.
- Apport intéressant du C++11, la *Standard Template Library* (STL) :
  - fournit des conteneurs sous la forme de class templates,
  - un site de référence, avec les classes et méthodes de la STL : `https://en.cppreference.com/w/`
- Le but n'est pas de faire un cours sur le C++, juste de donner des points d'entrée... Dans la suite :
  - un exemple avec `vector<T>` (tableaux dynamiques),
  - un autre avec `list<T>` (listes doublement chainées),
  - un dernier avec `string` (remplacement des tableaux de `char`).

```cpp
#include <iostream>
#include <vector>

int main(void) {
  std::vector<int> tab;

  std::cout << "Entrez des entiers (< 0 pour arrêter) :"
            << std::endl;
  while(true) {
    int n;
    std::cin >> n;
    std::cout << "lu : " << n << std::endl;
    if(n < 0) break;
    tab.push_back(n);
  }

  std::cout << "Vos " << tab.size() << " nombres :"
            << std::endl;
  for(unsigned int i = 0; i < tab.size(); i++)
    std::cout << tab[i] << " ";

  return 0;
}
```

```
#include <iostream>
#include <list>

using namespace std;

int main(void) {
  list<float> l;

  cout << "Entrez des flottants (< 0 pour arrêter) :" << endl;
  while(true) {
    float x;
    cin >> x;
    cout << "lu : " << x << endl;
    if(x < 0) break;
    l.push_front(x);
  }
  cout << "Vos " << l.size() << " nombres :" << endl;
  while(!l.empty()) {
    cout << l.front() << " ";
    l.pop_front();
  }
  cout << endl;
  return 0;
}
```

La classe `string` permet de se passer des tableaux de `char` (`char[]` ou `char*`) pour la gestion des chaînes de caractères :

```
#include <iostream>
#include <string>
using namespace std;
int main(void) {
  string ch;
  cout << "Entrez une chaîne de caractères : ";
  getline(cin, ch);
  cout << "Vous avez entré : " << ch << endl;
  return 0;
}
```

On peut avoir besoin d'accéder au tableau de caractères sous-jacent :

```
  string ch;
  cout << "Entrez une chaîne de caractères : ";
  getline(cin, ch);
  cout << "Vous avez entré : " << flush;
  write(STDOUT_FILENO, ch.data(), ch.length());
```

Problème avec les *templates* : ils compliquent les messages d'erreur...

Par exemple :

```
string chaine = "coucou";
cout << chaine + 4 << endl;
```

Donne de copieux messages d'injures :

```
template.cpp:9:18: erreur : no match for « operator+ »
   (operand types are « std::__cxx11::string
   {aka std::__cxx11:basic_string<char>} » and « int »)
   ...
   /usr/include/c++/6.3.1/bits/stl_iterator.h:341:5:
   note : candidate: template ...
   ...
```

Le compilateur tente de nous aider : à la ligne 9 du fichier, il ne trouve pas d'opération + entre un `string` et un `int`.

Tout le même, vous avez beaucoup moins de chance d'introduire des bugs qu'en reprogrammant les structures de données de base !

Compilation

La compilation est le fait de traduire votre code pour en faire un programme éxécutable.

Schématiquement, elle comporte 4 phases :

- *précompilation* : utilisation de directives pour modifier le source.
- *compilation* : traduction du code en langage d'assemblage.
- *assemblage* : traduction du code en langage machine.
- *édition de liens* : résolution des liens avec les fonctions utilisées.

À chaque phase il y a des options et des erreurs différentes...

Un compilateur comme g++ combine ces quatre phases, donc il n'est pas toujours facile de savoir lors de quelle phase se produit une erreur...
Mais il faut le comprendre pour pouvoir corriger !

Supposons que l'on veuille compiler `prog.cpp` en un exécutable `prog`.

Précompilation. Le compilateur modifie le code source d'après les ordres donnés par les directives :

- `#include <fichier>` = recopie le fichier ici ;
- `#define NOM VAL` dans la suite remplace `NOM` par `VAL`
- `#ifdef NOM ... #endif` = ne conserve le code que si `NOM` est défini.

Pour arrêter après le préprocessing : `g++ -E prog.cpp -o prog_tmp.cpp`

Compilation. Le compilateur traduit le source en langage langage machine, et lorsqu'il détecte un problème, il génère un message d'erreur :

- les messages sont là *pour vous aider* ;
- corriger une erreur suppose *un choix éclairé du programmeur* ;
- si un algorithme était capable de corriger, il le ferait !

Pour arrêter après la compilation : `g++ -S prog_tmp.cpp -o prog_tmp.s`

Assemblage. L'assembleur transforme le code en langage d'assemblage en un code en langage machine, appelé *code objet, dans lequel les adresses des fonctions ne sont pas résolues.*

Pour arrêter après l'assemblage : `g++ -c prog.cpp` ⇝ `prog.o`

Edition des liens. L'éditeur de lien, *ld* en l'occurence, se débrouille pour retrouver chaque fonction appelée, et place son adresse dans le code objet.

Quand vous voyez apparaître un message d'erreur en provenance de *ld*, c'est souvent que l'éditeur de lien n'arrive pas à trouver une fonction : il faut indiquer dans quel code objet ou bibliothèque se trouve la fonction.

Par défaut, le compilateur s'arrête après l'édition des liens :

```
g++ prog.cpp -o prog
```

Exemple. Dans `tst.cpp`, on veut utiliser une fonction `void msg(void)` :

```
int main(void) {
  msg();
  ...
```

`g++ -o tst tst.cpp` donne une erreur *de compilation* :

```
tst.cpp: In function 'int main()':
tst.cpp:4:3: error: 'msg' was not declared in this scope
```

Du coup, on modifie le source on ajoutant une déclaration de la fonction :

```
void msg(void); // déclaration
int main(void) { ...
```

`g++ -o tst tst.cpp` râle maintenant *lors de l'édition des liens* :

```
/tmp/ccL8xKgc.o : Dans la fonction "main"
tst.cpp:(.text+0x5) : référence indéfinie vers "msg()"
collect2: error: ld returned 1 exit status
```

En fait, le code objet de `msg()` se retrouve dans `utils.o` :
on peut compiler avec `g++ -o tst tst.cpp utils.o`

Exemple. On peut indiquer à l'éditeur de liens d'aller chercher du code dans des fichiers objets et dans des bibliothèques. Par exemple :

```
g++ -o prog main.o bilioprof.o -L/opt/lib/ -lpthread -llibopt
```

Cela signifie : ≪ fabrique moi s'il te plaît l'exécutable `prog` :

- en prenant le code objet des fonctions dans `main.o` et `bilioprof.o`,
- mais aussi dans les bibliothèques `libpthread.so` et `libopt.so`,
- qui sont dans le répertoire `/opt/lib` ou dans un répertoire habituel. ≫

Les ≪ endroits habituels ≫ dépendent de la configuration de votre système (`ldconfig`, `/etc/ld.so.conf...`).

Certaines bibliothèques sont parcourues d'office par l'éditeur de liens ; c'est le cas de la bibliothèque C++ standard utilisée par `g++` ; sur mon ordi :
`/usr/lib/x86_64-linux-gnu/libstdc++.so.6.0.25`

En résumé.

Une option utile du *préprocesseur* :

- `-D NOM` pour définir le nom dans le source ;

Options utiles du *compilateur* :

- `-g` rendre possible le *débogage* ;
- `-std=c++11` prévenir que le code est du C++11 ;
- `-Wall` demander le plus possible de messages d'aide ;
- `-c` s'arrêter à la compilation pour faire un fichier objet.

Options utiles pour l'*édition des liens (ld)* :

- `-LREP` : ajoute un répertoire où chercher des fichiers ;
- `-lNOM` : nom d'une biliothèque à utiliser (`libNOM.so` typiquement)
- `NOM.o` : fichier objet à utiliser.

De toute façon, il faut pratiquer !

Plan

Nous avons :

- passé en revue les quatre grands rôle d'un système d'expoloitation : interface, organisation, sécurité, interaction avec les utilisateurs.
- introduit des techniques de programmation C++ (STL),
- fait des rappels sur le compilation d'un programme C/C++.

Buts de l'UE :

- théorie $\rightsquigarrow$ présenter les concepts de base des sytèmes d'exploitation,
- programmation $\rightsquigarrow$ vous familiariser avec la prog. POSIX en C,
- administration $\rightsquigarrow$vous rendre plus autonomes sous GNU/Linux.

Que faire pour réussir ?

- pratiquer $\rightsquigarrow$ programmer les exemples des CM et TD, terminer les TP,
- se documenter $\rightsquigarrow$ sur le web par exemple, pages de manuel...
- progresser en « débrouillardise » $\rightsquigarrow$ c'est important aussi !

Fichiers

Nicolas Louvet

Univ. Claude Bernard Lyon 1

séance 2

Les responsables de l'UE : Nicolas Louvet et Laure Gonnord.

Ce support de cours est largement repompé de celui de Fabien Rico.

Introduction

Outre le stockage des données et des programmes, le système doit fournir plusieurs propriétés aux fichiers :

- indépendance vis-à-vis des médias de stockage,
- gestion automatique de l'espace disponible,
- permettre un accès le plus rapide possible aux données,
- protection des données contre l'accès concurrent (verrous),
- notion d'ouverture et fermeture de fichiers pour gérer les ressources (ex : démontage interdit d'une clé usb),
- protection des données privées (droits),
- en plus, archivage, sauvegarde, journalisation...

Plan

Plan

Types de fichiers

Sous Unix, de façon à uniformiser les traitements, « tout est fichier ».

On quand même quatre *types* :

- *Fichiers « réguliers » :*
  données ou programmes des utilisateurs.
- *Répertoires :*
  fichiers contenant une liste d'autres fichiers et permettant d'organiser l'ensemble des fichiers du système sur un mode hiérarchique.
- *Fichiers spéciaux de type caractères :*
  - périphériques d'entré/sortie de type caractère (terminaux),
  - fichiers de communication (pipes ou sockets).
- *Fichiers spéciaux de type bloc :*
  périphériques d'entrée/sortie accessibles par blocs (disques durs).

Les *fichiers réguliers* ont différentes utilisations (exécutable, fichier texte, image, ...) ; on parle aussi de *type* pour bien embrouiller tout le monde.

- *Typage fort* : le fichier a un type défini par son nom (*extention*). C'est le cas par exemple sous DOS ou Windows.
  - Un fichier exécutable doit se terminer par `.exe`, `.com` ou `.bin`.
  - Le système reconnaît le logiciel à utiliser en fonction de l'extension.
- *Typage déduit* : le type du fichier dépend de son contenu ou de ses propriétés. C'est le cas sous Unix/Linux.
  - Un fichier est présumé exécutable s'il a le droit d'exécution.
  - On utilise souvent un code ou des directives placées en début de fichier.
  - Voir la commande `file` sous Unix.
- *Type MIME* ou *Content-Type* : typage des données sur internet.
  - Les pages web ou les emails (pièces jointes) utilisent le type MIME.
  - Le navigateur ou le logiciel de lecture choisit le logiciel à appeler.

Chaque type de fichier présente une organisation interne qui lui est propre, pour pouvoir représenter un certain genre de données.

Quelques exemples :

- *Fichiers texte* :
  - choix d'un encodage pour les caractères accentués ou spéciaux (`UTF8` ou `iso8859-1` ; commande `iconv`).
  - Lignes terminées par des caractères spéciaux : Carriage Return (CR, `'\r'`), Line Feed (LF, `'\n'`), ou CRLF (les deux à la suite).
- *Fichier exécutable ELF* (sous Unix/Linux)
  - Une entête décrit la position des différentes parties du fichier, le sens du codage (little/big-endian), l'architecture du processeur...
  - Des sections (données `.data`, constantes `.rodata`, code `.text`, la table des symboles `.symtab`...)
- Chaque *Système de Gestion de Base de Données* (SGBD) utilise des formats qui lui sont propres pour stocker efficacement les données.

Répertoires

Définition (Répertoire)

Un répertoire (ou catalogue) est un fichier dont le rôle est d'organiser l'ensemble des fichiers :

- c'est une liste de fichiers.
- un répertoire peut faire partie d'un autre, d'où une structure arborescente.
- deux répertoires particuliers : . et ..

- Un même répertoire ne peut faire partie que d'*un seul autre*.
- Un fichier peut faire partie de plusieurs répertoires : *lien en dur*.
- Possibilité de fichiers qui sont des pointeurs, les *liens symboliques*

Appels systèmes (POSIX)

On se concentre sur les opérations de base sur les fichiers réguliers mais :

- il existe d'autres appels pour manipuler les fichiers réguliers,
- il existe d'autres fichiers que les fichiers réguliers !

Un fichier est manipulé à l'aide d'un entier appelé *descripteur de fichier* : il s'agit de son indice dans la table des fichiers ouverts du processus.

Tout processus dispose des descripteurs de fichiers suivants :

- 0, `STDIN_FILENO` : son entrée standard ;
- 1, `STDOUT_FILENO` : sa sortie standard ;
- 2, `STDERR_FILENO` : sa sortie d'erreur standard.

On va voir comment en ouvrir d'autres.

- **Ouverture ou création d'un fichier régulier :**

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

  - `pathname` est le chemin du fichier à ouvrir ou à créer.
  - `flag` détermine le mode d'ouverture du fichier ; il s'agit d'un « ou-bit-à-bit » entre différentes constantes :
    - `flag` doit inclure soit `O_RDONLY`, soit `O_WRONLY` soit `O_RDWR` ;
    - peut inclure `O_CREATE`, `O_APPEND`, `O_TRUNC`, ...
  - En cas d'échec, -1 est retourné, et `errno` contient un code d'erreur.
  - En cas de succès, l'appel retourne un *descripteur de fichier* qui est un entier que l'on va pouvoir utiliser par la suite pour manipuler le fichier.

- **Ecriture via un descripteur de fichier :**

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

  - `size_t` est un type entier non-signé, et `ssize_t` son homologue signé.
  - L'appel tente d'écrire *au plus* `count` octets à partir de l'adresse `buf` sur le descripteur de fichier `fd`, et retourne `nbwr`.
  - En cas de succès, `nbwr` > 0 (mais on peut avoir `nbwr` $<$ `count`).
  - En cas d'échec, `nbwr` $= -1$, et `errno` contient un code d'erreur.
  - Si `nbwr` $= 0$, la situation dépend du code présent dans `errno` : dans certains cas on peut reprendre, dans d'autres il faut abandonner.

- **Lecture via un descripteur de fichier :**

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

  - L'appel tente de lire *au plus* `count` octets sur `fd` en les les rangeant à partir de l'adresse `buf`, et retourne `nbrd`.
  - En cas de succès, `nbrd` > 0 (mais on peut avoir `nbrd` $<$ `count`).
  - En cas d'échec, `nbwr` $= -1$, et `errno` contient un code d'erreur.
  - Si `nbrd` $= 0$, à nouveau, ça dépend de `errno` ; mais pour un fichier régulier, cela indique souvent que l'on a atteint la fin du fichier.

- **Fermeture d'un descripteur de fichier :**

```
#include <unistd.h>
int close(int fd);
```

  - `fd` est le descripteur de fichier à fermer.
  - En cas de succès, 0 est retourné, et les ressources associées avec le descripteur de fichier ouvert sont libérées.
  - En cas d'échec, -1 est retourné, et `errno` contient un code d'erreur. Un échec peut indiquer que des erreurs se sont produites précédemment.

- **Remarque importante :** personne ne sait tout ça par cœur (?)

Il faut être capable de retrouver ces infos dans les pages du manuel.

Exemple : une commande cat, sans gérer les erreurs :c

```
#define LEN 16
int main(int argc, char *argv[]) {
  int fd = open(argv[1], O_RDONLY); // descripteur de fichier
  char buf[LEN];                    // servira ici de buffer

  int nbrd = read(fd, buf, LEN);
  while(nbrd > 0) {
    int nrem = nbrd;  // nb. d'octets restant à récrire
    int nbwr = 0;     // nb. d'octets qui ont déjà été écrits
    while(nrem > 0) {
      int t = write(STDOUT_FILENO, buf+nbwr, nbrd);
      nrem -= t;
      nbwr += t;
    }
    nbrd = read(fd, buf, LEN);
  }
  close(fd);
  return 0;
}
```

Avec une gestion minimaliste des erreurs pour `read` et `write` :

```
int nbrd = read(fd, buf, LEN);
while(nbrd > 0) {
  int nrem = nbrd;
  int nbwr = 0;
  while(nrem > 0) {
    int t = write(STDOUT_FILENO, buf+nbwr, nbrd);
    if(t < 0) {
      cerr << strerror(errno) << endl;
      return 1;
    }
    nrem -= t;
    nbwr += t;
  }
  nbrd = read(fd, buf, LEN);
}
if(nbrd < 0) {
  cerr << strerror(errno) << endl;
  return 1;
}
```

Il existe beaucoup d'autres appels POSIX pour l'accès aux fichiers :

- `fcntl()`, `lseek()`, `stat()`,
- pour des fichiers de communication entre processus : `dup()`, `socket()`,
- pour accéder aux répertoires : `opendir()`, `readdir()`, `scandir()`, ...

Nous reparlerons de certains en temps voulu.

Il existe des fonctions d'accès de plus haut niveau aux fichiers :

- Les types et fonctions de la bibliothèque C définis dans `stdio.h` : `FILE*`, `fopen()`, `fclose()`, `fread()`, `fwrite()`, `fprintf()`, `fscanf()`, ...
- Les objets de biliothèque C++ standard définies dans `iostream` : `cin`, `cout`, `cerr`, `operator<<`, `operator>>`, ...

Dans les deux cas, les fichiers sont manipulés comme des flux pour faciliter les opérations de lecture et d'écritures. Mais attention : *vous ne pouvez pas utiliser directement ces bibliothèques sur des descripteurs de fichier POSIX*.

Plan

On parle de *système de fichiers* pour désigner la façon dont le stockage des fichiers est organisé sur un périphérique de mémoire secondaire.

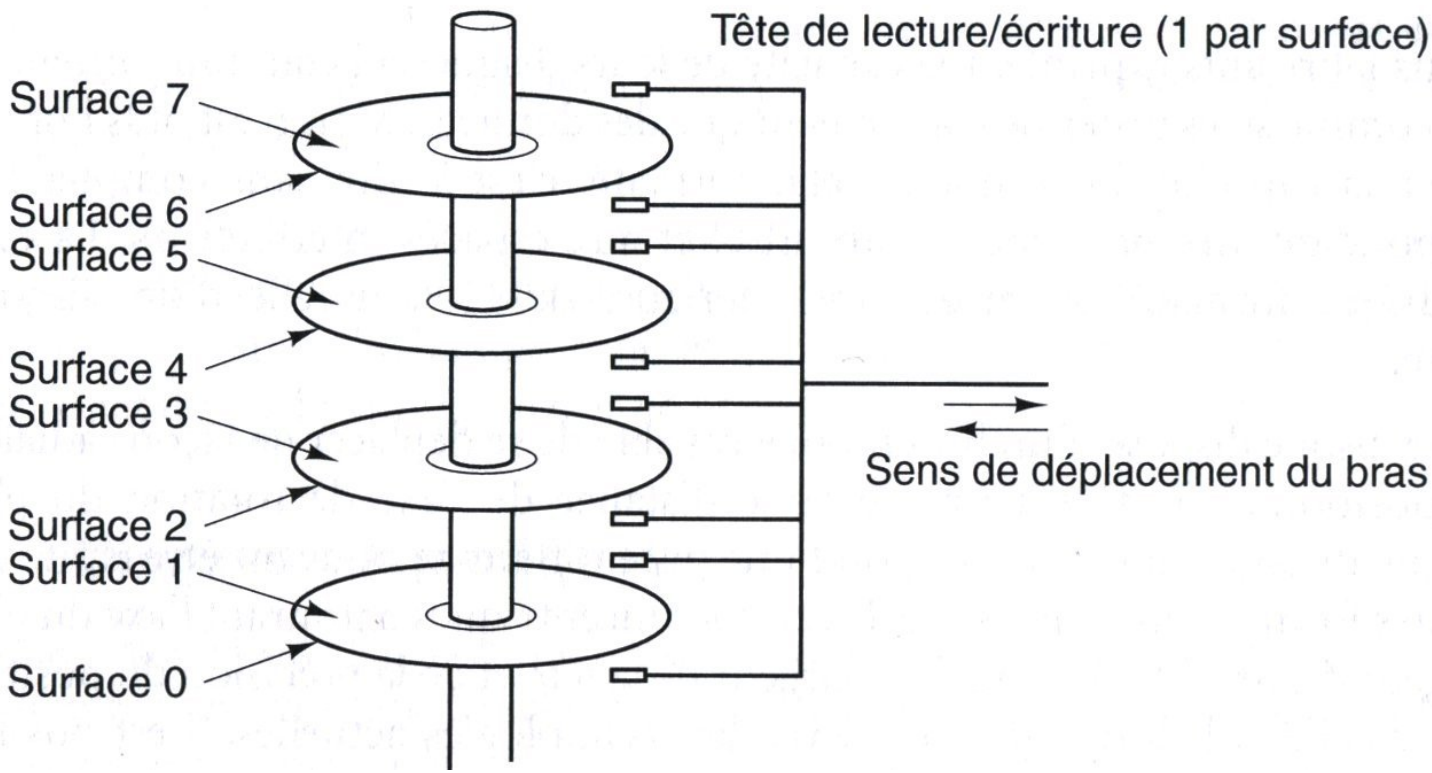
- Il existe différents types de mémoires secondaires :
  disque dur, mémoire flash (clé USB, carte SD, Solid Stat Drive),...
- De nombreux systèmes de fichiers ont été développés :
  NTFS, FAT, FAT16, FAT32, ext, ext2/3/4, zfs...
- Il existe aussi des systèmes de fichiers réseaux :
  SMB (Server Massage Block), NFS (Network File System)...

Dans la suite, on va

- surtout prendre le disque dur comme exemple de support,
- donner quelques idées sur les systèmes FAT et ext.

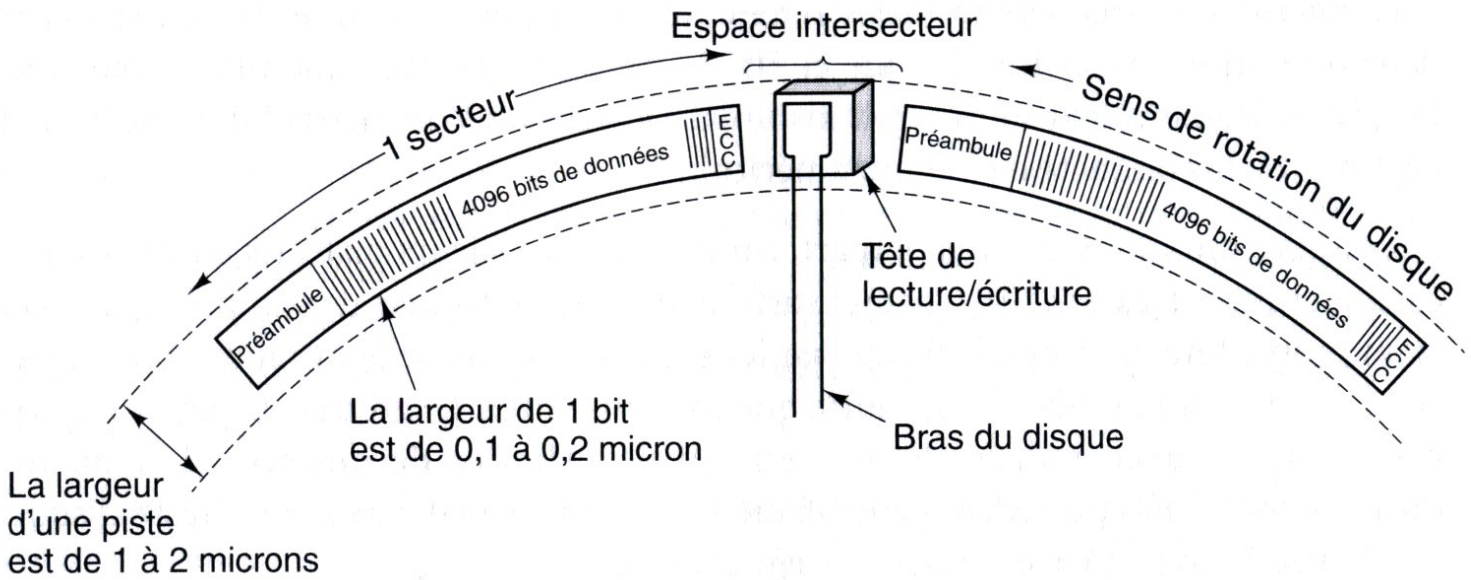
Pour les détails spécifiques à un système de fichiers, il n'y a pas de mystère : il faut aller lire les spécifications !

Les disques durs sont constitués de disques magnétiques rigides :



- *piste* : une zone couverte par une tête de lecture lors d'un tour de disque si le bras ne bouge pas ;
- *cylindre* : surface couverte par tous les bras sur tous les disques
- *secteur* : portion représentant une fraction de la surface angulaire

A l'emplacement déterminé par les valeurs (tête, cylindre, secteur, piste), se trouve physiquement un bloc de données appelé (aussi. . .) *secteur*.
Par exemple :



En général, le disque dur présente au système une interface :

- chaque secteur est adressable par un unique entier,
- il y a parfois une distinction entre secteur physique et logique.

Mais on n'a pas besoin d'entrer dans ces détails.

En tout cas, cela correspond bien au fait que les disques durs sont des *périphériques accédés par blocs*.

Le *bloc* est l'abstraction, au niveau du système, du secteur (un bloc peut être composé de un ou plusieurs secteurs logiques) :

- les données sont lues ou écrites *par bloc* (`:D`),
- l'on ne peut pas lire ou écrire moins d'un bloc,
- la lecture de toutes les données d'un bloc est « rapide »,
- entre deux blocs accédés, le déplacement du bras de lecture et l'attente du passage du bon secteur est plus lent.

Comme on ne peut pas lire ou écrire moins d'un bloc :

- tendance à provoquer de la fragmentation interne,
- perte de place (nombre de blocs occupés $\geq$ taille du fichier).
- (possibilité de faire de la stéganographie !)

Exemple (Un disque dur)

```
Disque /dev/sda: 931,5GiB, 1000204886016 octets, 1953525168 secteurs
Unités: secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 4096 octets
taille d'E/S (minimale / optimale) : 4095 octets / 4096 octets
```

$1000204886016 = 1953525168 \times 512$: ça tombe bien !

Par contre, une clef USB ne contient aucune partie mécanique, pas de tête, pas de cylindre, pas de piste. . . Mais le système fait l'interface :

Exemple (Une clef USB)

```
Disque /dev/sdb : 7,5GiB, 8011120640 octets, 15646720 secteurs
Unités : secteur de 1 * 512 = 512 octets
Taille de secteur (logique / physique) : 512 octets / 512 octets
taille d'E/S (minimale / optimale) : 512 octets / 512 octets
```

$8011120640 = 15646720 \times 512$: ça colle toujours !

Structure au niveau du fichier

Objectifs :

- allouer les blocs aux fichiers ;
- pouvoir retrouver les blocs dans le bon ordre ;
- la plupart des fichiers sont petits (quelques blocs) ;
- certains sont très gros.

1er exemple, allocation contiguë.

- Les fichiers sont stockés en un seul morceau.
- Le répertoire ne contient que le numéro du premier bloc et la taille.
- Les accès sont très rapides,
- Cause une grosse fragmentation externe.
- Problème pour augmenter un fichier.

2ème exemple, organisation par listes chainées.

- Le répertoire ne contient que le premier bloc du fichier.
- Ce bloc renvoie au suivant . . .
- C'est le principe des système des fichiers utilisant une table FAT (*File Allocation Table*) :
  - il y a exactement une entrée dans la table par bloc du disque,
  - chaque entrée contient l'indice de l'entrée suivante, ou EOF.

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
		EOF	13	2	9	8		4	12	3		EOF	EOF	

Répertoire	
A	6
B	10
C	5

A → 8 → 4 → 2 → EOF

B → 3 → 13 → EOF

C → 9 → 12 → EOF

3ème exemple, organisation par inœud.

On utilise plusieurs niveaux d'indirections :

- « une table, qui pointe sur une table, . . ., qui pointe sur un bloc »
- comme beaucoup de fichiers sont petits, la capacité doit être variable.

Supposons des blocs d'1 kio (soit 256 entiers de 32 bits).

- L'*inœud* contient 13 adresses de blocs.
- Les blocs 0 à 9 contiennent des données (au plus 10 kio).
- Le bloc 10 contient une table d'indirections simples :
  - 1 table qui pointe vers 256 blocs de données, soit 256 kio max.
- Le bloc 11 contient des indirections doubles :
  - 1 table, qui pointe vers 256 tables, qui pointent vers 256 blocs, soit $2^8 \times 2^8 \times 2^{10}$ o $= 64$ Mio max.
- le bloc 12 contient des indirections triples :
  - 1 table, qui pointe vers 256 tables, qui pointent vers 256 tables, qui pointent vers 256 blocs, soit 16 Gio max.

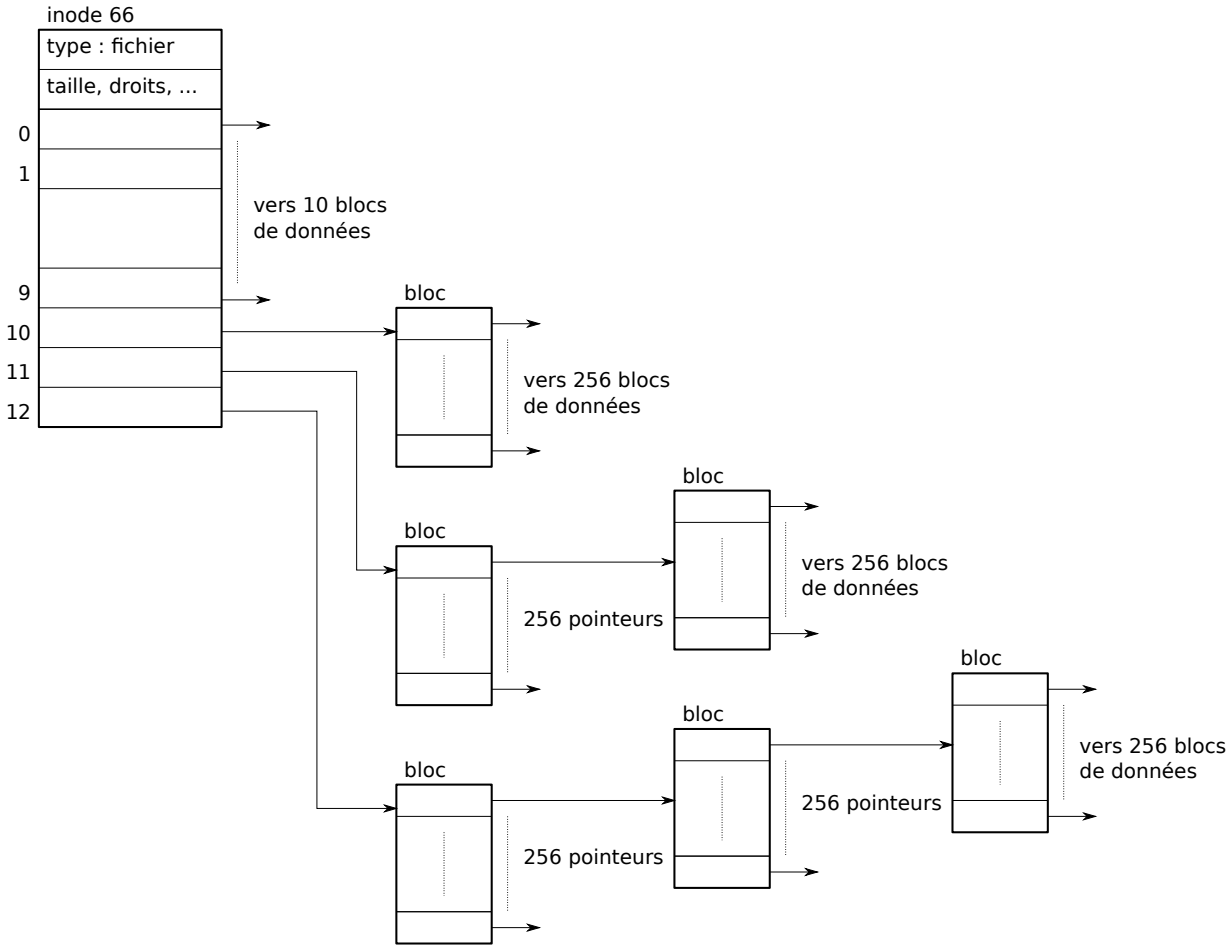
C'est le principe des systèmes de fichiers utilisés avec Linux : ext, ext2/3/4.

Outre les adresses des blocs, l'inœud contient :

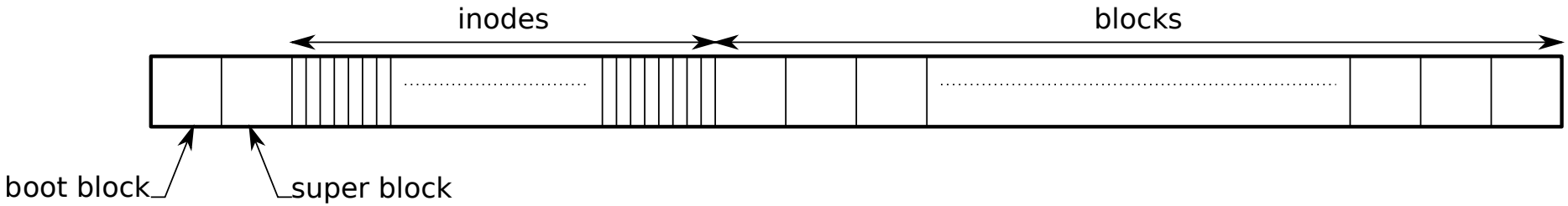
- la taille du fichier en octets,
- la taille du fichier en nombre de blocs,
- lidentifiant du périphérique contenant le fichier,
- l'identifiant du propriétaire du fichier,
- l'identifiant du groupe auquel appartient le fichier,
- les droits (lecture/écriture/exécution, plus setuid bits...),
- Les dates de dernière modification l'inœud (ctime), de dernière modification du fichier (mtime), du dernier accès (atime),
- Un compteur indiquant le nombre de liens matériels sur cet inœud.

Il ne contient pas le nom du fichier

- Un bloc contient soit un fichier, soit une table d'indirections :
  - les blocs 0 à 9 sont des blocs de données,
  - le bloc 10 des indirections simples,
  - le bloc 11 des indirections doubles,
  - le bloc 12 des indirections triples.

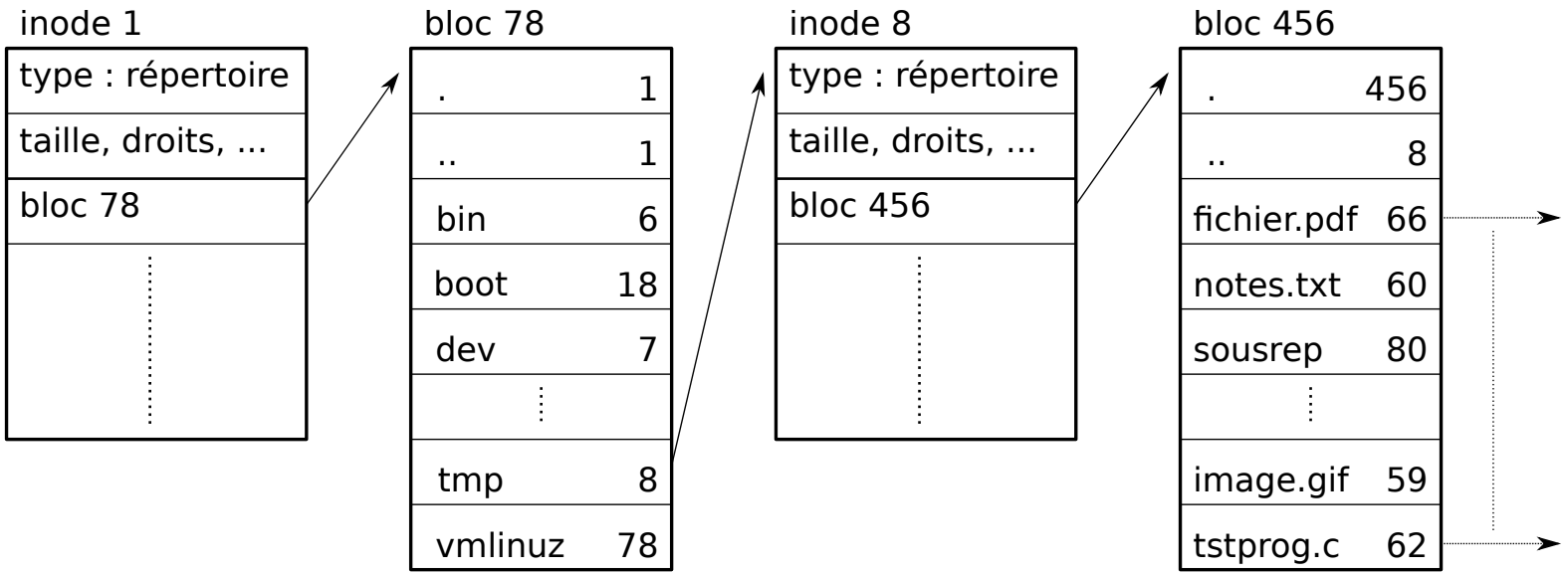


Une table des inœuds est stockée sur le disque :



Un fichier peut être : un fichier régulier, un répertoire, un fichier spécial . . .

Exemple avec des répertoires :



Contenu des répertoires.

Pour retrouver un fichier, le répertoire contient :

- Pour les systèmes de blocs chainés ou contigus :
  - L'adresse du premier bloc.
- Pour le système FAT :
  - Le numéro du premier bloc dans la table.
- Pour les systèmes ext 2/3 (avec inœud) :
  - le numéro de l'inœud unique pour la partition.
  - nécessite une table des inœuds sur le disque.
  - cette opération donne un nom au fichier.
  - si le même inœud est référencé plusieurs fois, ce fichier a plusieurs noms : on parle de *lien en dur* (*hard link* en australien).

Gestion des blocs

Le système doit maintenir la liste des blocs libres pour pouvoir les attribuer rapidement à un fichier.

- Utilisation d'une chaîne de blocs libres :
  - Chaque bloc contient l'adresse du bloc libre suivant.
  - Mais il contient aussi le plus grand nombre possible d'adresses de bloc.
  - Cela permet d'obtenir une liste de blocs libres en quelques lectures.

- Utilisation d'une ≪ carte ≫ des blocs du disque :
  - Un vecteur de bits : chacun correspond à un bloc et est égal à 1 si le bloc est libre.
  - Vu la taille des blocs, cette liste est relativement petite. Par exemple pour un disque de 2Gio décomposé en blocs de 2Kio, il suffit de 64 blocs de 2Kio.

Gestion des droits

La plupart des systèmes doivent géer plusieurs utilisateurs :
il faut donc de gérer leurs différents droits.

- Un utilisateur standard a le droit
  - d'utiliser les logiciels,
  - d'utiliser son espace de stockage (compte / répertoire personnel),
  - de lire les données partagées.
- Certains utilisateurs particuliers servent à
  - limiter les droits des serveurs (ex. apache),
  - gérer des accès distants (administration à distance),
  - avoir des configurations particulières (ex : oracle).
- Un utilisateur spécial a tous les droits :
  - l'*administrateur* sous Windows,
  - *root* sous Unix.

Gestion des droits Unix

- Les droits sont les droits sur les fichiers (tout est fichier).
- Les droits de base sont :
  - `read` lecture du fichier, liste du contenu du répertoire ;
  - `write` écriture dans le fichier, ajout/suppression de fichier dans le répertoire ;
  - `execute` exécution du fichier, aller dans le répertoire *ou un sous répertoire*.
- Pour un fichier un utilisateur est dans l'une des classes :
  - `user, u` : propriétaire ;
  - `group, g` : groupe du propriétaire ;
  - `other, o` : tous les autres.
- `root` a toujours le droit
- Tout processus a un propriétaire égal à :
  - celui qui a lancé la commande (`SetUID bit = 0`) ;
  - celui à qui appartient la commande (`SetUID bit = 1`).

Exemple.

- Pour mettre en place sa page internet personnelle, il faut que l'utilisateur *apache* ou *html* ait le droit de lire le contenu du répertoire `~/public_html/` donc :
  - `~/` doit être autorisé en exécution pour les autres.
  - `~/public_html/` doit être autorisé en exécution et lecture pour les autres.
- Les mots de passes doivent être protégés . Mais la commande `password` doit permettre de lire et modifier son mot de passe :
  - `/etc/shadow` est en lecture uniquement pour son propriétaire `root`
  - `/usr/bin/passwd` appartient à `root`, est autorisé en exécution pour tous avec un `setUID bit` $= 1$.
- Les droits permettent de protéger le système tout en délégant des droits étendus via certaines commandes.

Les liste de contrôle d'accès (ACL)

Le système de droits n'est pas suffisant :

- Il n'y a pas de droits négatifs (tous sauf) :
  - par exemple avec apache, les accès sont basés sur `allow` et `deny` et un ordre de lecture des droits
- Seulement 3 types de personnes...
  - Fastidieux, pour gérer finement les droits : les utilisateurs doivent être dans de nombreux groupes
  - Quand un utilisateur crèe un fichier, à quel groupe appartient-il ?
- Une solution est d'associer à chaque objet une liste de droits (ou déni de droits) accordés à des utilisateurs ou des groupes. Ce sont les *Access Control List* ou *ACL*.

- Une ACL est une liste d'*ACE* (*AC Entries*)
- Les droits sont positifs ou négatifs
- Une ACE est formé :
  - d'un droit particulier (lecture, écriture, contrôle total, changer les droits...) ;
  - d'un utilisateur ou d'un groupe ;
  - d'un objet sujet ;
  - d'un booléen `Allow` ou `Deny`.
- Exemple
  - Windows (droits de base, droit sur NTFS), Linux (compat. mais peu utilisé).
  - LDAP, firewall, Andrew File System (AFS), ...
  - Forums, blogs ...

Par exemple, dans un serveur LDAP, la syntaxe des ACE ressemble à :

```
olcAccess: to [ressource]
  by [à qui] [type d'accès autorisé]
  by [à qui] [type d'accès autorisé]
  by [à qui] [type d'accès autorisé]
```

Un exemple de base, qui limite l'accès à l'attribut `userPassword` :

```
olcAccess: to attrs=userPassword
 by self write
 by group.exact="cn=adm,ou=groups,dc=example,dc=com" write
 by anonymous auth
 by * none
```

Pour gérer les accès sur un objet de l'arborescence :

```
olcAccess: to dn.subtree="ou=applications,dc=example,dc=com"
 by group.exact="cn=adm,ou=groups,dc=example,dc=com" write
 by users read
 by * none
```

Plan

Ce dont on a peu ou pas parlé...

- Besoin des utilisateurs :
  - *sécurité des données* (détection et correction d'erreurs) ;
  - *confidentialité* (chiffrement des disques, droits d'accès) ;
  - *sauvegardes* (incrémentales, clichés).
- Au niveau du matériel :
  - support à mémoire flash, avec *répartition des écritures* ;
  - *RAID* (*Redundant Array of Independent Disks*) ;
  - disques de plus en plus gros ($\gg$ To).
- *Systèmes de fichier en réseau* :
  - Un (ou des) serveurs partagent leurs fichiers.
  - Le système client présente les fichiers comme des fichiers locaux : les applications ne font pas la différence
    - ★ Exemples « historiques » : NFSv3/4 (Unix), SMB (Windows).
    - ★ Autre systèmes : Lustre, GFS (RedHat), GoogleFS, OCFS (Oracle).

- *Journalisation* : chaque modification est d'abord écrite dans un journal, puis oubliée quand elle est effectuée ; le système garde sa cohérence (ex : ext3, NTFS, HFS+).
- *Pré-allocation de zone continue* : lors d'une écriture, une zone plus grande est allouée ; si le fichier est agrandi, il utilise cette zone ; cela évite la fragmentation (ex : ext4, NTFS, HFS+, Btrfs).
- *Vérification et défragmentation en ligne* : ces opérations sont faites durant l'utilisation normale ; permet la remise en route rapide.
- *Partionnement ou Logical Volume management (LVM)* :
  - partitionnement : découpage prévu en matériel d'un disque en plusieurs partitions ;
  - LVM : technologie permettant d'assouplir les schémas de partionnement usuels.

Processus et communication

LIF12-Système d'Exploitation

Nicolas Louvet

Univ. Claude Bernard Lyon 1

Séance 3

Les responsables de l'UE : Nicolas Louvet et Laure Gonnord.

Ce support de cours est largement repompé de celui de Fabien Rico.

Vous connaissez déjà bien la *programmation séquentielle*.

Programmation *concurrente*.

- Écrire un programme qui fait plusieurs choses en même temps ou plusieurs programmes qui interagissent,
- Organiser tout ça pour arriver au bon résultat :
  - résoudre les problèmes d'accès concurrent, de synchronisation,
  - gérer les échanges de données.

Rôle du système pour la programmation concurrente :

- commencer, exécuter, terminer un programme,
- passage d'une tâche à l'autre, déroutement.
- sécurité.

Problématiques pour le programmeur :

- comment utiliser le multi-tâche ?
- comment faire communiquer des tâches entre elles ?

Notion de processus

Exécution d'un programme

Les tâches du système sont de :

- trouver le fichier sur le disque ;
- trouver de la place en mémoire ;
- charger le programme en mémoire ;
- trouver le point d'entrée du programme (`main()`) ;
- exécuter la première instruction du programme ;
- poursuivre le déroulement du programme (*pointeur d'instruction*) ;
- à la fin du programme, libérer les ressources qu'il occupe ;
- obtenir et traiter son résultat.

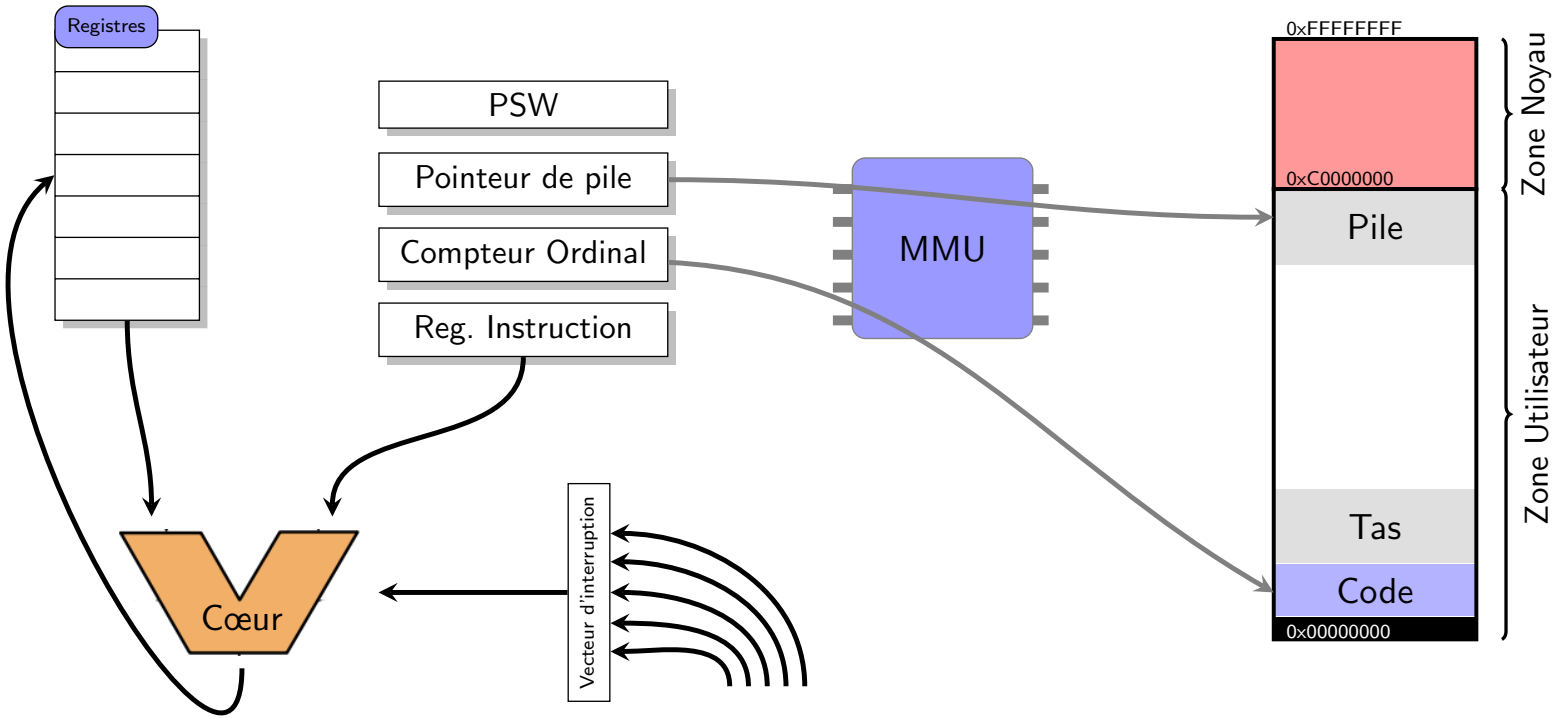
Notion qui définit une exécution d'un programme : le *processus*.

Définition (Processus)

Un processus est une instance (une exécution) d'un programme. C'est un ensemble de données gérées par le noyau qui contient toutes les informations nécessaires afin de suivre le déroulement du programme, de le stopper et de le reprendre.

Un processus doit :

- identifier le programme (dont il est un instance) ;
- vérifier que ce programme ne fait que ce qu'il a le droit de faire ;
- lui permettre d'accéder aux fichiers et aux ressources ;
- isoler ce programme de tous les autres ;
- libérer les ressources à la fin de l'exécution (fermer les fichiers, libérer toute la mémoire allouée, etc.) ;
- permettre le passage d'une tâche à l'autre.



Dans le processeur, à un instant donné, *l'état du processus* est décrit par :

- les valeurs contenues dans les registres : compteur ordinal, registres généraux, registre de status, pointeur de pile...
- l'état de la zone mémoire qui lui a été attribuée par le système.
- le contenu de son vecteur d'interruption.

Du point de vue du système, *l'état du processus* comporte également :

- l'état d'exécution du processus (en cours, prêt, bloqué, ...) ;
- les ressources qu'il utilise (fichiers, sockets, ...) ;
- son environnement (répertoire courant, id de l'utilisateur, ...) ;
- informations de gestion (id, pid, gid, interruptions en attente, ...) ;
- ...

Un processus n'accède jamais directement à la mémoire, mais à une *mémoire virtuelle* :

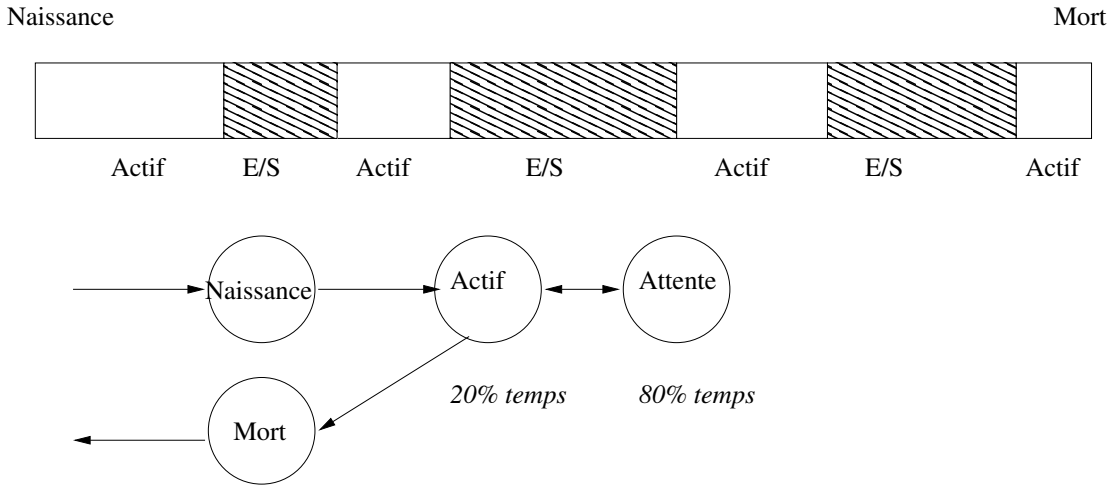
- le processus accède à des adresses virtuelles ;
- l'espace mémoire du processus est découpé en pages, qui peuvent être placées « n'importe où » dans la RAM, voir sur le disque ;
- les adresses virtuelles sont converties en adresses réelles à la volée.

Chaque processus dispose ainsi de son propre espace mémoire (4 Gio pour une installation 32 bits), et *ne peut pas interférer avec les autres processus*.

Commutation de processus

Partons d'un constat simple :

- les périphériques sont (souvent) plus lents que le processeur
- dans le cas de transferts par bloc (Direct Memory access, DMA), le processeur ne fait rien, si ce n'est attendre la fin du transfert, ...



D'où une idée simple : allouer le processeur à un autre processus !
↪ c'est la *commutation de processus*

En gros, lors de la commutation d'un processus A à un processus B :

- l'exécution de A est interrompue,
- le système reprend la main, et sauvegarde l'état actuel de A,
- le système restaure l'état du processus B dans le processeur,
- le processus B reprend son exécution.

Chaque processus a son espace mémoire réservé : peu à faire de ce côté.

La *commutation permet* notamment :

- d'exploiter le temps requis pour effectuer des entrées/sortie,
- l'exécution concurrente de plusieurs processus.

Un processus est interrompu à chaque fois qu'il effectue un appel système, par des interruptions matérielles, ou par une horloge : à chaque fois le système reprend la main, et choisit le prochain processus à exécuter.

État d'un processus

Rappel : Le système doit

- gérer l'accès au processeur
- gérer l'occupation de la mémoire

Définition (état d'un processus)

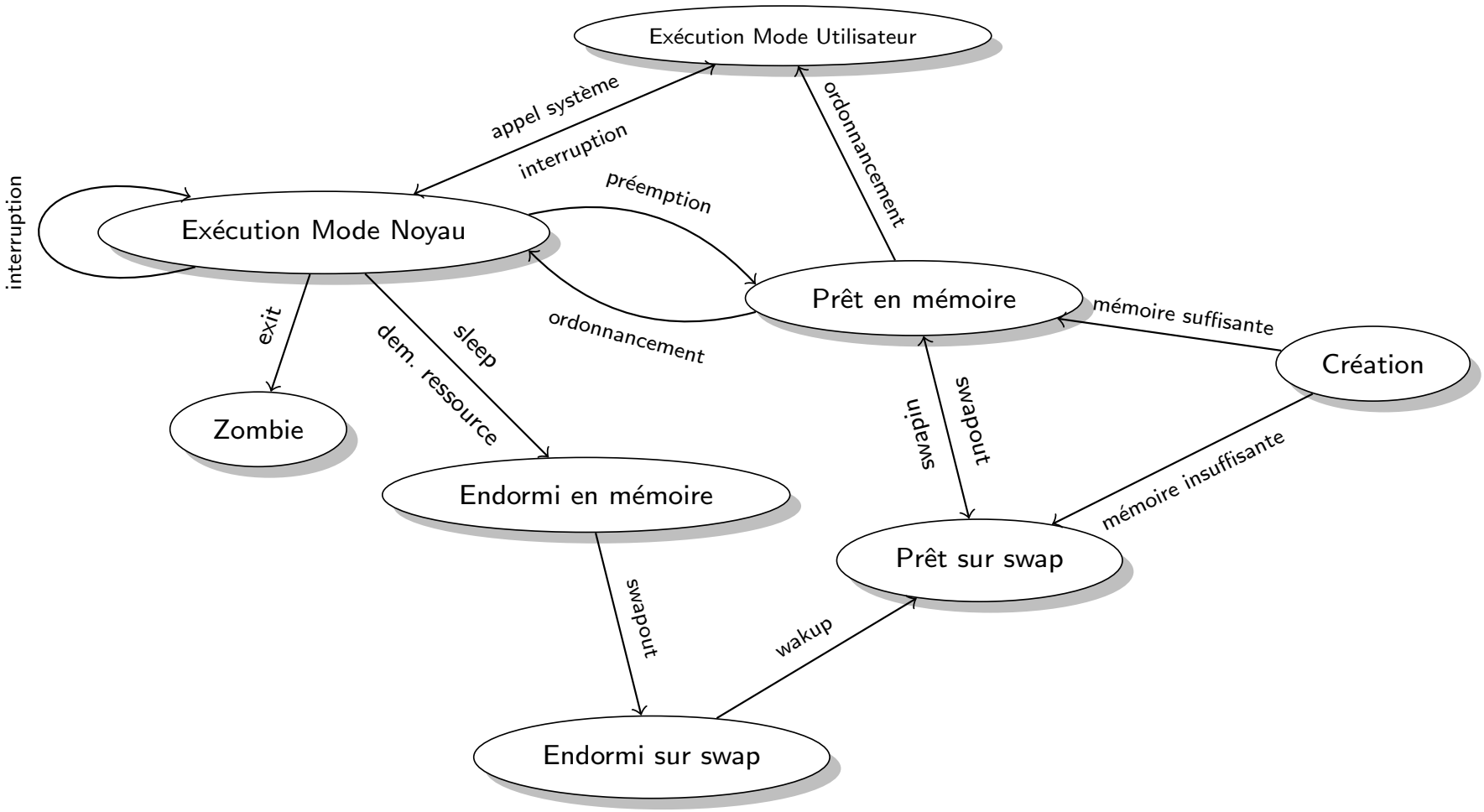
Le système doit donc gérer plusieurs files d'attentes de processus :

- ceux qui peuvent s'exécuter,
- ceux qui sont en mémoire,
- ceux qui sont bloqués car demandent une ressource occupée,
- ...

On parle *d'état du processus*.

Quand le système reprend la main, il met à jour ces files d'attentes, et détermine le prochain processus à s'exécuter : c'est *l'ordonnancement*.

Exemple : possible automate d'évolution de l'état d'un processus.



Rappel. Pour voir l'état des processus : `ps -l` ou `top` (`q` pour quitter).

Observation des processus

Sous Unix, les processus sont organisés de façon *hiérarchique* :

- Chaque processus est identifié de façon unique par un entier : c'est son *PID*, pour *Process Identifier*.
- Chaque processus peut créer des processus appelés *fils*.
- Tous les processus ont accès à l'identifiant de leur *père* : c'est leur *PPID*, pour *Parent Process Identifier*.

A la racine de la hiérarchique il y a l'*ancêtre* de tous les processus :

- son PID et son PPID sont par convention égaux à 1 ;
- sous GNU/Linux, `systemd` (avant c'était `init` avec System V).

Deux commandes utiles pour observer les processus :

- `ps` avec notamment les options `-e` et `-l` ;
- `pstree` pour avoir une vue de l'arborescence des processus.

Un exemple :

```
nlouvet:~$ xclock & xcalc
nlouvet:~$ bash
nlouvet:~$ xcalc &
nlouvet:~$ ps -l
F S   UID   PID  PPID  C PRI  NI ADDR SZ WCHAN  TTY          TIME CMD
0 S  1000  5503  4261  0  80   0 -  6317 wait   pts/3    00:00:00 bash
0 S  1000  5555  5503  0  80   0 - 18513 poll_s pts/3    00:00:00 xclock
0 S  1000  5556  5503  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
0 S  1000  5558  5503  0  80   0 -  6065 wait   pts/3    00:00:00 bash
0 S  1000  5566  5558  0  80   0 - 13683 poll_s pts/3    00:00:00 xcalc
4 R  1000  5567  5558  0  80   0 -  7558 -      pts/3    00:00:00 ps
nlouvetk:~$ pstree -p 5503
bash(5503)---bash(5558)---pstree(5568)
           |             |-xcalc(5566)
           |-xcalc(5556)
           |-xclock(5555)
```

On retrouve bien :

- les informations sur chaque processus (PID, PPID, état (S)...) ;
- l'organisation hiérarchique des processus.

Création de processus

Un exemple : la fonction `int system(const char * commande);` :

- permet à un programme d'exécuter une autre commande ;
- toute commande reconnue par le `shell` peut être utilisée ;
- la fonction stoppe le processus, attend la fin de la commande,
- puis récupère et donne le retour de la commande.

Ce qui se passe :

- le processus A crée un nouveau processus et stoppe ;
- le nouveau processus B « lance la commande » ;
- A attend pour récupèrer le retour de B, puis il reprend son cours.

↪ Un appel à `system` se decompose en trois appels systèmes : `fork()`, un appel de la famille `exec()`, `waitpid()`.

pid_t fork(void)

Que dit le manuel ?

- `fork()` creates a new process by duplicating the calling process. The new process, referred to as the child, is an exact duplicate of the calling process, referred to as the parent, except for the following...
- On success, the PID of the child process is returned in the parent, and 0 is returned in the child. On failure, -1 is returned in the parent, ..., and errno is set appropriately.

Après un appel réussi à `fork()` :

- le fils est une copie du père (même code, même mémoire),
- seule la valeur de retour de `fork()` permet de les distinguer.

Il faut tester la valeur de retour pour différentier leurs exécutions.

`pid_t waitpid(pid_t pid, int *wst, int opt);`

L'appel permet d'attendre qu'un fils se termine :

- `pid` est le PID du fils qu'on attend, ou 0 si l'attend n'importe lequel ;
- `wst` permet de récupérer les infos de retour du fils ;
- si `opt` est à 0 alors l'attente est bloquante ;
- si `opt` est à `WNOHANG`, l'appel retourne immédiatement.

Que dit le manuel ?

In the case of a terminated child, performing a wait allows the system to release the resources associated with the child ; if a wait is not performed, then the terminated child remains in a "zombie" state.

Valeur de retour :

- en cas d'échec, -1 est retourné et `errno` mise à jour ;
- si un fils s'est terminé, le PID de ce fils ;
- si l'appel est non-bloquant et qu'aucun fils n'est terminé, alors 0.

Fin d'un processus :

- Lorsqu'il se termine, un processus passe forcément en état *zombie* : il n'est plus jamais prêt mais ses ressources ne sont pas toutes libérées.
- Notamment, la valeur de retour de la fonction `main()` est en attente.
- Le père doit lire ce résultat par l'appel `waitpid()` :
  dans le cas, le processus fils disparaît de la table des processus.
- Si le père meurt, le processus est adopté par `systemd`, qui se charge de libèrer les ressources sans lire le résultat.

Un zombie d'occupe des ressources tant que son père n'a pas fait un appel à `waitpid()` ou n'est pas terminé : pour des processus de longue durée, il est très important de ne pas laisser leurs fils dans cet état lamentable !

Un exemple :

```
pid_t code= fork();
if (code == -1) {... /* traitement de l'erreur */}
if(code == 0) {/* code du fils */
   ...
   exit(23);
}
else{
   /* code du père */
   int status;
   ...
   waitpid(-1,&status, 0); /* Attente d'un fils */
   if (WIFEXITED(status)) {
      fprintf(stdout, "Le fils a retourné %d\n",
                        WEXITSTATUS(status));
   }
}
```

`int exec();`

En fait, il existe toute une famille de fonctions, listées dans `man exec`.

Que dit le manuel ?

The exec() family of functions replaces the current process image with a new process image. The functions [listed here] are front-ends for execve.

The exec() functions return only if an error has occurred. The return value is -1, and errno is set to indicate the error.

Le processus appelant est remplacé par la commande en argument d'`exec()`. Par exemple, la primitive

```
int execvp(const char *file, char *const argv[]);
```

permet de remplacer le processus appelant par le commande dont le chemin est donné par `file`, en lui passant les arguments qui sont dans `argv`.

Un exemple :

```
int main(void) {
  while(1) {
    string cmd;
    cout << "Entrez une commande : ";
    cin >> cmd;
    if(cmd == "quitter") break;

    pid_t pid = fork();
    if(pid == -1) exit_err();
    if(pid == 0) { // processus fils
      char *args[2];
      args[0] = new char[cmd.length()+1];
      args[1] = NULL;
      strcpy(args[0], cmd.c_str());
      if(execvp(cmd.data(), args) == -1) {
        // afficher un message qui va bien
        return 1;
      }
    }
    else // processsus père
      if(waitpid(pid, NULL, 0) == -1) exit_err();
  }
  return 0;
}
```

Environnement

Les processus sont séparés, mais fonctionnent dans un environnement :

- ils sont exécutés depuis un working directory,
- ils appartiennent à un utilisateur et héritent de ses droits,
- ils ont une définition de l'environnement linguistique,
- ...

Beaucoup de ces informations sont transmises par des variables héritées de leur processus père : les *variables d'environnement*.

Exemples de variables d'environnement :

- `PATH` : liste des répertoires où sont cherchés les exécutables.
- `USER` : nom de l'utilisateur.
- `HOME` : répertoire de l'utilisateur.
- `LD_LIBRARY_PATH` : liste des répertoires de recherche des bibliothèques.
- ...

Commandes utiles (Bash) :

- `echo $VARIABLE` : pour afficher une variable !
- `env` : pour lister toutes les variables d'environnement.
- `export VARIABLE=valeur` : exporter une variable dans l'environnement d'un processus et tous ses descendants.

En C/C++ :

- `extern char **environ` : tableau des variables d'environnement.
- `char *getenv(const char *name)` : valeur d'une variable.
- `int setenv(const char *name, const char *value, int overwrite)` : pour changer ou ajouter une variable d'environnement.
- `char *getcwd(char *buf, size_t size)` : répertoire courant.
- `int chdir(const char *path)` : change le répertoire courant.

Méfiez-vous des changement inconsidérés de variables d'environnement !

La communication entre processus

Le système sépare les processus :

- pour éviter les perturbations,
- pour assurer la protection des données,
- pour faciliter la gestion.

Sans faire appel au système, un processus ne peut pas agir sur un autre.

Mais la communication entre processus est nécessaire

- certains processus doivent communiquer, *e.g.*, serveur
- tout processus nécessite un moyen d'être contacté, ne serait-ce que pour pouvoir l'annuler !

Un exemple :

Lorsqu'on appuie sur une touche :

1. le matériel prévient le système,
2. qui prévient le serveur graphique,
3. qui prévient le logiciel concerné,
4. qui calcule une nouvelle image,
5. et demande au serveur graphique de l'afficher.

Il faut être capable :

- de prévenir un processus que quelque chose s'est passé ;
- d'échanger des informations avec un processus.

Le matériel utilise communiquer avec le système :

- des *interruptions* pour prévenir et dérouter le processus en cours ; à chaque interruption correspond une action prédéfinie ;
- des zones mémoires pour échanger les informations ; l'accès à ces zones est sécurisé par le système (mode noyau).

De la même manière, les processus disposent pour communiquer :

- de *signaux* qui sont des alarmes qui stoppent le fonctionnement du processus pour lui faire exécuter une fonction *gestionnaire de signal*
- de *tubes*, *sockets*, fichiers, partages mémoire, ...

Utilisation des signaux

Définition

Un signal est un moyen de prévenir un processus d'un évènement :

- c'est un message simple, essentiellement un entier ;
- il existe un petit nombre de messages prédefinis :
  - `SIGINT` = 2 (touche Ctrl+c),
  - `SIGSTOP` = 19 (touche Ctrl+z),
  - `SIGKILL` = 9 (kill),
  - `SIGSEGV` = 11 (erreur de segmentation),
  - ...
- c'est un évènement asynchrone : il peut arriver n'importe quand ;
- il provoque le déroutement du processus vers un gestionnaire ;
- la communication est très limitée : type du signal, émetteur, ...

Par défaut, un signal est ignoré ou il est traité par un gestionnaire prédéfini.
Pour changer ce comportement, il faut installer un gestionnaire de signal.

Commandes shell utiles :

- `kill -s signal pid` : pour envoyer un signal,
- `kill -l` : pour lister les signaux existants.

En C/C++ POSIX :

- `int kill(pid_t pid, int sig)` : pour envoyer un signal,
- `int sigaction(int signum, const struct sigaction *act, struct sigaction *oldact)` : mettre en place un gestionnaire de signal.

La structure `sigaction` est ≪ intimidante ≫ :

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

Mais on peut déjà se débrouiller en repartant d'une structure déjà existante, et en modifiant le pointeur de fonction `sa_handler`.

Un exemple :

```
// le gestionnaire de signal
void fct(int s) {
  static int cpt = 0;
  cpt++;
  std::cout << "M'enfin ! (" << cpt << ")"<< std::endl;
  if (cpt == 10) {
    std::cout << "Aie !!!" << std::endl;
    exit(0);
  }
}

int main(int argc, char *arv[]) {
  struct sigaction s;

  // installation du gestionnaire pour SIGINT (Ctrl+c)
  if( sigaction(SIGINT, NULL, &s) == -1 ) exit_err();
  s.sa_handler = fct;
  if( sigaction(SIGINT, &s, NULL) == -1 ) exit_err();

  while(1) {
    std::cout << "RRRRR..." << std::endl;
    sleep(1);
  }

  return 0;
}
```

Conclusion

Appels systèmes / primitives :

- pour la gestion des processus : `fork()`, `exec()`, `waitpid()`.
- pour les variables d'environnement : `getenv()`, `setenv()`.
- pour l'utilisation des signaux : `kill()`, `sigaction()`.

En ligne de commande :

- gestion des processus : `ps`, `top`.
- pour les variables d'environnement : `env`, `export`
- pour l'utilisation des signaux : `kill`.

Nous verrons deux autres moyens de communication entre processus :

- les pipes, pour la transmission de données localement,
- les sockets, pour la transmission de données en réseau.

Communication et sockets

ASR5 - Système d'Exploitation

Nicolas Louvet

Univ. Claude Bernard Lyon 1

11 mars 2019

Les responsables de l'UE : Nicolas Louvet et Laure Gonnord.

Ce support de cours est largement repompé de celui de Fabien Rico.

Il faut être capable d'échanger des données : de tous types, de toutes tailles, de manière sécurisée et synchronisée.

Pour la communication entre processus, nous avons déjà vu :

- les signaux $\rightsquigarrow$ peu adapté à l'échange de données.
- les fichiers réguliers $\rightsquigarrow$ certes oui, mais...
  - $\ll$ plutôt lent $\gg$ car cela sollicite le disque (inutilement),
  - problèmes d'accès conccurents entre lecteurs et rédacteurs.
- fichiers spéciaux :
  - entrées et sorties standards des processus,
  - tubes anonymes (processus sont créés dans un même programme),
  - tubes nommés (qui ont un nom dans le système de fichiers).

Ces modes fonctionnent entre processus d'un même système...

Ils faut aussi pourvoir échanger des données entres processus via le réseau : c'est le rôle des sockets.

Les systèmes Unix utilisent le même système de descripteur de fichiers pour (presque) tous les types de fichiers : réguliers, spéciaux, blocs [1].

Cela permet d'utiliser les mêmes primitives d'accès, que ce soit sur des canaux de communication ou de vrais fichiers :

- on peut toujours utiliser `read()` et `write()` ;
- pour les sockets, ils existe des adaptations comme `recv()` et `send()`.

On peut utiliser des bibliothèques de haut niveaux pour les fichiers :

- la bibliothèque C avec `stdio.h` : `FILE *`, `fread()`, `fwrite()`... ;
- la bibliothèque C++ avec `iostream` : `cin`, `>>`, `cout`, `<<`... ;
- il existe des biblio. pour le échanges réseau (`libcurl`, `Boost.Asio`, ...).

On s'en tient aux primitives de bas niveau (`open()`, `read()` `recv()`, ...)

1. Les répertoires font exception : ils sont destinés à l'organisation des données.

Plan

Tubes anonymes (rappel)

Les fichiers spéciaux les plus simples pour la communication interprocessus sont les tubes anonymes, appelès généralement tubes ou pipes.

Les tubes fournissent un canal de interprocessus unidirectionnel :

- ils ont une extrémité d'écriture et une de lecture ;
- ils ont une taille limitée et peuvent être remplis ;
- Ils n'ont pas de nom et doivent donc être partagés dès la création.

`man pipe`

- `int pipe(int pipefd[2]);`
- `pipe()` creates a pair of file descriptors and places them in the array pointed to by `pipefd` :
  - `pipefd[0]` is for reading,
  - `pipefd[1]` is for writing.
- On success, zero is returned. On error, -1 is returned.

Pour échanger via un tube, deux processus doivent avoir un lien familial :

- l'un est fils de celui qui a créé le tube,
- les deux ont un ancêtre commun qui a créé le tube.

Chaque processus doit fermer le descripteur qu'il n'utilise pas :

- le lecteur ferme le descripteur en écriture du pipe (`pipefd[1]`),
- le rédacteur ferme le descripteur en lecture du pipe (`pipefd[0]`).

Lorsque le rédacteur a fini d'écrire, il ferme aussi le descripteur en écriture, pour que le lecteur sache qu'il n'aura plus rien à lire (`EOF`), et se termine.

Toujours libérer les ressources non utilisées ! On a vu en TD/TP que ne pas fermer les descripteurs non-utilisés peut conduire à un interblocage.

Tubes nommés (fifo)

Pour que deux processus sans lien familial puissent échanger via un tube, il faut qu'ils aient un nom dans le système de fichier : notion de tube nommé.

Un tube nommé ou fifo :

- est un tube qui a un nom dans le système de fichiers ;
- il est géré comme tout autre fichier :
  - nom, droits,
  - ouverture avec `open()`, fermeture avec `close()`,
  - lecture avec `read()`, écriture avec `write()` ;
- c'est un tube, et une fois ouvert, il s'utilise comme tel.

- Appel système pour créer un tube nommé :

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

  - `pathname` : est le nom du fichier ;
  - `mode` : représente les droits d'accès (UNIX).

- Commande pour créer un tube nommé :

```
mkfifo [-m mode] pathname
```

`pathname` et `mode` jouent les mêmes rôles que pour l'appel système.

- Chaque processus doit l'ouvrir :

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

  - `pathname` : le nom du tube à ouvrir ;
  - `flags` : mode d'ouverture notamment,
    `O_RDONLY` pour la lecture du côté du lecteur,
    `O_WRONLY` pour l'écriture du côté de l'ecrivain.

Exemple, uniquement sur la ligne de commande :

Dans un permier terminal, on crée un tube nommé, et on met la commande `cat` en attente de lecture sur l'entrée standard ; tout ce que `cat` lit sur l'entrée standard est redirigé vers le tube :

```
nlouvet:~/tmp$ pwd
/home/nlouvet/tmp
nlouvet:~/tmp$ mkfifo test
nlouvet:~/tmp$ ls -l test
prw-r--r-- 1 nlouvet nlouvet 0 mars 8 13:31 test
nlouvet:~/tmp$ cat > test
toto fait du vélo
```

Dans un autre terminal, on met `cat` en attente de lecture sur le tube nommé ; tout ce que `cat` lit sur le tube est redirigé vers la sortie standard :

```
nlouvet:~/tmp$ pwd
/home/nlouvet/tmp
nlouvet:~/tmp$ cat test
toto fait du vélo
```

Quelques notions sur les réseaux

Un dialogue via le réseau suppose l'existence de deux processus sur deux ordinateurs qui sont capables de se reconnaître et se transmettre des informations.

Dans les modèles les plus souvent utilisés (TCP/IP et UDP/IP) les données sont adressés :

- à un ordinateur particulier via une adresse IP destination ;
- à un processus particulier via un numéro de port destination.

De la même manière, l'origine des données est connue via *l'adresse IP source* et le *numéro de port source*.

Ces 4 valeurs permettent d'identifier un échange d'informations, voire une connexion.

Deux modèles de transport sont couramment utilisés :

- le modèle déconnecté (analogie : courrier papier), les données sont envoyées et reçues sous forme de paquets :
  - pas d'outil pour savoir si une donnée est perdue ;
  - pas d'outil pour assurer l'ordre dans lequel elles arrivent.

  ↪ protocole UDP (User Datagram Protocol)

- le modèle connecté (analogie : téléphone), une connexion est mise en place qui permet de gérer les échanges paquets :
  - si des paquets disparaissent ou arrivent dans le désordre, cela est automatiquement corrigé ;
  - la connexion est maintenue ; si elle se coupe de manière irréversible, une erreur est générée (`Broken Pipe`) ;
  - pas d'outil pour délimiter les messages ou les « ensembles de données » (page web, fichier...).

  ↪ protocole TCP (Transmission Control Protocol).

Nous n'utiliserons que le mode connecté, TCP/IP !

Avec TCP, il faut établir puis terminer la connexion ; deux acteurs :

- Le serveur qui attend une demande connexion et accepte le client.
- Le client qui est à l'origine de la demande de connexion.

↪ modèle client-serveur.

Chacun à des actions à faire :

- Le client doit contacter le processus serveur dont il connaît le nom de machine (ou son adresse IP) et le port.
- Le serveur doit faire deux choses :
  - se mettre en attente sur un certain port et certaines adresses IP ;
  - quand un client se connecte, dialoguer avec lui.

Un serveur peut accepter des connexions de plusieurs clients. Par exemple, un serveur web écoute sur le port 80 et répond à plein de clients.

Question : un serveur est en contact avec plusieurs clients. Lorsqu'il reçoit des données, comment reconnaît-il le client qui les envoie ?

Socket

L'outil central de la communication réseau est la *socket* (une « prise »).

Définition (Socket)

Tout comme un tube, une socket socket permet de définir un canal de communication entre deux processus, mais :

- elle est bidirectionnel ;
- elle permet l'utilisation du réseau ;
- elle permet de choisir différents protocoles.

En pratique, une socket est un descripteur de fichier de type `int`.

- Le client contacte le processus serveur dont il connaît l'adresse IP et le port, et met en place une socket de dialogue avec lui.
- Le serveur doit faire deux choses :
  - mettre en place une socket d'écoute, qui se met en attente sur un certain port et sur certaines adresses IP ;
  - créer la socket de dialogue avec le client qui se connecte.

L'API (interface) POSIX des sockets a peu changé depuis sa création. Cela indique sa souplesse mais explique aussi sa difficulté d'utilisation...

La première fonction de l'API est utile pour le client et le serveur : elle transforme les adresses de machines et les services en une liste de structures utilisables par d'autres fonctions de l'API (`bind()` et `connect()`).

```
int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

- `node` : nom ou adresse de la machine demandée ;
- `service` : le nom du *service* (`"http"`, `"ldap"`, `"ssh"`) ou son numéro de port (`"80"`, `"389"`, `"22"`) ;
- `hints` : un formulaire pour filtrer certains résultats ;
- `res` : l'adresse d'un pointeur où sera stocké le résultat.

Exemple de code :

```
struct addrinfo hints; // pour faire la demande de port
struct addrinfo *addr; // pour parcourir les résultats
int res; // gestion des erreurs

// on initialise le "formulaire de demande" hints
memset(&hints, 0, sizeof(struct addrinfo));
hints.ai_family = AF_UNSPEC; // IPv4 ou IPv6
hints.ai_socktype = SOCK_STREAM; // socket flux connectée
hints.ai_flags = AI_PASSIVE; // toutes les addresses

// on effectue la demande pour un certain le port
res = getaddrinfo(NULL, port, &hints, &addr);
if (res != 0) exit_error("getaddrinfo");
```

Si l'appel réussi, alors `addr` pointe désormais vers le premier élément d'une liste chaînée de `struct addrinfo`, et le chaînage est établie par le champ `ai_next` ; il faut parcourir la liste d'adresses possibles jusqu'à en trouver une que l'on arrive à associer à une socket avec `bind()`.

Mise en place côté serveur

Il faut une socket qui ne sert que pour être contactée par le(s) client(s).

- Les paramètres sont :
  - le port utilisé ;
  - la liste des adresses possibles (par défaut, toutes).
- Le résultat est une socket d'écoute.
- Elle ne permet pas de communiquer.

En C POSIX, il plusieurs fonctions doivent être utilisées à la suite :

- `int getaddrinfo(...)`
- `int socket(int domain, int type, int protocol)`
  ↪ crée le descripteur de fichier.
- `int bind(int s, const struct sockaddr *addr, socklen_t len)`
  ↪ réserve un port réseau de la machine.
- `int listen(int s, int bl)`
  ↪ transforme la socket en socket d'écoute.

On suppose que l'on a déjà obtenu une liste d'adresses `addr` :

```
struct addrinfo *p = addr; // pour parcourir la liste
int s;                     // socket d'écoute
int sd;                    // socket de dialogue

while (p != NULL) { // on parcourt la liste
  // tentative de création de la socket serveur
  s = socket(p->ai_family, p->ai_socktype,p->ai_protocol);
  if (s != -1) { // la création de la socket a fonctionné
    // on essaye de réserver le port
    res = bind(s, p->ai_addr, p->ai_addrlen);
    if(res == 0 ) break; // la socket est àl'écoute
    else close(s);
  }
  p = p->ai_next; // on passe àl'adresse suivante
}
if(p == NULL) exit_error("impossible de réserver le port");

res = listen(s, 5);
if(res == -1) exit_error("listen");
```

Le serveur doit créer une socket de dialogue pour chaque nouveau client.

- les données envoyées par le client seront lues depuis cette socket ;
- les données écrites sur la socket seront envoyées à ce client particulier.

En C POSIX, on utilise :

```
int accept(int s, struct sockaddr *addr, socklen_t *len);
```

- retourne la socket de dialogue (descripteur de fichier), ou -1 en cas d'erreur.
- `addr` (*résultat*) : permet d'obtenir l'adresse du client.
- `len` (*résultat*) : la longueur de `adresse`.

Fin de l'exemple côté serveur :

```
int sd; // socket de dialogue
struct sockaddr_storage addr;
socklen_t addr_len = sizeof(struct sockaddr_storage);

// on reste en attente de la connexion d'un client
// sur la socket d'écoute s
sd = accept(s, (struct sockaddr*) &addr, &addr_len);
if(sd == -1) exit_error("sd");
```

L'appel `sd = accept(s, ...)` est bloquant : le processus ne reprend que lorsqu'un client est connecté, et alors `sd` permet le dialogue.

Si la connexion du client réussi, la structure `addr` résultant de l'appel à `accept` permet de réccupérer des informations sur le client, notamment son adresse (grâce à `getnameinfo()`).

Mise en place côté client

Le client doit se connecter à un port du serveur et lui demander de créer une socket de dialogue.

- Si le port est fermé, ou qu'aucun processus n'y est à l'écoute, la requête sera refusée par le système de la machine contactée.
- Les paramètres nécessaires sont le nom et le port du serveur.
- Le résultat est une socket de dialogue.

En C POSIX, il faut encore plusieurs fonctions :

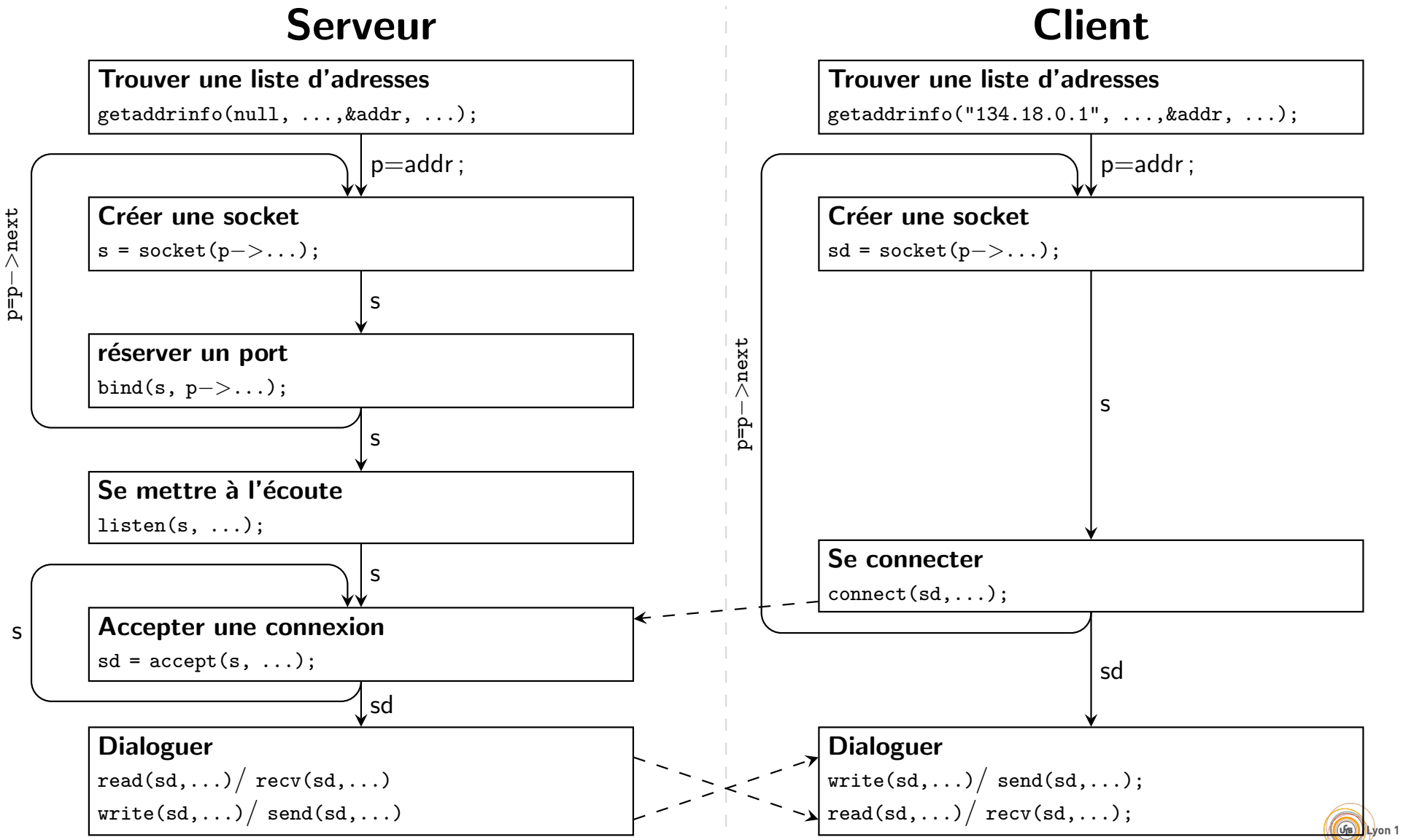
- `int getaddrinfo(...)`
- `int socket(int domain, int type, int protocol)`
  ↪ pour créer la socket.
- `int connect(int s, const struct sockaddr *adr, socklen_t l)`
  ↪ pour se connecter au serveur, et obtenir la socket de dialogue.

On a obtenu avec `getaddrinfo()` une liste d'adresses `addr` pour le serveur :

```
int sd; // socket de dialogue
struct addrinfo *p = addr;
while(p != NULL) {
  // on essaye d'abord de créer la socket demandée
  sd = socket(p->ai_family, p->ai_socktype,p->ai_protocol);
  if(sd != -1) { // la création de socket a réussi
    // on essaye de se connecter àl'hôte distant
    res = connect(sd, p->ai_addr, p->ai_addrlen);
    if(res == 0 ) break; // cela a fonctionné
    else close(sd);     // le connect a échoué
  }
  p = p->ai_next; // on passe àl'adresse suivante
}
// si p == NULL, on est sorti sans réussir
if(p == NULL) exit_error("échec de la connexion")
```

Si on sort de la boucle avec `p != NULL`, alors on est connecté, et on peut utiliser `sd` pour dialoguer avec le serveur.

En résumé


Serveur
Trouver une liste d'adresses
getaddrinfo(null, ...,&addr, ...);
p=addr;
p=p->next
Créer une socket
s = socket(p->...);
s
réserver un port
bind(s, p->...);
s
Se mettre à l'écoute
listen(s, ...);
s
s
Accepter une connexion
sd = accept(s, ...);
sd
Dialoguer
read(sd,...)/ recv(sd,...)
write(sd,...)/ send(sd,...)
Client
Trouver une liste d'adresses
getaddrinfo("134.18.0.1", ...,&addr, ...);
p=addr;
p=p->next
Créer une socket
sd = socket(p->...);
s
Se connecter
connect(sd,...);
sd
Dialoguer
write(sd,...)/ send(sd,...);
read(sd,...)/ recv(sd,...);

Une fois la connexion établie, les processus peuvent envoyer des données grâce à `read()` et `write()`, mais aussi grâce à des primitives dédiées :

- `ssize_t recv(int sd, void *buf, ssize_t len, int flags);`
- `ssize_t send(int sd, const void *buf, size_t len, int flags);`
  - `sd` : la socket de dialogue,
  - `buf` : les données, au plus de taille `len`,
  - `flags` : des options...

Lire des données est une opération bloquante et en envoyer peut aussi l'être. Attention aux interblocages !

Les fonctions d'ouverture des sockets (surtoût côté serveur) sont complexes à manipuler.

- Un simple serveur en C demande 120 lignes de code...
- La plupart des exemples « sur internet » utilisent du code qui est incompatible avec IPv6.
- Certaines fonctions n'ont *pas d'intérêt*, elle existent plus que pour des raisons historiques.

Pour simplifier, nous utiliserons en TP une bibliothèque *ad hoc* pour :

- créer une socket serveur,
- accepter les connexions entrantes côté serveur,
- demander une connexion côté client.

Position du problème

Une grosse part des difficultés dans l'utilisation des sockets vient du transfert de données. Il n'y a pas de méthode applicable dans tous les cas :

- transfert de texte, données numérique, données binaires, ...
- données plus au moins importantes (messages, page web, ...)

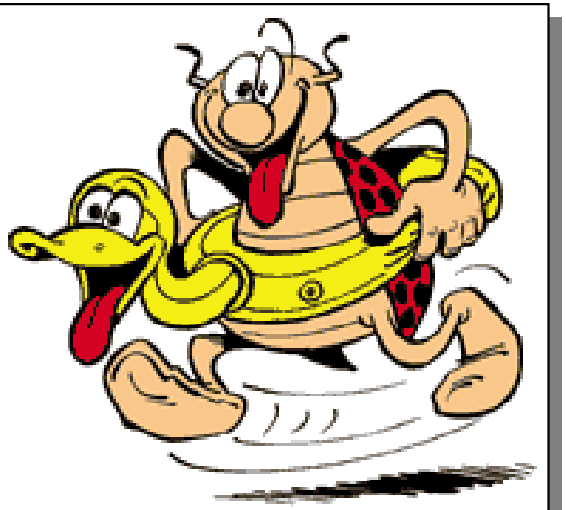
Il y a deux niveaux de programmation pour le transfert de données :

1. Utiliser les primitives de base pour créer des primitives plus complexes : entiers, textes, données binaire de tailles connues...
2. Utiliser les primitives complexes pour implémenter un véritable protocole de communication.

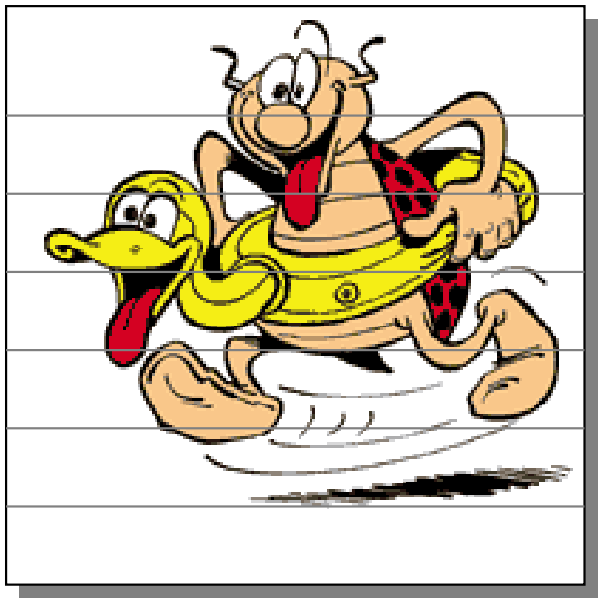
On va utiliser en TD/TP les primitives POSIX :

- `recv()`, `read()` pour lire des octets.
- `send()`, `write()` pour écrire des octets.

Exemple d'un transfert d'image : quand le réseau est peu chargé.



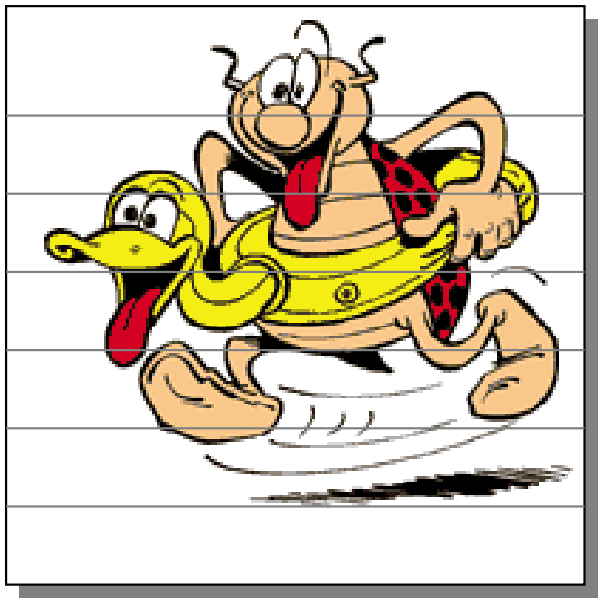
Exemple d'un transfert d'image : quand le réseau est peu chargé.

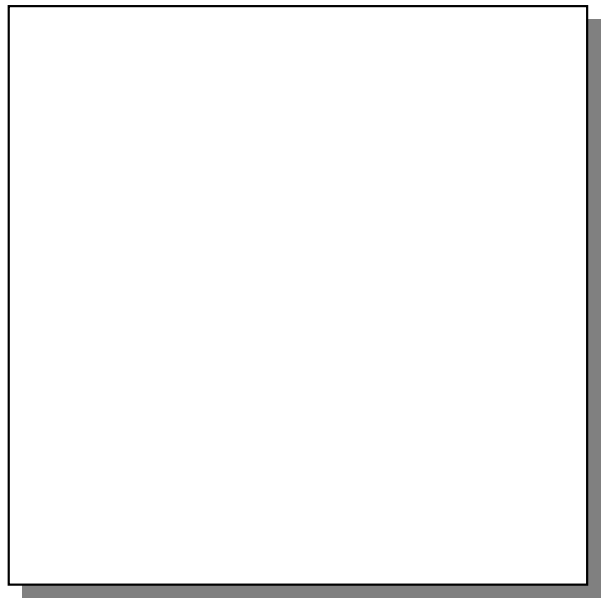


```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Exemple d'un transfert d'image : quand le réseau est peu chargé.

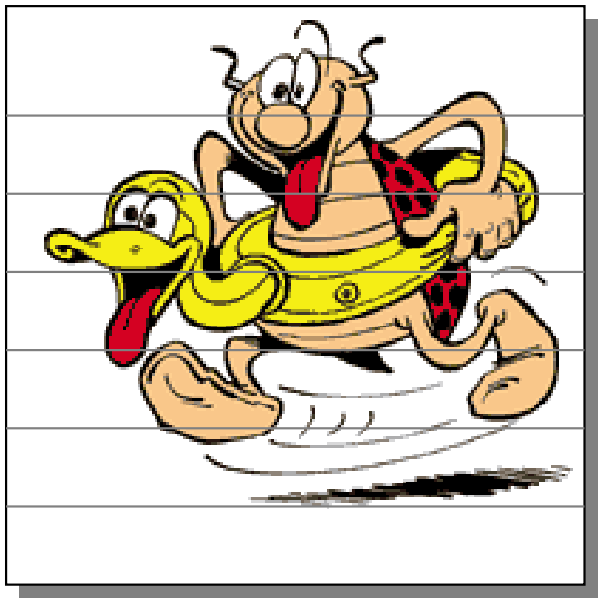


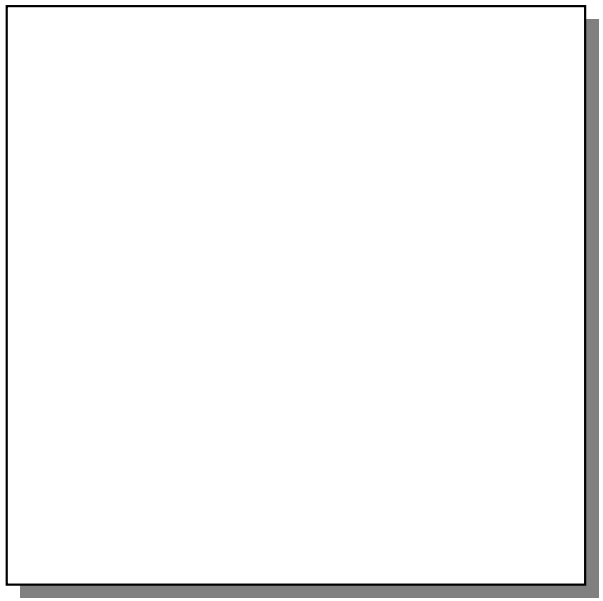


```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Exemple d'un transfert d'image : quand le réseau est peu chargé.

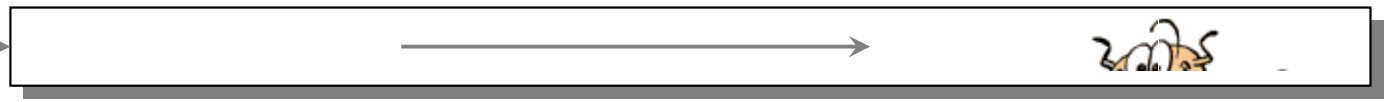




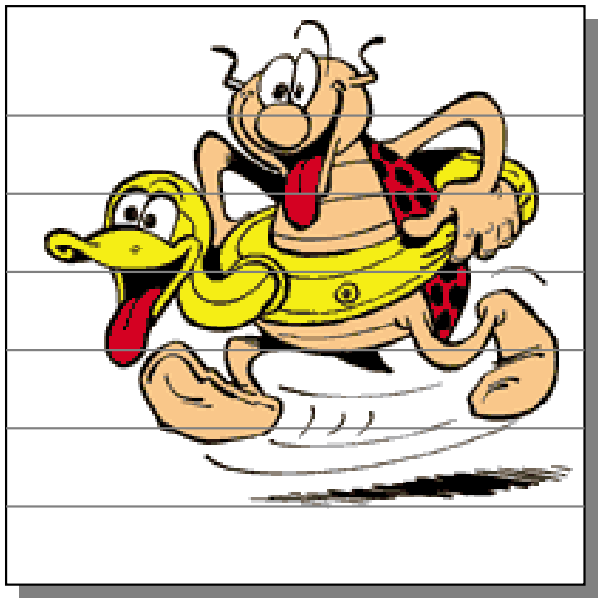
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

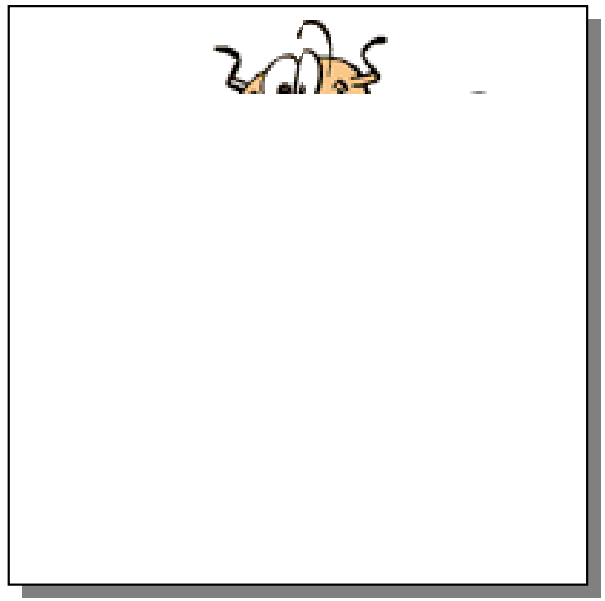
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

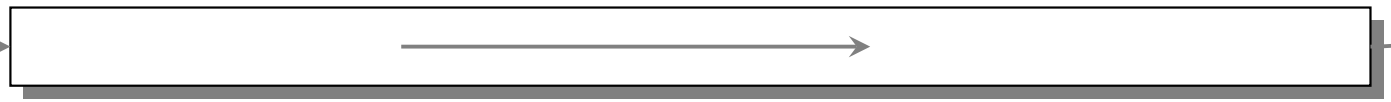




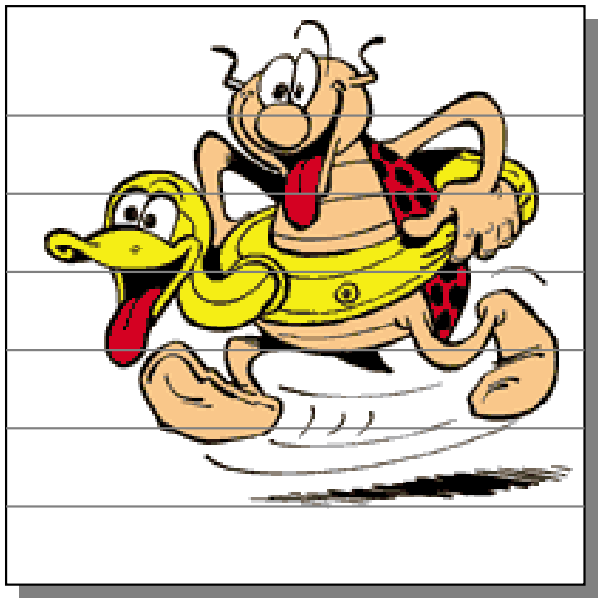
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

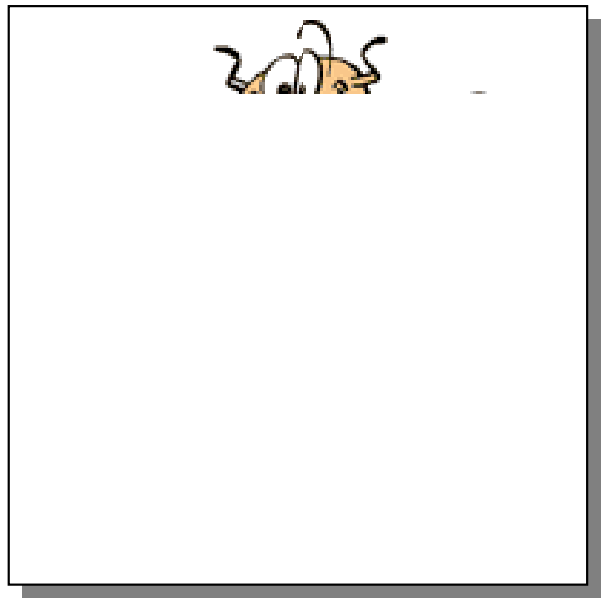
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

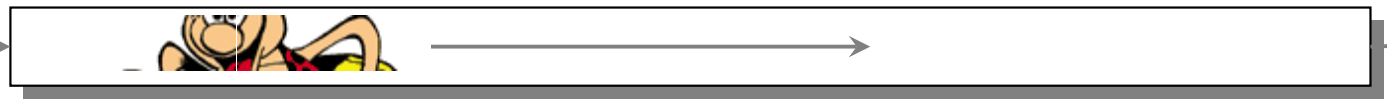




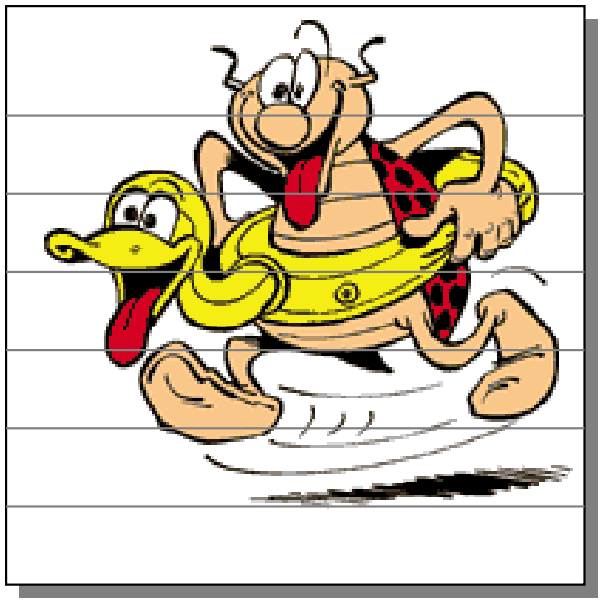
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

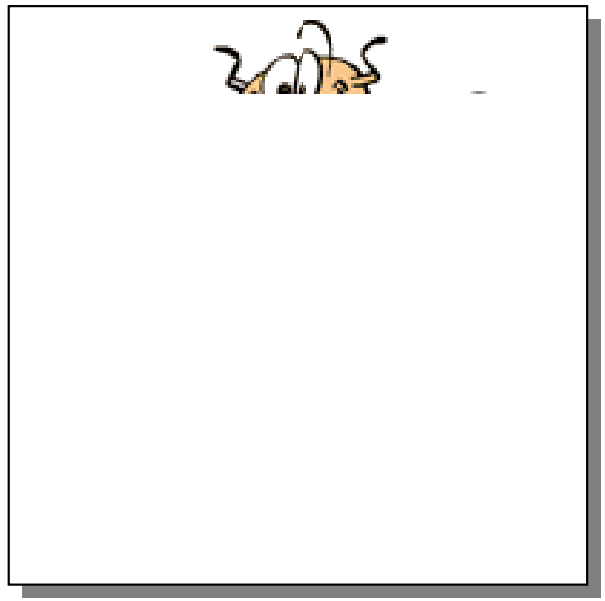
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

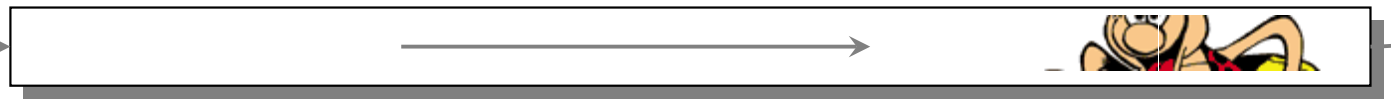




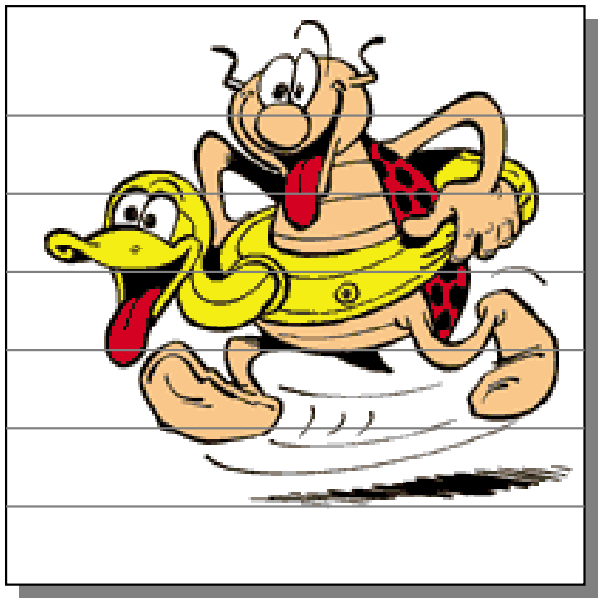
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

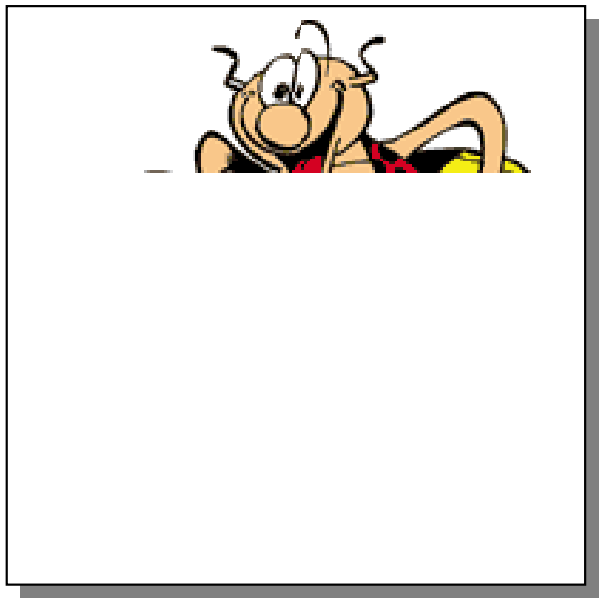
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

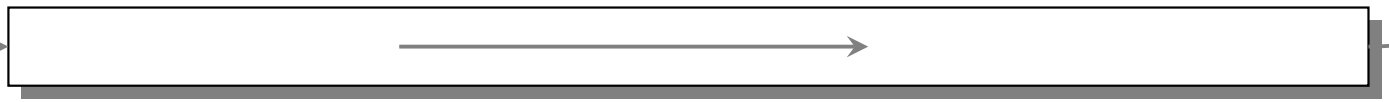




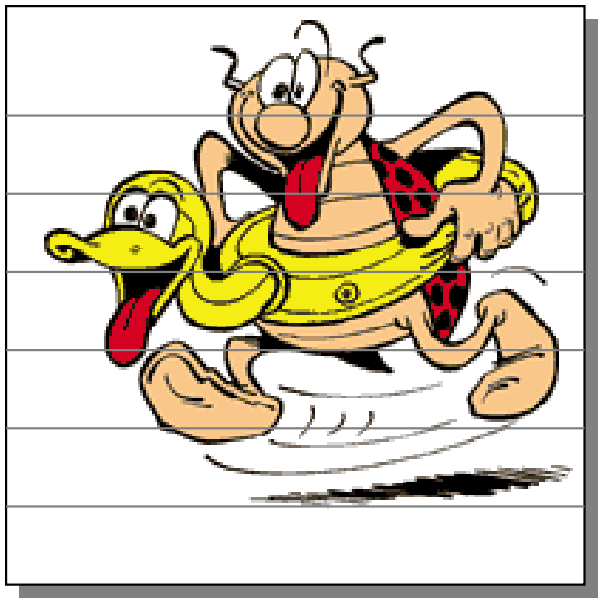
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

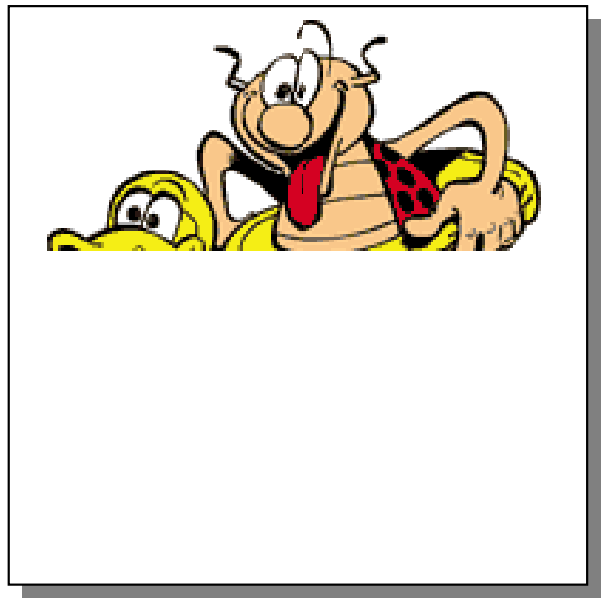
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

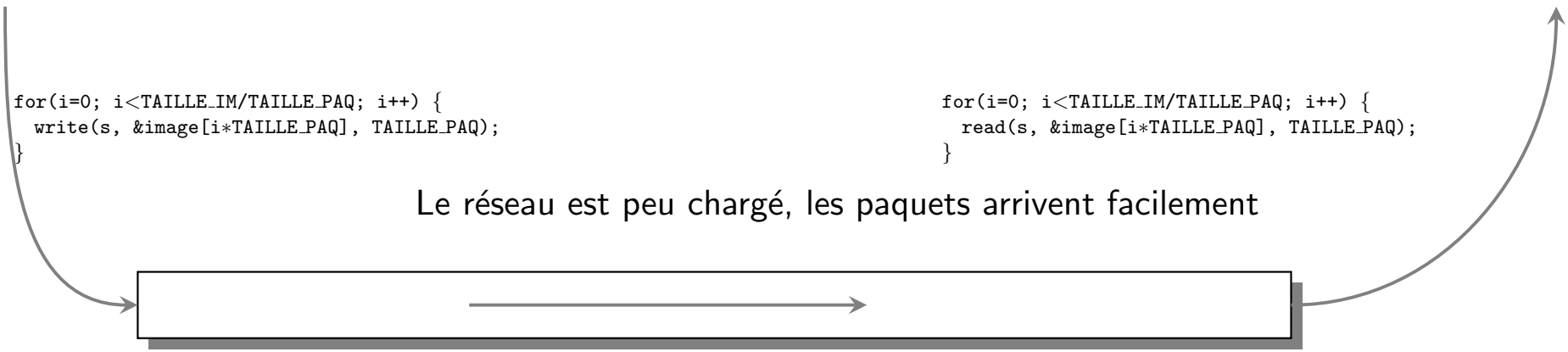




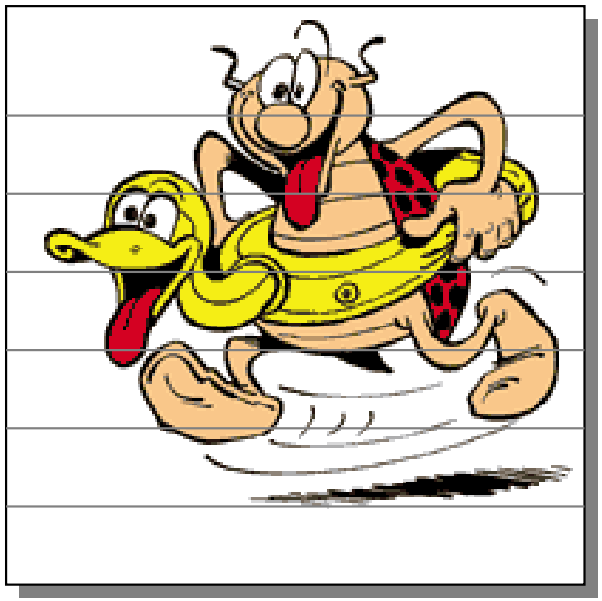
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

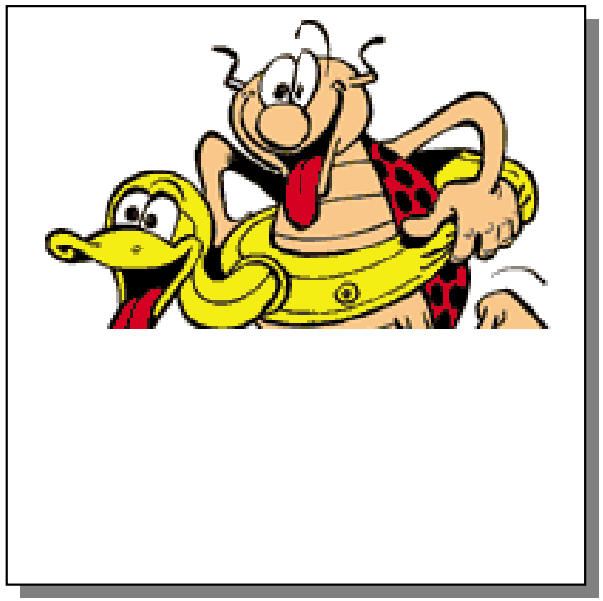
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

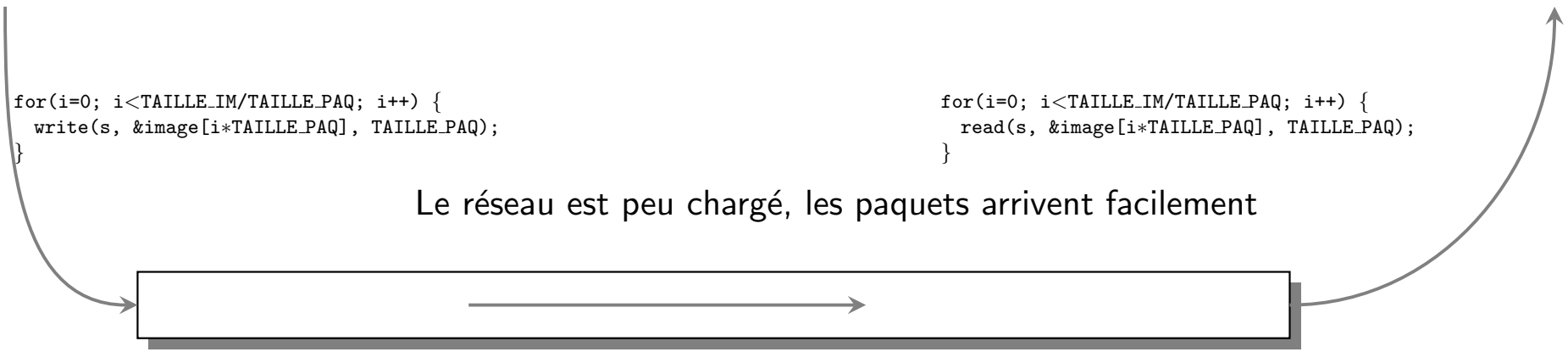




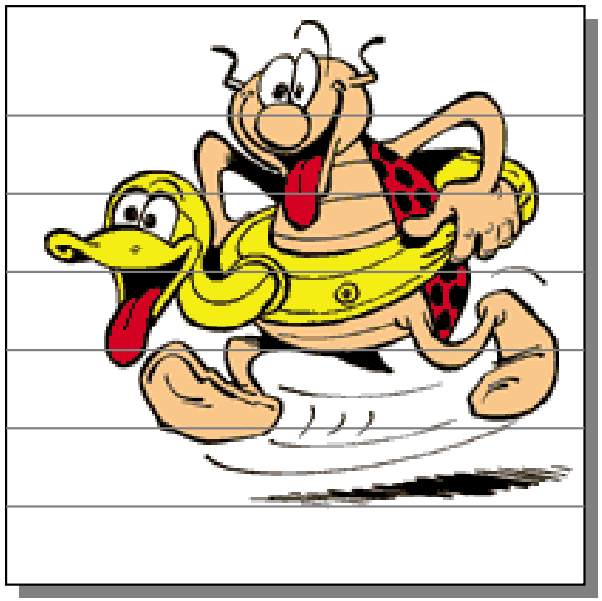
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

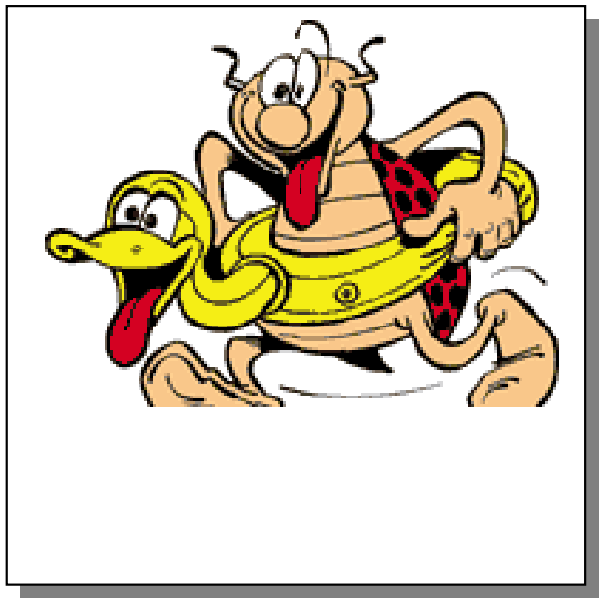
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

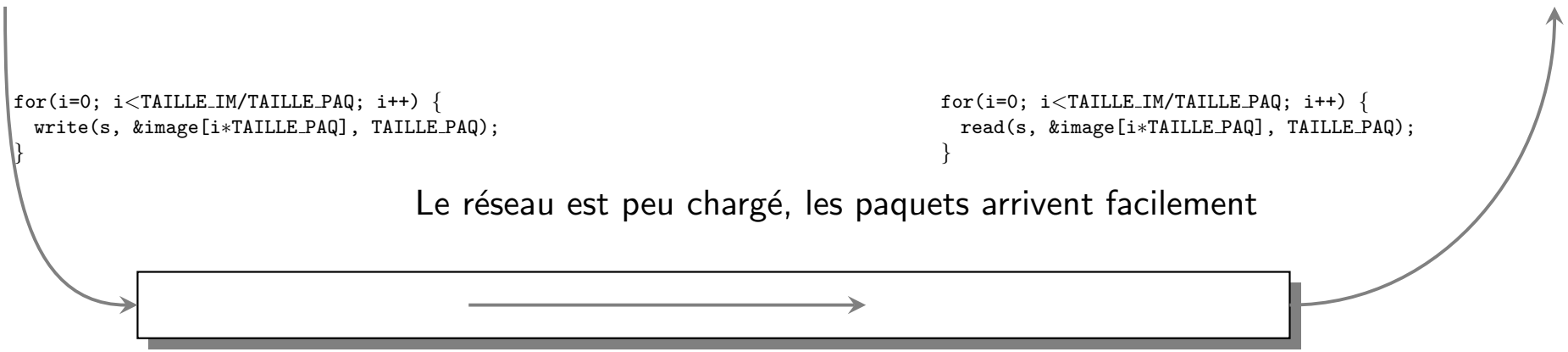




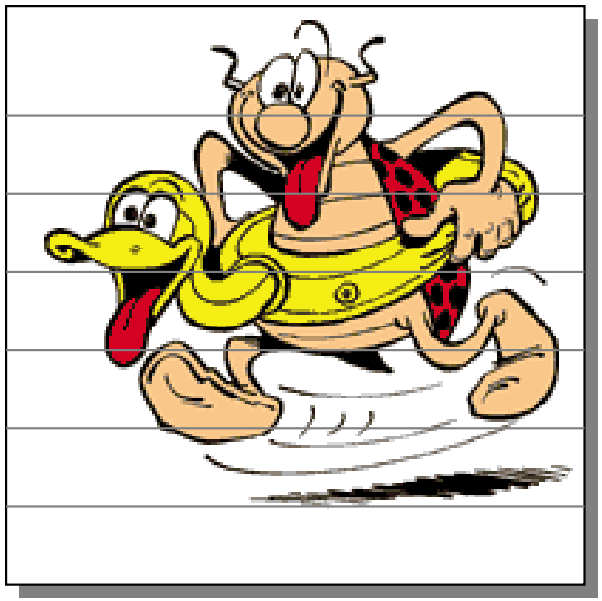
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

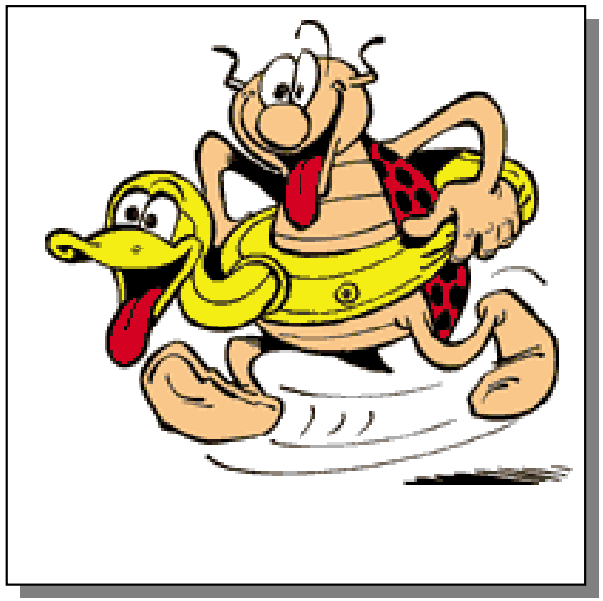
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

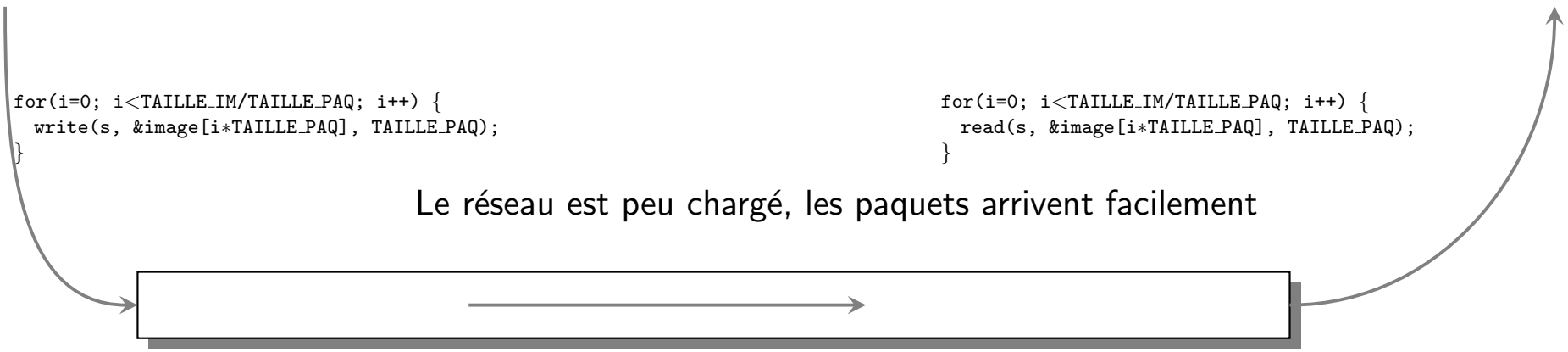




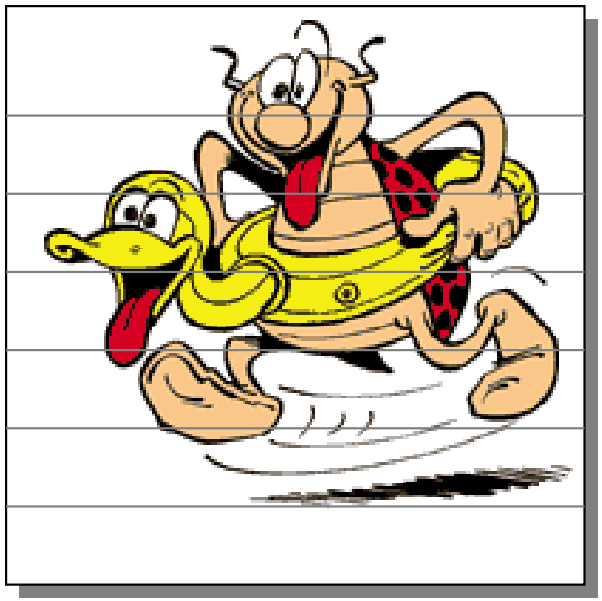
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

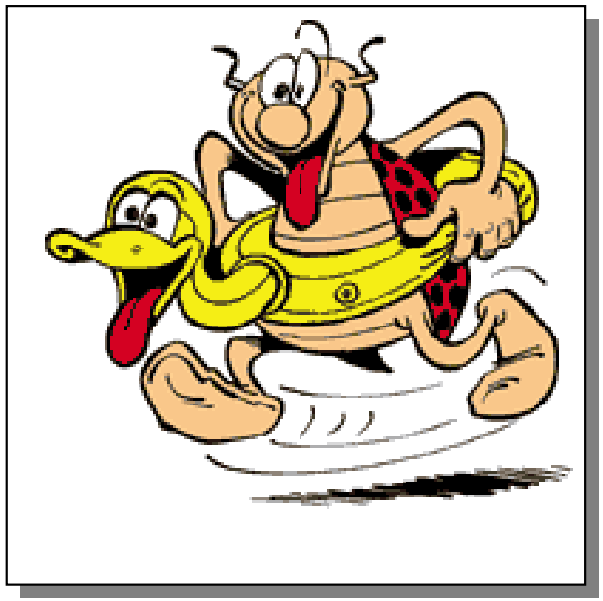
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est peu chargé.

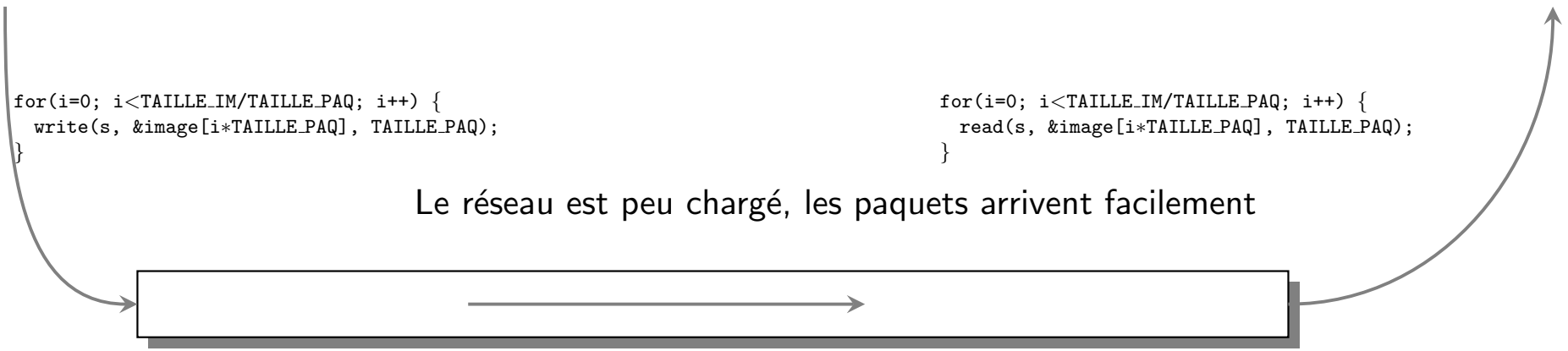




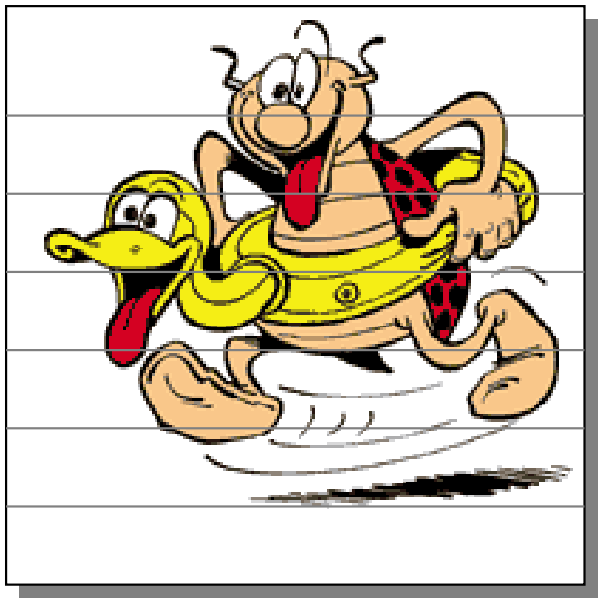
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

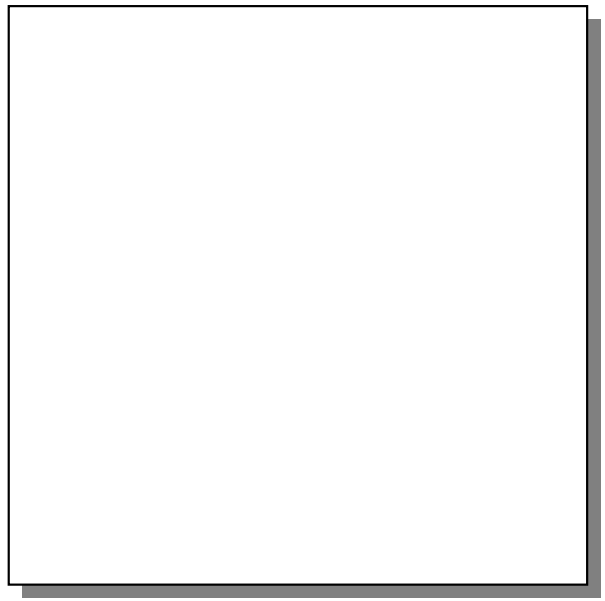
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est peu chargé, les paquets arrivent facilement



Exemple d'un transfert d'image : quand le réseau est très chargé.

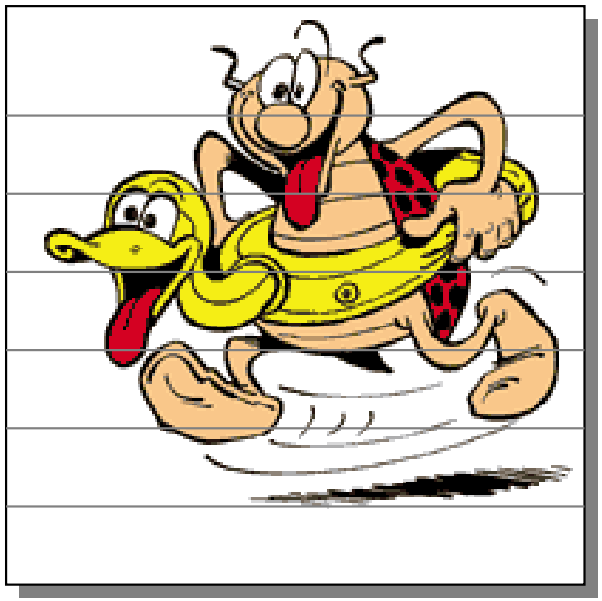


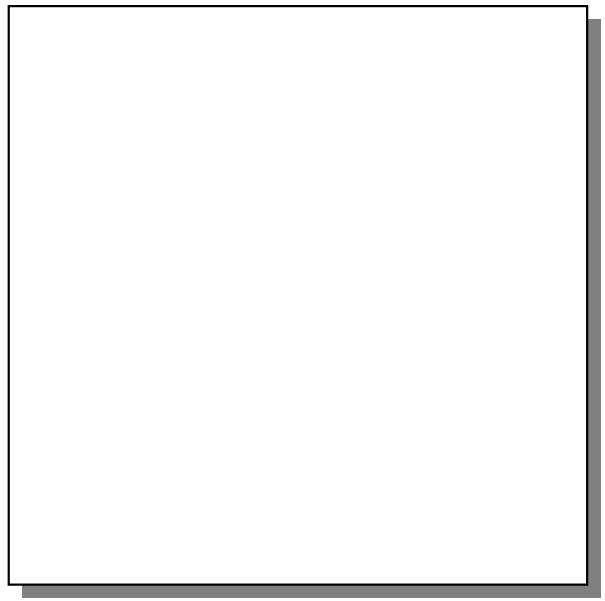


```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

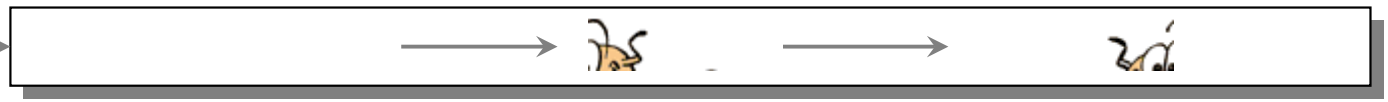
Exemple d'un transfert d'image : quand le réseau est très chargé.



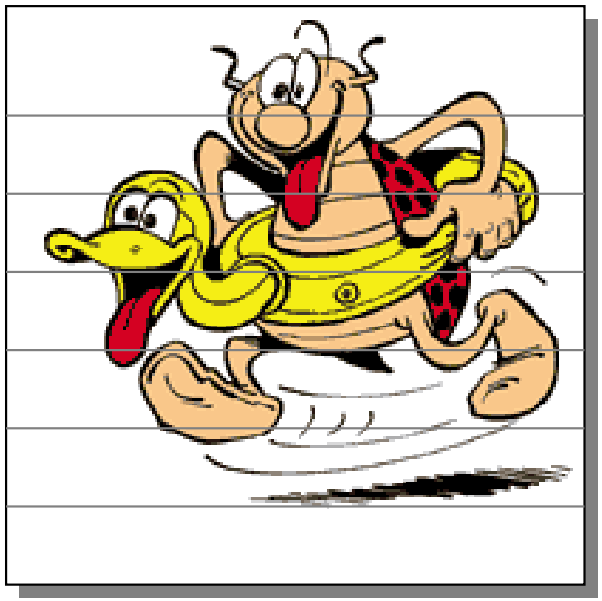


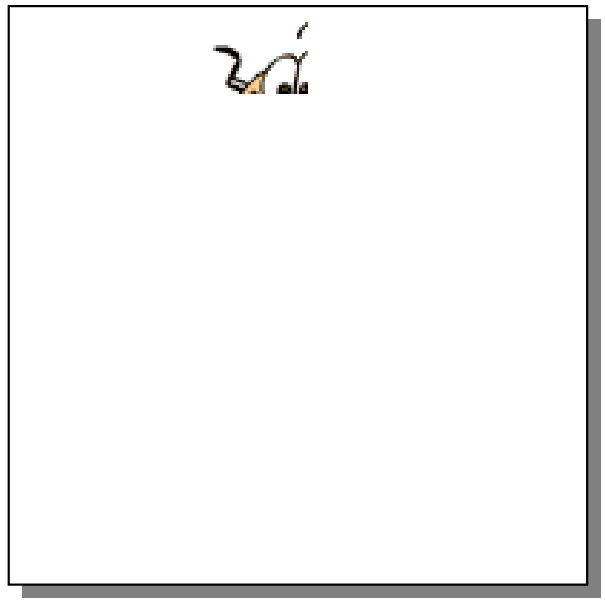
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```



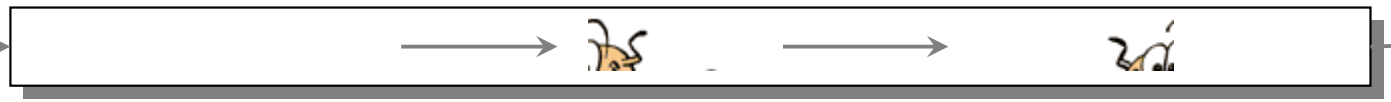
Exemple d'un transfert d'image : quand le réseau est très chargé.



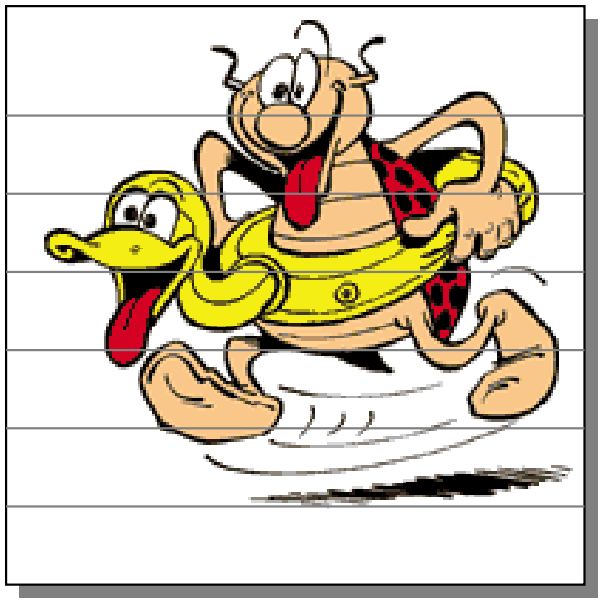


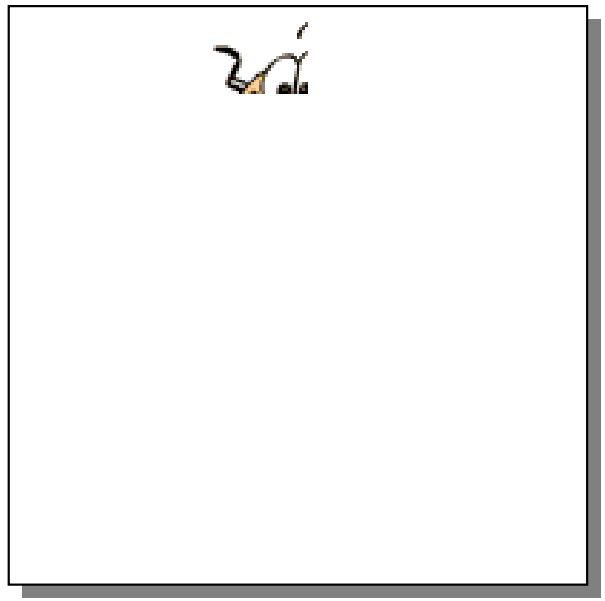
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```



Exemple d'un transfert d'image : quand le réseau est très chargé.



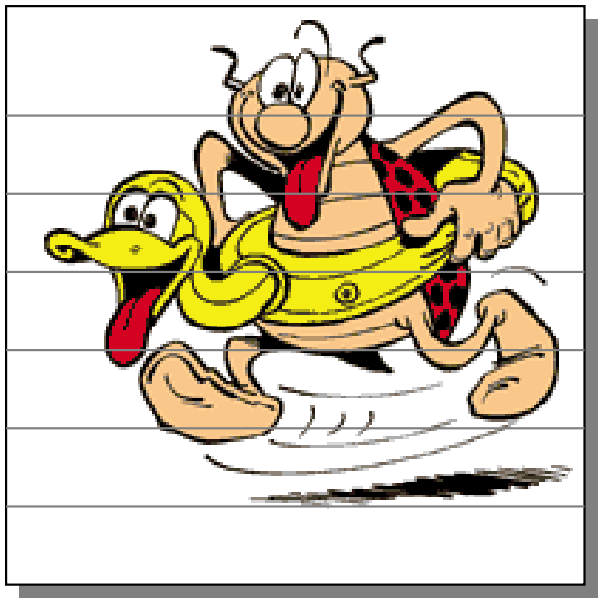


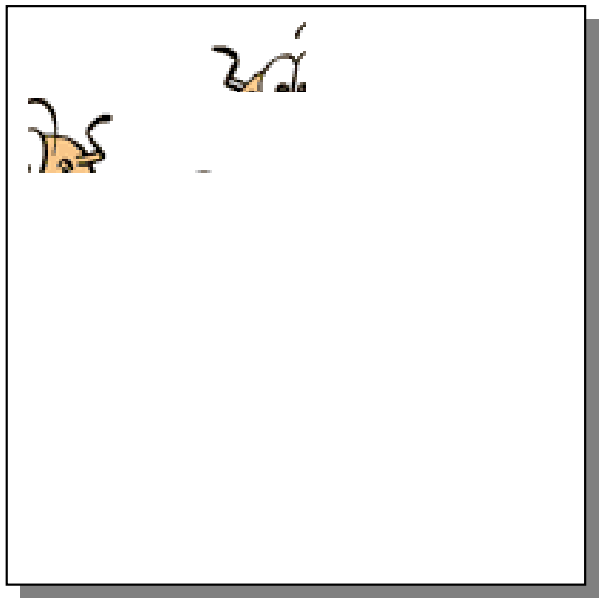
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



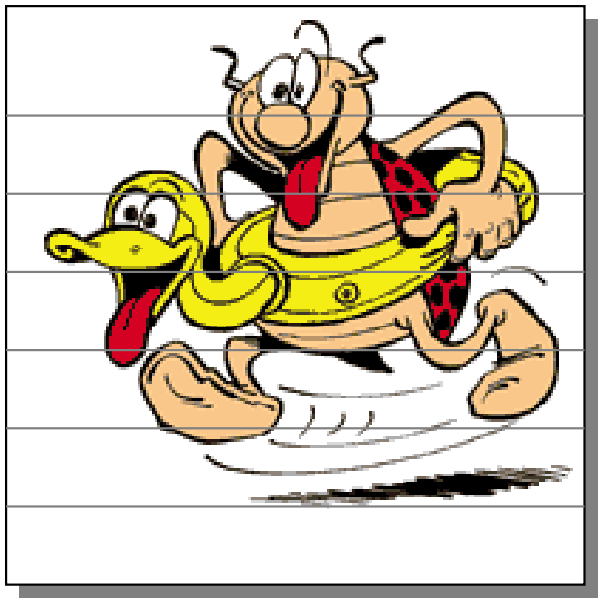


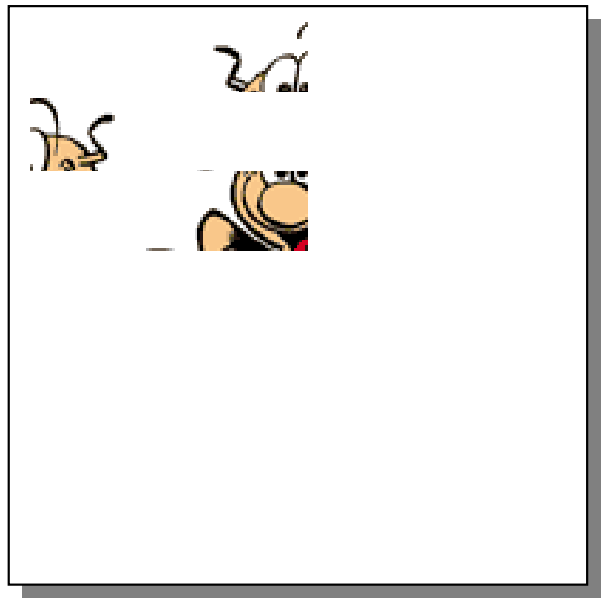
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



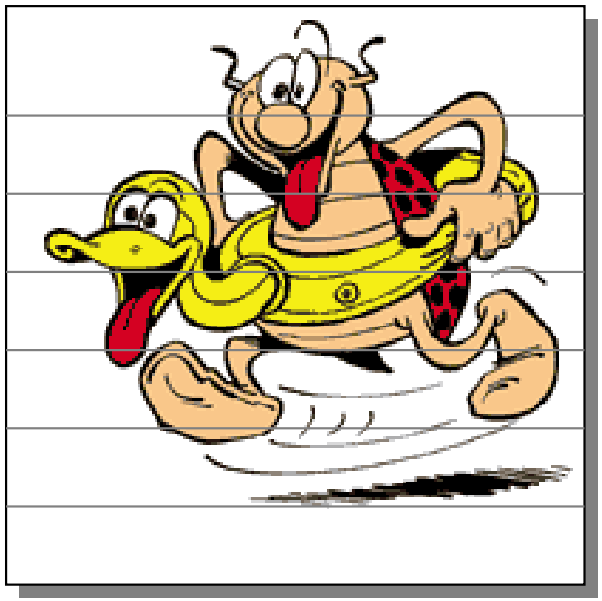


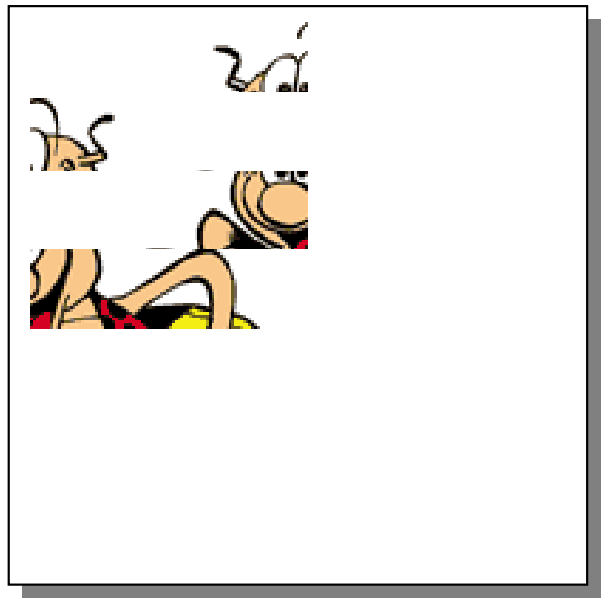
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



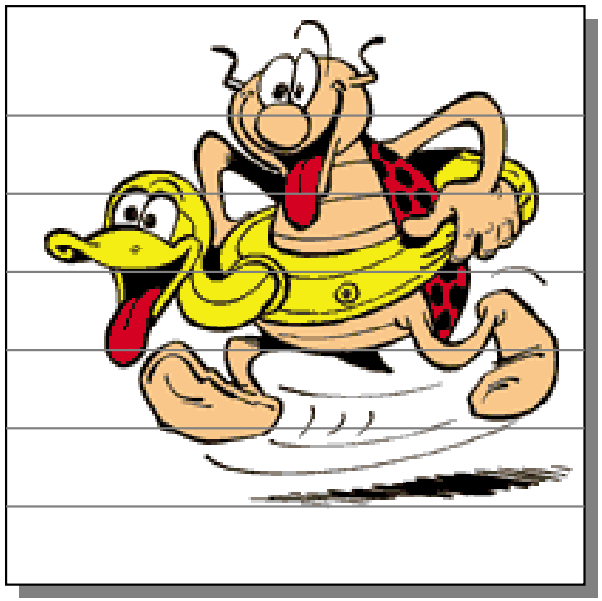


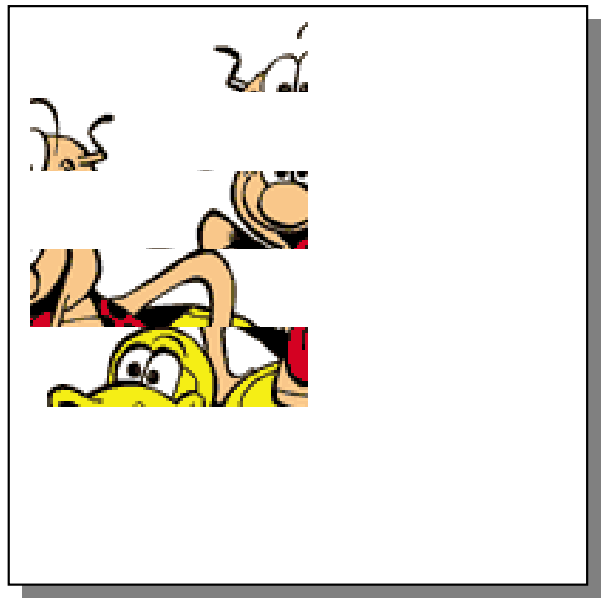
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



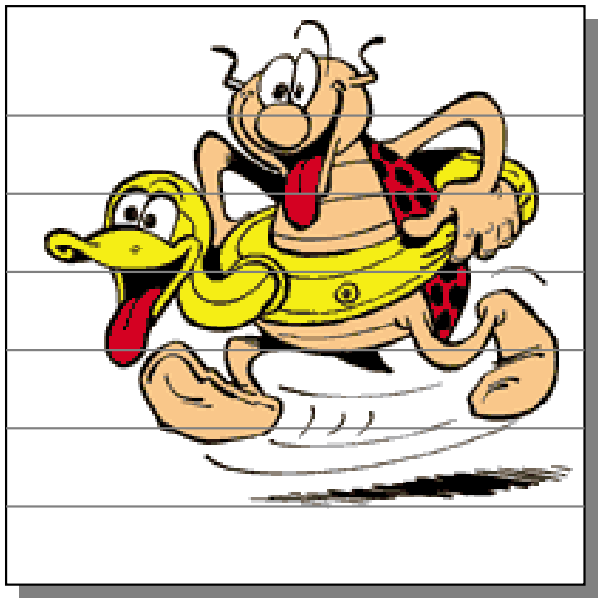


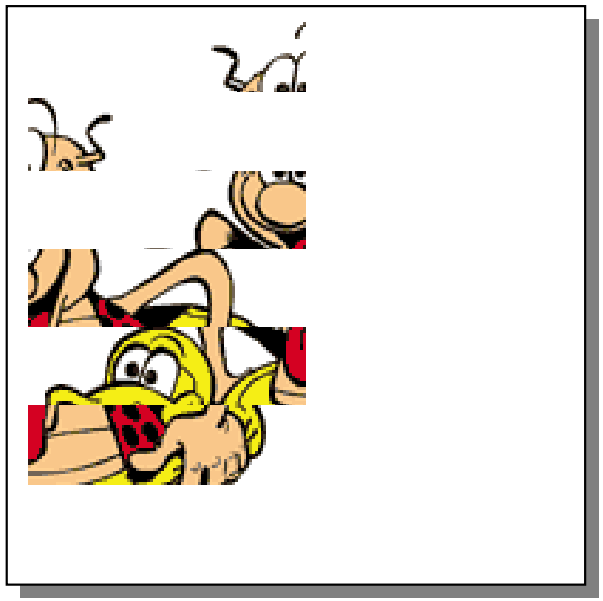
```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



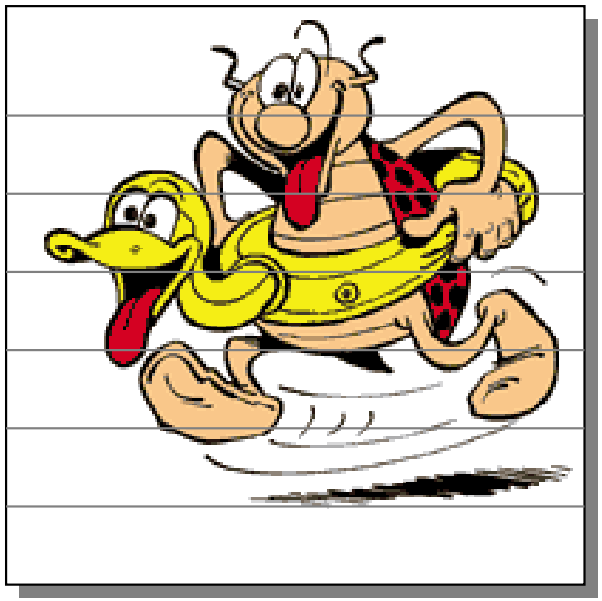


```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Exemple d'un transfert d'image : quand le réseau est très chargé.



```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  write(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

```
for(i=0; i<TAILLE_IM/TAILLE_PAQ; i++) {
  read(s, &image[i*TAILLE_PAQ], TAILLE_PAQ);
}
```

Le réseau est chargé, les paquets sont coupés en deux, mais le récepteur ne le sait pas.

Le programme précédent est buggé, mais le bug

- n'est visible que sur de gros transferts, jamais lors de tests simples ;
- n'apparaît que dans des conditions de « stress » du réseau.

↪ bug difficile à identifier et à corriger.

Messages

Le transfert d'un flux d'octets n'est pas naturel, et cause de nombreuses erreurs :

- il faut vérifier que tout le message est arrivé ;
- il faut vérifier qu'on ne déborde pas sur le message suivant.

Pour éviter les erreurs, il faut définir un protocole de transfert qui permet de retrouver les frontières des messages.

Pour réussir un échange de données, il faut :

- empêcher l'environnement de perturber la communication.
  - Il faut implémenter des primitives robustes.
  - Ex : `vector<char> read_all(int fd, int n)` pour lire `n` octets exactement sur le descripteur de fichier `fd`.
  - Attendre si moins d'octets arrivent, les laisser de côté s'il y en a trop.

- s'assurer à tout moment que chaque processus sait ce qu'il a à faire.
  - Envoyer ou lire des données.
  - La taille des données ou le moyen de stopper la lecture.
  - Si on ne le fait pas, on risque l'interblocage.
  - C'est le rôle du protocole de communication.

Exemple : le protocole HTTP (1.1) permet au client (un navigateur) de télécharger le contenu d'une page web.

- Le client parle en premier : il envoie une requête accompagnée de lignes d'entête. Par exemple :

```
GET /nicolas.louvet/index.html HTTP/1.1
Host:perso.ens-lyon.fr
Connection: Close
```

  Remarque : chaque ligne se termine par `CRLF`.
- Le serveur répond avec un entête sur le même modèle qui mentionne la méthode pour lire la page, puis il envoie la page.
- Dès que la page a été envoyée, le serveur se met en attente d'une nouvelle requête, et le client peut en envoyer une autre.

À chaque instant, le client et le serveur savent ce qu'ils doivent lire ou envoyer. Tout décalage provoque une erreur de lecture ou un blocage.

Quelques pièges classiques

1er exemple : un serveur attent un connexion, lit par bloc d'au plus 13 caractères sur la socket de dialogue en affichant les blocs lus :

```
while(1) {
    char buf[13];
    res = read(sd, buf, 13);
    if(res == 0) break; // reception de EOF
    if(res < 0) exit_error("read");
    printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Un client tente d'envoyer une chaîne :

```
nlouvet:~/$ echo -n "toto fait du velo" | nc localhost 8083 -N
```

Le serveur affiche :

```
Lecture de 13 octets : *toto fait du *
Lecture de 4 octets : *velo fait du *
```

2ème exemple : Dans le 1er exemple, on ne plaçait pas de '\0' en fin de la chaîne à afficher, d'où l'erreur. Corrigeons :

```
#define LEN 13
while(1) {
    char buf[LEN];
    res = read(sd, buf, LEN-1);
    if(res == 0) break; // reception de EOF
    if(res < 0) exit_error("read");
    buf[res-1] = '\0';
    printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Un client envoye une chaîne :

```
nlouvet:~/$ echo -n "toto fait du velo" | nc localhost 8083
```

Le serveur affiche :

```
Lecture de 12 octets : *toto fait du*
Lecture de 5 octets : * velo*
```

C'est déjà plus satisfaisant, mais pas forcément parfait !

On veut envoyer le fichier suivant au serveur :

```
nlouvet:~/$ cat test.txt
certains mots disparaissent dans les réseaux
```

On fait donc :

```
nlouvet:~/$ cat test.txt | nc localhost 8083 -N
```

Le serveur affiche :

```
Lecture de 12 octets : *certains mot*
Lecture de 12 octets : *s*              <- !!!
Lecture de 12 octets : *sent dans le*
Lecture de 11 octets : *s réseaux
*
```

Indication :

```
nlouvet:~/$ xxd test.txt
00000000: 6365 7274 6169 6e73 206d 6f74 7300 2064 certains mots. d
00000010: 6973 7061 7261 6973 7365 6e74 2064 616e isparaissent dan
00000020: 7320 6c65 7320 72c3 a973 6561 7578 0a s les r..seaux.
```

Un dernier pour la route : On veut échanger des messages de 4 caractères, donc on utilise l'option `MSG_WAITALL` de `recv()` :

```
#define LEN 4
while(1) {
  char buf[LEN+1];
  res = recv(sd, buf, LEN, MSG_WAITALL);
  if(res == 0) break;
  if(res < 0) exit_error("read");
  buf[res] = '\0';
  printf("Lecture de %d octets : *%s*\n", res, buf);
}
```

Le client envoye :

```
nlouvet:~/$ echo -n "héhé" | nc localhost 8083
```

Le serveur affiche :

```
Lecture de 4 octets : *héh*
Lecture de 2 octets : *é*
```

C'est encore une histoire d'encodage des caractères !

Comment s'y prendre ?

Grâce à TCP, vous savez que tous les octets arrivent dans le bon ordre, sinon une erreur est détectée. Mais vous ne savez pas :

- si un envoi arrive en un seul paquet ;
- si plusieurs envois arrivent dans un même paquet ;
- quelle est la taille de l'information à lire ;
- quel processus doit lire et à quel moment doit-il le faire ;
- quand doit-il s'arrêter ;
- si les deux programmes utilisent la même façon de coder les choses (à part pour les caractères sur 1 octet).

Il est nécessaire de définir un protocole de communication :

- savoir quand lire et quand écrire,
- savoir comment détecter la fin d'un message,
- savoir quelles informations sont échangées

Conclusion

Moyen de communication entres processus

Nous avons vu :

- les fichiers réguliers,
- les signaux (messages simples),
- les tubes anonymes ou nommés (entre processus d'un même système),
- les sockets (échanges via le réseau).

Nous n'avons pas parlé des systèmes de fichier réseau (comme NFS), des segments mémoire partagés, ou des Remote Procedure Calls (RPC)...

Difficultés

- Choisir la structure d'un fichier (fichiers réguliers).
- Définir un protocole de communication (tubes ou sockets).