

Programmation Système et Réseaux

Christophe LOHR

Automne 2018

Sommaire

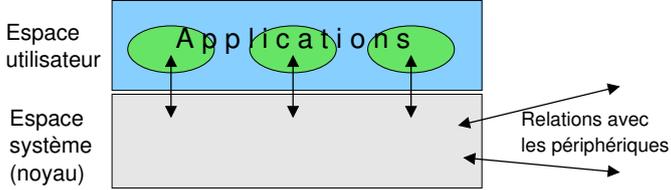
1	Les outils pour la programmation système	3
1.1	Appels systèmes, fonctions, libc	3
2	Les processus	10
2.1	Création, environnement, signaux, terminaison	10
3	Les entrées sorties	32
3.1	Descripteurs et pointeurs de fichiers, primitives et fonctions	32
4	Structure d'un logiciel Unix/Linux	41
4.1	Compilation, édition de liens, bibliothèques	41
5	Les outils d'aide à la mise au point	48
5.1	Débugage, tracement, profilage	48
6	L'utilitaire Make	50
6.1	automatisation des phases de compilation	50
7	Paquetages logiciels : rpm, debian, Gnu tar	60
7.1	gnu tar, debian, red hat, etc.	60
8	Programmation d'applications Réseau	64
8.1	Concepts généraux	64
8.2	L'API Socket	67
8.3	L'API RPC	104

1 Les outils pour la programmation système

1.1 Appels systèmes, fonctions, libc

Les applications et le système 3/169

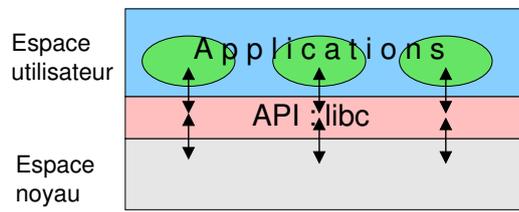
- ▶ Les applications en cours et le système résident en mémoire centrale
- ▶ La mémoire est divisée en deux parties :
 - ▶ L'espace système : le noyau
 - ▶ L'espace utilisateur : où résident les applications



The diagram illustrates the memory layout. It is divided into two horizontal sections. The top section, labeled 'Espace utilisateur' (User Space), is light blue and contains three green ovals labeled 'Applications'. The bottom section, labeled 'Espace système (noyau)' (System Space / Kernel), is light grey. Three vertical double-headed arrows connect the 'Applications' layer to the 'Espace système' layer. To the right of the 'Espace système' layer, two arrows point outwards, labeled 'Relations avec les périphériques' (Relations with peripherals).

La programmation système 4/169

- ▶ C'est le développement d'applications en utilisant les ressources et les outils fournis par le système
 - ▶ Utilisation de fonctions standards fournies avec le système
 - ▶ la bibliothèque standard du langage C pour Unix, la **libc**
 - ▶ Dialogue avec le noyau et contrôle de ce dialogue, utilisation des ressources du noyau



This diagram shows the interaction between user space and kernel space. It is divided into three horizontal sections. The top section, 'Espace utilisateur' (User Space), is light blue and contains three green ovals labeled 'Applications'. The middle section, 'API : libc', is light red. The bottom section, 'Espace noyau' (Kernel Space), is light grey. Vertical double-headed arrows connect the 'Applications' layer to the 'API : libc' layer, and the 'API : libc' layer to the 'Espace noyau' layer.

- ▶ Il existe différentes versions de libc
- ▶ Sous linux aujourd'hui nous en sommes à la version 6 : libc-6
 - ▶ La libc-6 est en réalité d'origine GNU, c'est la glibc-2 :-)
 - ▶ La libc-5 et la glibc sont différentes :-)
- ▶ Les pages du manuel de référence via la commande `man` ne sont pas obligatoirement à jour
 - ▶ Préférer la commande `info` qui lit des pages dans un format particulier appelé `textinfo`, ces pages sont normalement à jour...

GNU (jeu de mots récursif (!) : Gnu is Not Unix) est une émanation de la FSF (Free Software Foundation) fondée par Richard Stallman (grand programmeur s'il en est, le père de emacs, ...) au début des années 90. Le but premier de la FSF était de créer un nouveau système d'exploitation de type Unix, totalement libre, ainsi que des outils. Le noyau de ce nouveau système existe, il s'appelle Hurd.

Indépendamment de ces travaux et en parallèle, Linux Thorvald développait Linux et toute une communauté de développeur adhérait à ce nouveau système et développait des outils, dont une bibliothèque C de base : la fameuse libc.

Dans le monde du logiciel libre le développement logiciel est de type «bazar» plutôt que de type «cathédrale» (E. S. Raymond, La cathédrale et le bazar, 1998) c'est à dire que le développement logiciel est sans direction et, de ce qui peut sembler un chaos, il en sort le meilleur mais possiblement en plusieurs solutions. Parfois les diverses solutions logicielles coexistent, parfois une est encore meilleure et prend le dessus sur l'autre. Dans le cas de la libc sous Linux il semble que la solution GNU soit la meilleure.

L'effigie de GNU et de la FSF est une tête souriante (mais non moqueuse) de gnou.
Voir le lien www.fsf.org

- ▶ Deux types fondamentaux :
 - ▶ Les appels système
 - ▶ Ce sont les fonctions permettant la communication avec le noyau
 - ▶ Exemples : open, read, write, ioctl, fcntl, etc.
 - ▶ Les fonctions
 - ▶ Ce sont les fonctions standard du C
 - ▶ Exemples : printf, fopen, fread, fwrite, strcmp, etc.

Un appel système est une fonction du plus bas niveau permettant une interaction avec le noyau du système d'exploitation. Certains auteurs appellent ce genre de fonctions des primitives.

Un appel système est une fonction dont la partie principale du corps (le code exécutable) est située dans le noyau. L'implémentation utilisable (en libc) est réduite à une préparation de contexte (le numéro de l'appel système et ses arguments sont placés dans des registres du processeur) et à un appel de l'interruption 0x80 (sur architecture x86).

Voir par exemple :

http://world.std.com/~slanning/asm/daytime_cli.txt

<http://www.ruxcon.org/files/asm.pdf>

Lors d'un appel système, le processeur exécute le code du noyau dans un contexte de processus, le noyau sait de quel processus il s'agit.

Et pour plus d'information : `man libc`, ainsi que `man syscalls`.

Utilisation des appels système

7/169

- ▶ Travaillent en relation directe avec le noyau
- ▶ Rendent un entier positif ou nul en cas de succès et **-1** en cas d'échec
- ▶ Par défaut le noyau peut *bloquer* les appels systèmes et ainsi mettre en attente l'application si la fonctionnalité demandée ne peut être servie immédiatement
- ▶ **Ne peuvent réserver de la mémoire dans le noyau.** Les résultats sont obligatoirement *stockés dans l'espace du processus* (dans l'espace utilisateur), il faut *prévoir cet espace* par allocation de variable en pile ou de mémoire

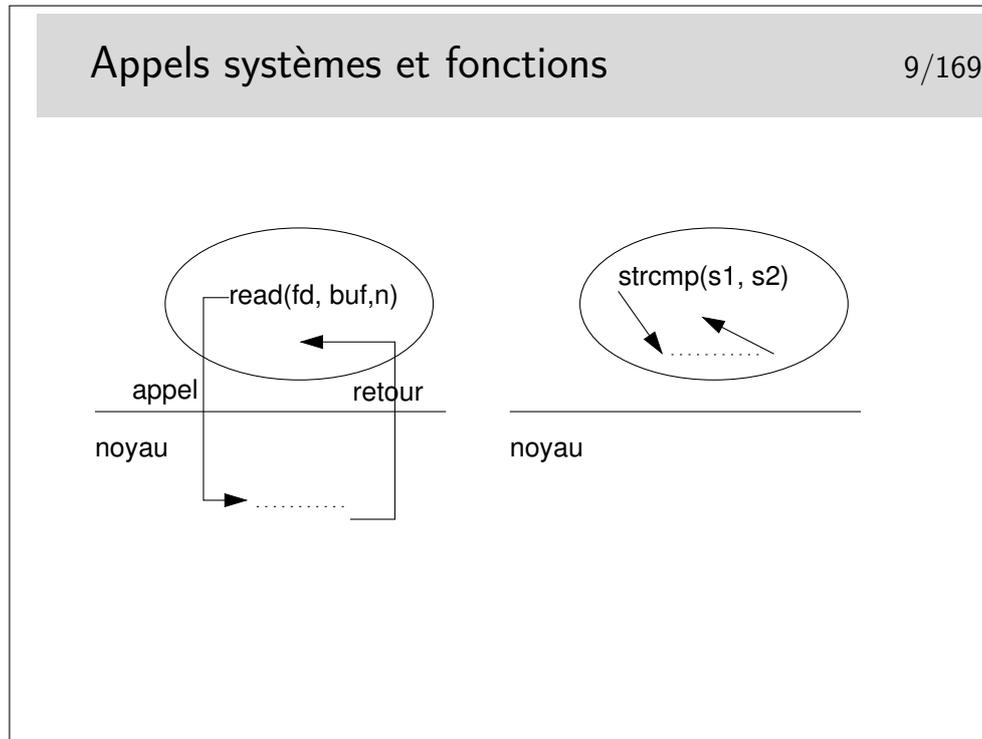
Le résultat d'un appel système peut être l'entier retourné directement. C'est le cas pour `open()` ou `socket()` par exemple, qui renvoient un entier appelé «descripteur de fichier». Ce n'est pas le cas pour `stat()` qui doit renvoyer toutes les informations possibles sur un fichier dont le nom est passé en argument. Dans ce cas, le résultat est complexe, il se présente sous la forme d'une structure qui doit être allouée dans l'espace mémoire du processus (par `malloc()`, ou par déclaration en pile). Il faut alors passer l'adresse de la structure à l'appel système afin que celui-ci puisse renvoyer un résultat correct.

Utilisation des des fonctions

8/169

- ▶ Rendent une valeur de type **divers** (entier, caractère, pointeur), voir le manuel de référence pour chacune d'entre elles
 - ▶ Lorsqu'elles rendent un pointeur, celui-ci est le **pointeur NULL** en cas d'échec
- ▶ Certaines peuvent utiliser un appel système (`fopen`, `fread`, `fwrite`, `fgets`, `fputs`, etc.)
- ▶ Les fonctions rendant un pointeur ont généralement alloués de la mémoire dans l'espace du processus et le pointeur rendu y donne accès

Les fonctions qui rendent leur résultat via un pointeur peuvent poser des problèmes de ré-entrance dans le cas d'utilisation de threads. En effet, le pointeur indique une zone allouée par la fonction, repérée en interne par une variable de type statique (si elle est nulle on alloue, sinon on réutilise). Dans le cas d'un processus qui utilise le pointeur rendu plusieurs fois de suite il n'y a pas de problème. Dans le cas de plusieurs threads du même processus, il n'en va pas de même, chaque thread s'attendant à trouver une donnée privée. Le problème est généralement indiqué dans le manuel de référence et les fonctions en cause sont doublées par des fonctions presque identiques mais «thread safe».



L'exécution d'un appel système se réalise dans le noyau. Si ce dernier ne peut pas effectuer le travail demandé, l'appel système bloque (par défaut) et le processus se trouve ainsi arrêté. Par exemple lorsqu'un processus tente de lire une socket en réseau et qu'il n'y a rien à lire, le processus se trouve bloqué. Il est généralement possible de paramétrer les «objets» bloquants (les descripteurs de fichier comme les sockets) pour que les appels systèmes soient non bloquants. Dans ce dernier cas, les appels rendent `-1` et la variable `ERRNO` est placée à la valeur `EAGAIN` (`EWOULDBLOCK` sur BSD).

Le traitement d'une fonction normale se fait dans l'espace mémoire du processus. Il peut y avoir des appels systèmes sous-jacents, par exemples avec `fopen()` (appel système `open()`), `fread()` (appel système `read()`), `fwrite()` (appel système `write()`), etc.

- ▶ Voyez la page du manuel de l'appel système `stat(2)`
 - ▶ Que fait cet appel système ?
 - ▶ Pourquoi faut-il lui passer un pointeur sur une structure `stat` en paramètre ?
 - ▶ Ce pointeur doit être alloué auparavant, pourquoi ? Sinon que se passe-t-il ?
- ▶ Voyez la page du manuel de la fonction `gethostbyname(3)`
 - ▶ Quel est le rôle de cette fonction ?
 - ▶ Comment récupère-t-on son résultat ?
 - ▶ Où est réalisée l'allocation de l'espace mémoire nécessaire pour stocker son résultat ?

Notation `stat(2)`, `gethostbyname(3)` : le chiffre entre parenthèses indique la section du manuel de référence où est définie la fonction : 2 pour les appels systèmes, 3 pour les fonctions normales.

Extrait du manuel de référence :

```
int stat(const char *file_name, struct stat *buf);
struct hostent *gethostbyname(const char *name);
```

Test du retour des fonctions et appels systèmes

11/169

- ▶ **IL FAUT TOUJOURS TESTER LA VALEUR DE RETOUR D'UN APPEL SYSTÈME**
 - ▶ Si valeur rendue égale à -1
 - ▶ il faut gérer le problème
 - ▶ une variable externe de nom `errno` est positionnée à une valeur indiquant l'erreur
- ▶ Il faut presque toujours tester la valeur de retour d'une fonction
 - ▶ Pour les fonctions rendant un pointeur, si la valeur rendue est `NULL`, il faut gérer le problème
- ▶ Envoi de messages d'erreurs
 - ▶ Fonctions `perror()` et `fprintf()`

La variable `errno` et la fonction `perror()` 12/169

- ▶ Lorsqu'un appel système échoue, le noyau positionne la variable externe `errno` à une valeur significative de l'erreur
 - ▶ `errno` est de type entier, à 0 par défaut (lorsqu'il n'y a pas eu d'erreur)
 - ▶ Le fichier `errno.h` associe des mnémoniques à chaque erreur «standard»
- ▶ La fonction `perror("texte");` affiche le texte indiqué suivi par « : » puis par le message système correspondant à l'erreur

Le manuel de référence 13/169

- ▶ Partie 2 : les appels systèmes
- ▶ Partie 3 : les fonctions
- Regarder attentivement les syntaxes, la valeur retournée, les erreurs possibles et les valeurs `errno` associées.

Exemple extrait de la section `ERRORS` de `open(2)` :

`ERRORS`

`EEXIST` pathname already exists and `O_CREAT` and `O_EXCL` were used.

`EISDIR` pathname refers to a directory and the access requested involved writing (that is, `O_WRONLY` or `O_RDWR` is set).

- ▶ Partie 4 : les pilotes de périphériques
- ▶ Partie 7 : divers (ip en particulier)

Dans l'exemple ci-dessus le mot `EEXIST` correspond à la valeur affectée à la variable `errno` lorsque l'appel système `open` échoue lorsque les conditions indiquées sont remplies (le fichier existe mais les drapeaux `O_CREAT` et `O_EXCL` étaient utilisés dans le `open`. Faire `man open` pour une description plus exhaustive).

Ayez le réflexe «RTFM» : *Read That Fine* (ou *F...*) *Manual*!

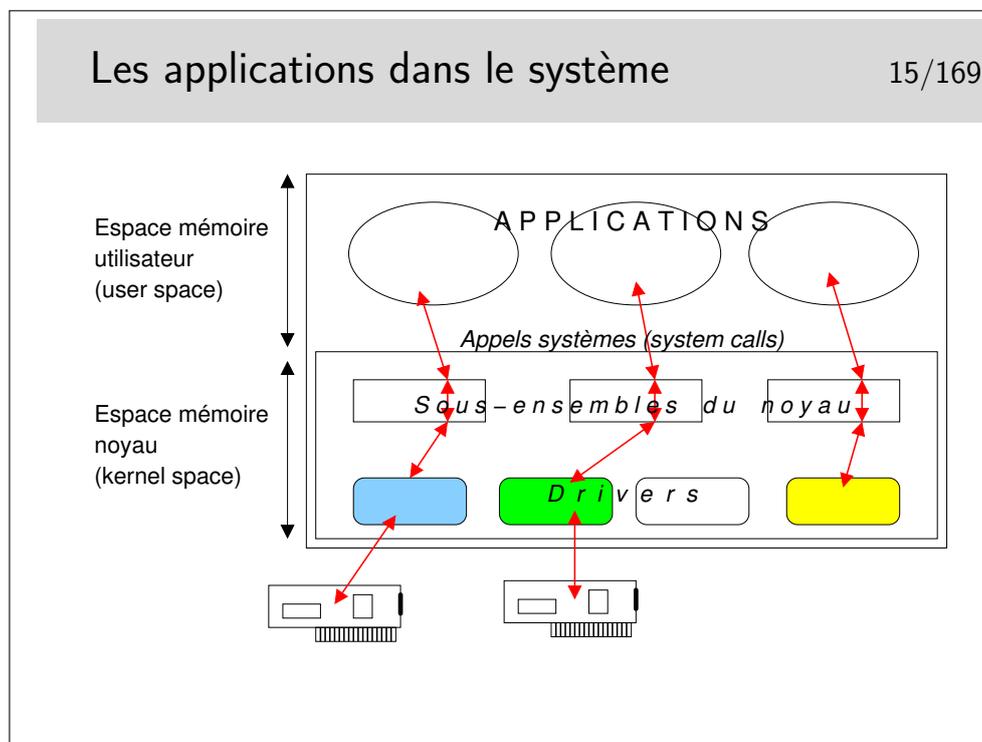
Exercice :

- Prenez la page du manuel concernant la fonction `fopen()`.

- Quel est le type de la valeur rendue ?
- Où est déclaré ce type ?
- Que se passe-t-il si on omet d'inclure la ligne `#include <stdio.h>` dans un programme qui utilise `fopen()` ?
- Prenez maintenant la page du manuel concernant la fonction `getpwuid()`.
 - Comment interprétez vous la syntaxe suivante :
`struct passwd *getpwuid(uid_t uid);`
 - Que se passe-t-il si vous n'incluez pas les lignes `#include` indiquées ?

2 Les processus

2.1 Création, environnement, signaux, terminaison

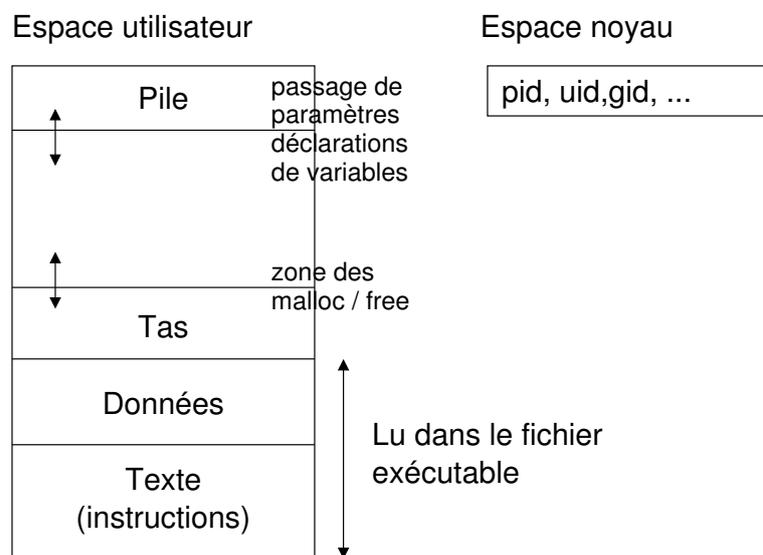


Les applications sont chargées en mémoire pour être exécutées.

Elles s'exécutent dans des entités appelées «processus», dans une partie de la mémoire non occupée par le noyau.

Les applications (processus) communiquent avec le noyau via des fonctions particulières appelées «appels systèmes» et parfois «primitives»

- ▶ Application
 - ▶ Au moins un fichier exécutable
 - ▶ Plus éventuellement des bibliothèques dynamiques
- ▶ Lancement d'une application
 - ▶ Chargement en mémoire du fichier exécutable et lancement de l'exécution de la fonction `main()`
 - ▶ Le fichier chargé en mémoire et en cours d'exécution est appelé processus
- ▶ Processus
 - ▶ Un fichier exécutable en cours d'exécution
 - ▶ Des informations complémentaires d'environnement



- ▶ *Informations complémentaires pouvant paramétrer l'exécution du programme du processus*
- ▶ Trois types importants d'informations
 - ▶ Les variables d'environnement
 - ▶ Contenues dans une structure de données de type tableau de pointeurs de caractères
 - ▶ L'identité de l'utilisateur et du groupe pour lesquels ce processus est lancé et les droits associés
 - ▶ Utilisateur/groupe réel et utilisateur/groupe effectif
 - ▶ Les fichiers standards d'entrée-sortie
 - ▶ d'entrée
 - ▶ de sortie
 - ▶ de sortie d'erreur

- ▶ Chaînes de caractères au sens C, en majuscules par coutume
 - ▶ Syntaxe (du shell) : NOM=valeur
- ▶ Accessibles par
 - ▶ La variable externe char ****environ**

```
environ -> [ * * * * ..... NULL ]
```

↙

HOSTNAME=polux

↘

TERM=vt100
 - ▶ La fonction `getenv()` pour obtenir la valeur d'une variable
 - ▶ La fonction `setenv()` pour positionner une nouvelle valeur
 - ▶ (Ou éventuellement le troisième paramètre du `main()`)

Le nom des variables d'environnement n'est pas obligatoirement en majuscules, il s'agit simplement d'une coutume qu'il en soit ainsi.

Attention, il n'y a pas d'espace de part et d'autre du signe égal (voir la syntaxe de votre shell).

Les variables d'environnement sont en général positionnées dans le Shell de l'utilisateur, via des fichiers de paramétrage tels que **.bashrc**.

- ▶ Quelques variables standard
 - ▶ HOSTNAME, PATH, HOME, LOGNAME, TERM, DISPLAY, etc.

- ▶ Deux identités d'utilisateur !
 - ▶ L'utilisateur réel : celui qui a lancé le processus, identifié par son numéro d'utilisateur dans `/etc/passwd` (ruid : *real user ID*)
 - ▶ L'utilisateur effectif :
 - ▶ Dans la majorité des cas il s'agit de l'utilisateur réel
 - ▶ Si le fichier exécuté (qui a donné naissance au processus) a les bit `set_user_ID` positionné (une lettre «s» apparaît à la place du «x» des droits du propriétaire du fichier) alors l'utilisateur effectif est le propriétaire du fichier exécuté (euid : *effective user ID*)
 - ▶ Attention : trou de sécurité potentiel, surtout si le fichier appartient à root

L'identité du groupe associé au processus 22/169

- ▶ Deux identités de groupe
 - ▶ Concept identique à ci dessus : groupe réel, groupe effectif (bit `set_group_ID`)
 - ▶ Le bit `set_group_ID` est visible par `ls -l` par un «s» qui remplace le «x» marquant l'exécutabilité pour le groupe

Les fichiers standards d'entrée-sortie 23/169

- ▶ Trois fichiers standards
 - ▶ Le fichier standard d'entrée (descripteur 0, `FILE Pointer stdin`)
 - ▶ Le fichier standard de sortie (descripteur 1, `FILE Pointer stdout`)
 - ▶ Le fichier standard de sortie d'erreur (descripteur 2, `FILE Pointer stderr`)
- ▶ Ouverts par défaut lors du lancement d'un exécutable
- ▶ Associés virtuellement au clavier pour l'entrée standard et à l'écran pour les deux autres
- ▶ Ils peuvent être redirigés vers des fichiers réels ou des tubes de communication

Voir plus loin les notions de descripteur de fichier et de `FILE Pointer`. `stdin`, `stdout` et `stderr` sont définis dans `/usr/include/stdio.h`.

Le concept de fichier standard est surtout utile pour les redirections. Voir le mécanisme plus loin.

Création d'un nouveau processus

24/169

- ▶ Un processus est toujours créé par un autre processus via l'appel système `fork()`
 - ▶ Le processus créé est appelé processus fils
 - ▶ Le processus créateur est appelé le parent ou le père
 - ▶ Non, il n'y a pas de processus «saint esprit» mais il existe des zombies
- ▶ Le processus fils est créé par le noyau dans une zone mémoire allouée spécifiquement
- ▶ Le processus fils est une copie du processus père. À un élément près c'est un clone du père
 - ▶ À noter l'existence de l'appel système `clone()` permettant de créer un processus fils capable de partager des ressources t.q. mémoire, gestionnaires de signaux et descripteurs de fichiers. Utilisé pour créer des *threads*, via les fonctions de création appropriées.

Différentiation père/fils

25/169

- ▶ Le fils est un clone du père à une mutation génétique près...
Le seul élément qui diffère est la valeur rendue par l'appel système de création `fork()`
 - ▶ 0 dans le processus fils
 - ▶ différent de 0 dans le père (égal au numéro de processus du fils créé)
 - ▶ -1 rendu dans le père si le fils n'est pas créé
- ▶ **Attention** : les deux processus sont des quasi clones, et exécutent le même code sur les mêmes données!
 - ▶ on distingue les instructions exécutées par le père de celles exécutées par le fils selon le code de retour du `fork()` via une structure de contrôle `if` ou `switch`

- ▶ Frontières étanches
 - ▶ Les deux processus ne partagent pas de mémoire commune, la communication entre les deux est impossible par des moyens simples (voir notes)
 - ▶ Le fils commence sa vie en exécutant le code situé **après** la fonction de création (fork). Il ne commence pas au début du programme bien qu'il en possède le code
- ▶ Pourquoi créer un processus fils ?
 - ▶ Pour pouvoir exécuter deux tâches simultanément (ou quasi simultanément dans le cas de machines mono processeur)

La communication entre processus père et fils peut se faire grâce à des tubes de communication (pipe), des sockets, de la mémoire partagée (Voir IPC SysV).

La communication entre deux processus de la même machine est aussi complexe à mettre en œuvre pour des processus locaux à la même machine que pour des processus sur des machines distantes.

Exemple de code de création de processus 27/169

```
int pid;
.....
pid = fork();
switch (pid) {
    case -1: /* Problème, la table de processus est pleine,
            ou il manque de la mémoire */
        /* Réagir selon le contexte */
        break;
    case 0 : /* Nous sommes dans le processus fils*/
        /* écrire ici les instructions du fils */
        break;
    default: /* Nous sommes dans le père */
        /* écrire ici les instructions du père */
}
}
```

La section de code entre le **case 0** et le **break** qui suit ne sera exécutée que dans le processus fils puisque ce n'est que dans lui que pid vaut 0. Dans le père la variable pid

est égale au numéro du processus fils ($0 < pid < 30000$).

Le même programme peut donc contenir du code qui ne sera exécuté que par l'un des deux processus. Pour le développeur, toute la difficulté sera maintenant de construire son programme en envisageant le parallélisme introduit.

Pour des raisons de clarté, il est préférable de développer le code du fils dans une fonction spécifique et d'appeler celle-ci dans le "case 0".

Numéros de processus

28/169

- ▶ Un processus a toujours un numéro compris entre 1 et 32768
 - ▶ Le principal : `init`, le processus d'initialisation du système : $pid = 1$
 - ▶ On peut lister les processus et voir leurs numéros avec la commande `ps`
- ▶ Par défaut un processus ne connaît pas son numéro
 - ▶ Il peut demander à le connaître via la fonction `getpid()`
 - ▶ Il peut connaître le numéro de son père via `getppid()`

Le tout premier processus est le noyau lui-même, de numéro 0. Sous Linux, il n'apparaît pas directement avec la commande `ps`, il faut demander à voir les processus parents pour le voir (on voit ainsi que `init` a pour père 0).

`init` est le processus qui administre la machine, il lance les services, en particulier les services de connexion d'utilisateurs, les services réseaux, etc.

Sous Linux, cette valeur limite de 32768 processus est paramétrable via le fichier spécial `/proc/sys/kernel/pid_max` (voir le `man proc`).

Contrôle sur l'identité de l'utilisateur et les droits du processus

29/169

- ▶ Les droits du processus sont ceux de l'utilisateur effectif et du groupe effectif
- ▶ L'utilisateur effectif est différent de l'utilisateur réel si le bit `set_user_ID` est positionné dans les droits du fichier exécuté
- ▶ Les groupe effectif est différent du groupe réel si le bit `set_group_ID` est positionné dans les droits du fichier exécuté
- ▶ On peut accéder à ces paramètres avec `getuid()`, `geteuid()`, `getgid()` et `getgeid()`

Contrôle sur l'identité de l'utilisateur et les droits du processus

30/169

- ▶ Pour un processus dont l'utilisateur effectif (`euid`) est différent de l'utilisateur réel (`ruid`) il est possible de modifier l'`euid` pour le ramener à la valeur du `ruid` avec `setuid()`
 - ▶ Le processus reprend alors les droits de l'utilisateur réel
 - ▶ La valeur du `euid` est sauvegardée dans une variable interne `saved_user_ID`, pour permettre un retour aux droits de l'`euid` de départ
- ▶ Démarche identique pour le groupe effectif avec `setgid()` ou `setegid()`

Imaginons un fichier exécutable appartenant à root, ayant le bit `set_user_ID` positionné dans ses droits, exécuté par l'utilisateur dupont (uid 501 par exemple) et appelant `setuid()` de la manière qui suit, les droits changerons comme indiqué :

<code>/* début du programme */</code>	<i>ruid=501 (dupont), euid = 0 (root) : droits de root</i>
<code>...</code>	
<code>...</code>	
<code>euid = geteuid();</code>	<i>pour mémoriser euid</i>
<code>setuid(getuid());</code>	<i>euid = ruid = 501 (dupont) : droits de dupont</i>
<code>...</code>	<i>(saved_user_ID = 0)</i>
<code>...</code>	
<code>setuid(euid);</code>	<i>ruid=501 (dupont), euid = 0 (root) : droits de root</i>

Certains programmes tels que les shells (`/bin/sh`) repassent immédiatement sous l'identité de l'utilisateur réel, pour éviter les bêtises au cas où le programme ou le script aurait été installé avec le bit SUID...

Contrôle sur les processus : les signaux

31/169

- ▶ Un signal est une sorte d'interruption logicielle envoyée à un processus par le noyau après qu'un événement particulier soit intervenu
- ▶ L'événement peut être :
 - ▶ Une faute logicielle (division par 0, manipulation d'une adresse mémoire interdite, erreur d'alignement de donnée)
 - ▶ Terminaison d'un processus fils : par défaut (mais paramétrable) le père est prévenu
 - ▶ Intervention de l'utilisateur via le shell ou l'interface graphique pour tuer le processus ou le stopper ou autre (modification de la taille d'une fenêtre par exemple)
- ▶ Dans la plupart des cas le signal est fatal au processus

Signal	Numéro	Fonction
HUP	1	Signal envoyé au processus en premier plan associé à un terminal lorsque celui-ci est fermé
INT	2	Envoyé depuis le clavier avec la combinaison de touches <CTRL-C> (par défaut)
QUIT	3	Envoyé depuis le clavier avec la combinaison de touches <CTRL- > (par défaut)
KILL	9	Ne peut être intercepté, envoyé depuis le clavier via la commande <code>kill</code> (<code>kill -9</code> ou <code>kill -KILL</code>)
TERM	15	Envoyé via le clavier par la commande <code>kill</code> simple
SEGV		Erreur de segmentation, accès à une zone mémoire interdite
CLD		Terminaison d'un fils
WINCH		Modification de la taille de la fenêtre associée à l'application
STOP		Arrêt du processus sans le terminer. Envoyé via la combinaison de touches <CTRL-Z>
URG		Une données urgente a été reçue via le protocole TCP et est en attente de lecture (voir le cours sur la programmation réseau)
IO		Des données réseau sont arrivées et sont en attente de lecture. (voir le cours sur la programmation réseau)
USR1		Nom de signal utilisable par le développeur, à son gré
USR2		Nom de signal utilisable par le développeur, à son gré

Certains signaux destructeurs font qu'un fichier de nom **core** soit produit dans le répertoire courant. Ce sont essentiellement les signaux **SIGQUIT**, **SIGSEGV** et **SIGBUS**.

Ce fichier **core** est une copie de l'image mémoire du processus au moment où l'erreur s'est produite, elle permet donc un débogage ultérieur avec un outil adapté (**gdb** par exemple). Ce fichier n'est utile que si l'on possède les sources de l'exécutable qui a «fauté», dans le cas contraire il ne sert à rien et on peut l'effacer. Il faut aussi que le source ait été compilé avec l'option **-g** pour que le débogage soit possible.

Le fichier **core** ne sert donc pas très souvent et il est fréquent que le paramétrage de l'environnement de l'utilisateur interdise sa production. Pour cela il suffit que la fonction interne du bash **ulimit** ait été invoquée de la manière suivante : **ulimit -c 0**.

Pour ré-autoriser la production de fichier **core** sur erreur on peut faire : **ulimit -c unlimited** (voir le manuel de bash).

Gestion des signaux : la fonction `signal()` 33/169

- ▶ Un signal arrive de manière inattendue. Il faut préparer le processus si on désire qu'il gère l'arrivée du signal.
- ▶ La manière la plus simple est d'utiliser `signal()`
`signal(SIGXYZ, fct);`
 - ▶ `SIGXYZ` est le nom du signal à gérer
 - ▶ `fct` est le nom de la fonction de gestion du signal
 - ▶ `fct` peut prendre les valeurs suivantes :
 - ▶ `SIG_IGN` si on veut ignorer le signal
 - ▶ `SIG_DFL` si on veut restituer le comportement par défaut associé au signal
 - ▶ le nom (sans les parenthèses) d'une fonction de gestion du signal, définie quelque part dans le programme par le développeur de l'application

Exemple d'utilisation de `signal()`

34/169

```
1. void sighdl(int n) {
2.     printf("signal reçu %d\n", n)
3. }
4.
5. int main() {
6.     ...
7.     signal(SIGINT, sighdl);
8.     ...
9. }
```

Lignes 1 à 3 : corps d'une fonction de gestion de signal.

Lignes 5 à 9 : programme principal, les pointillés symbolisent des instructions diverses propres au programme.

Ligne 9 : appel à `signal()`. Le premier argument indique le signal `SIGINT` (envoyé au clavier par <Control-C>). Le second argument est l'adresse de la fonction de gestion. La fonction `signal()` mémorise, pour le processus, que celui-ci, s'il reçoit le signal `SIGINT`, devra se dérouter sur la fonction indiquée.

Remarques :

- Les fonctions de traitement de signal doivent être courtes. Ce n'est pas une bonne pratique que de placer le traitement principal de l'application dans des fonctions de gestion de signal.
- La mise en place de gestionnaires de signaux fait partie de l'initialisation de l'application, on le fait donc plutôt au début.
- Attention à ne pas confondre les choses : c'est bien le système d'exploitation qui appellera le gestionnaire de signal le jour où le signal arrivera, ce n'est pas la fonction `signal()` qui l'appelle (elle ne fait qu'indiquer sa présence au système d'exploitation).

Gestion des signaux : la fonction `signal()` 35/169

► Syntaxe :

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

- La fonction `signal()` renvoie donc un pointeur de fonction
- Ce dernier pointe sur la fonction qui était précédemment associée au signal. Ainsi, lors d'une première utilisation de `signal()` pour un signal donné, le pointeur renvoyé sera `SIG_DFL`. Il est évidemment possible de garder ce pointeur en mémoire dans une variable de type `sig_handler_t`

Le type rendu par `signal()` n'est pas toujours défini comme ci-dessus (`sig_handler_t`). Il peut varier selon les types de libc (`SignalHandler`, `sig_t`). C'est un inconvénient mineur pour la portabilité des sources, en aucun cas pour le fonctionnement, l'essentiel étant que `signal()` rende un pointeur sur la fonction de gestion précédemment associée au signal.

Autre problème, de comportement cette fois : lorsque la fonction est appelée quand le signal survient, le comportement associé au signal peut être réinitialisé à son défaut juste avant d'appeler la fonction. La fonction est quand même appelée mais elle ne le sera plus. C'est le comportement traditionnel UNIX système V. En Unix BSD par contre c'est l'inverse, le déroutement vers la fonction reste associé au signal même après la première fois. Une solution pour ne pas se poser de question est de rappeler `signal()` dans la fonction (ainsi on «réarme» à chaque fois).

Autre solution, recommandée, est d'utiliser `sigaction()`, plus complexe, plus riche, mais plus contrôlable.

▶ Syntaxe :

```
#include <signal.h>

int sigaction (
    int sig, /* le nom du signal */
    const struct sigaction *act, /* l'action nouvelle */
    struct sigaction *oldact /* l'ancienne action */
);

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

Notes sur la structure sigaction :

La fonction de traitement associée au signal est indiquée dans le champ `sa_handler` ou dans le champ `sa_sigaction` (si `SA_SIGINFO` est indiqué dans `sa_flags`). Sur certaines architectures on emploie une union, il ne faut donc pas utiliser ou remplir simultanément `sa_handler` et `sa_sigaction`.

▶ Syntaxe :

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

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

▶ `sig` : nom (dans `signal.h`) ou numéro du signal, préférer le nom▶ `pid` : obéit aux règles suivantes :

- > 0 : c'est le numéro du processus destinataire
- = -1 : le signal est envoyé à tous les processus sauf le processus numéro 1 et le processus en cours lui même
- < -1 : le signal est envoyé à tous les processus du groupe de processus de numéro pid indiqué
- = 0 : le signal est envoyé à tous les processus du groupe de processus dont fait partie le processus courant

La notion de «groupe de processus» est vue plus loin.

Pour s'envoyer un signal à soi même, on peut utiliser l'appel système `raise(int sig)`,

équivalent à `kill(getpid(), sig)`.

La terminaison d'un processus 1/2

38/169

- ▶ Terminaison normale
 - ▶ Par appel à la fonction `exit()`
 - ▶ Explicite
 - ▶ Implicite après la dernière instruction de la fonction `main()`
 - ▶ Le paramètre de `exit()` est passé au processus père du processus qui se termine, le père peut connaître ce paramètre via l'appel système `wait()`
 - ▶ Par appel à `return` en dernière instruction de `main()`. C'est équivalent à `exit()`
 - ▶ Le père reçoit le signal `SIGCLD`

La terminaison d'un processus 2/2

39/169

- ▶ Terminaison anormale
 - ▶ Par signal généré par le noyau sur faute du processus ou généré par un autre processus ou via le clavier
 - ▶ Le père reçoit le signal `SIGCLD`
 - ▶ Le processus père peut connaître le numéro du signal via `wait()` appelé typiquement dans une fonction handler du signal `SIGCLD`

- ▶ **Syntaxe : `void exit(int status)`**
 - ▶ Le paramètre `status` est un nombre compris entre 0 et 255. Une programmation conforme aux standards indique que :
 - ▶ `= 0` indique une terminaison normale (voir note)
 - ▶ `< 0` sert à indiquer que le programme n'a pas pu faire son travail pour une raison quelconque (voir note)
 - ▶ `exit()` permet d'appeler des fonctions de nettoyage final préalablement indiquées au processus via les fonctions `atexit()` ou `on_exit()`
 - ▶ Les fichiers temporaires créés avec `tmpfile()` sont effacés
 - ▶ Les fichiers encore ouverts sont fermés

Les shells stockent le paramètre du `exit` dans une variable interne pouvant être testée. Sous `bash` elle s'appelle «`?`».

Essayez les commandes suivantes :

```
bash$ grep root /etc/passwd
...
bash$ echo $?
...
bash$ grep toto /etc/passwd
bash$ echo $?
...
bash$ grep toto /etc/pwd
...
bash$ echo $?
...
bash$ true
bash$ echo $?
...
bash$ false
bash$ echo $?
```

Comment expliquez vous les réponses à la commande `echo $?`

Sous C-Shell la variable est `status`.

Le C standard recommande d'utiliser les mots clé `EXIT_SUCCESS` et `EXIT_FAILURE` pour des raisons de portabilité entre systèmes non Unix. Mais dans ce cas nous avons seulement une valeur pour indiquer un code d'erreur alors qu'avec les valeurs numériques nous en avons 255.

- ▶ Doit gérer ou ignorer explicitement le signal SIGCLD (ou SIGCHLD)
 - ▶ Ignorer : `signal(SIGCLD, SIG_IGN);`
 - ▶ Gérer : avec l'appel système `wait()` ou `waitpid()` placé dans une fonction de gestion du signal SIGCLD si on ne désire pas que le processus bloque
- ▶ L'appel système `wait()` :

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

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

`wait()` renvoie le numéro du processus fils terminé.

Si son argument (`*status`) n'est pas nul, on retrouvera l'information sur la terminaison du fils à l'adresse pointée par `status`. Cette information nous donnera la valeur du paramètre du `exit` du fils s'il s'est terminé par `exit()`. S'il s'est terminé sur un signal nous pourrions avoir le numéro de ce signal. Voir transparent suivant.

L'appel système `waitpid()` permet d'attendre la terminaison d'un processus fils particulier indiqué par son numéro de processus.

Ces appels systèmes sont bloquants. Si on ne désire pas que le processus qui les appelle bloque, il faut alors les placer dans une fonction de gestion du signal SIGCLD.

Autres fonctions issues de BSD : `wait3()` et `wait4()`. Voir le manuel de référence.

- ▶ **Macros spécifiques**
 - ▶ `WIFEXITED(status)` rend VRAI si le fils s'est terminé sur un exit. Alors :
 - ▶ `WEXITSTATUS(status)` : rend la valeur du paramètre du exit
 - ▶ `WIFSIGNALED(status)` : rend VRAI si le fils s'est terminé sur un signal. Alors :
 - ▶ `WTERMSIG(status)` : rend la valeur du signal de terminaison du fils
 - ▶ `WCOREDUMP(status)` : rend VRAI si la terminaison par signal a produit un «core dump»
 - ▶ `WIFSTOPPED(status)` : rend VRAI si le fils a été stoppé. Alors :
 - ▶ `WSTOPSIG(status)` : rend le numéro du signal de STOP

On n'utilise `WEXITSTATUS` que si `WIFEXITED` rend VRAI (différent de 0).

On n'utilise `WTERMSIG` que si `WIFSIGNALED` rend VRAI.

On n'utilise `WSTOPSIG` que si `WIFSTOPPED` rend VRAI.

Certains signaux font que le processus qui les reçoit se termine en produisant un fichier image de lui-même appelé core. Cette image peut ensuite être utilisée pour déboguer le programme (signaux tels que `SIGSEGV` et `SIGBUS`).

- ▶ Un processus zombie est un processus fils pour lequel son père n'a pas acquitté la terminaison
 - ▶ Le père n'a pas fait un `wait()` après la terminaison du fils
 - ▶ Ou le père n'a pas demandé d'ignorer le signal `SIGCLD`
- ▶ Le processus zombie est vidé de sa substance mais reste dans la liste des processus de la machine et peut être listé par `ps`
 - ▶ On ne peut plus le supprimer, il faut supprimer le père pour que le zombie disparaisse
 - ▶ Il est généralement dû à une erreur de programmation

- ▶ Groupes de processus
 - ▶ Un père génère des fils, par défaut ces fils font partie du groupe de processus du père
 - ▶ Un processus peut créer son groupe ou demander à en changer avec `setpgid()` ou `setpgrp()`
 - ▶ Un processus qui crée son groupe devient *Process Group Leader*
 - ▶ Un processus peut connaître son groupe avec `getpgid()` ou `getpgrp()`

Syntaxe de `setpgid()` :

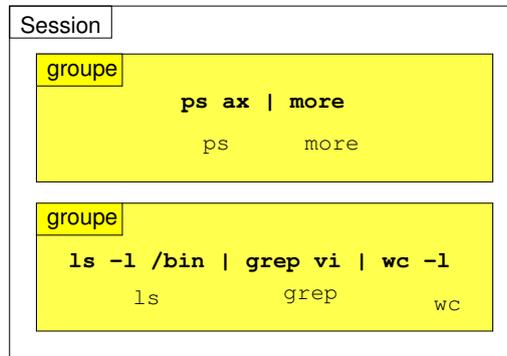
```
int setpgid(pid_t pid, pid_t pgid);
```

Si `pid` et `pgid` sont égaux à 0, alors le processus devient *Process Group Leader*

Un processus peut changer de groupe pour lui même ou pour l'un de ses fils.

Un processus *Session Leader* ne peut pas changer de groupe (voir ci-après).

- ▶ Lorsqu'un utilisateur se connecte, le premier processus qui lui est alloué se libère de la «tutelle» de son père en ouvrant une session de processus. Il devient *Process Session Leader*
- ▶ Une session contiendra plusieurs groupes de processus
- ▶ Un terminal, dit «terminal de contrôle» sera associé à la session, le terminal sera libéré lorsque la session se terminera (à la déconnexion de l'utilisateur)
- ▶ Un processus peut devenir *Session Group Leader* par appel à `setsid()`. Il perd alors le terminal de contrôle, il en retrouve un dès qu'il ouvre un terminal

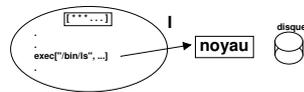


- ▶ Le processus lancé par l'appel du fichier exécutable crée un fils
 - ▶ Il se termine tout de suite par `exit()`
 - ▶ Son fils appelle `setsid()` et devient *Session Leader* et perd son terminal de contrôle, le processus fils est le démon
 - ▶ Le processus `init(1)` «adopte» le fils orphelin
 - ▶ Remarque : à ne pas lancer via `inittab` en mode `respawn`

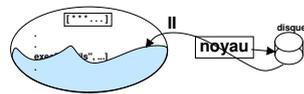
Note : voir également `daemon(3)` qui fait tout ça tout seul.

Exécution d'un fichier par un processus L'appel système `execve()`

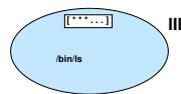
48/169



Étape I : le processus appelle `exec` en indiquant un fichier exécutable en paramètre



Étape II : le noyau va chercher le fichier sur le disque et le recopie à l'endroit où réside le processus. Le code original de celui-ci est écrasé et remplacé par le code du fichier



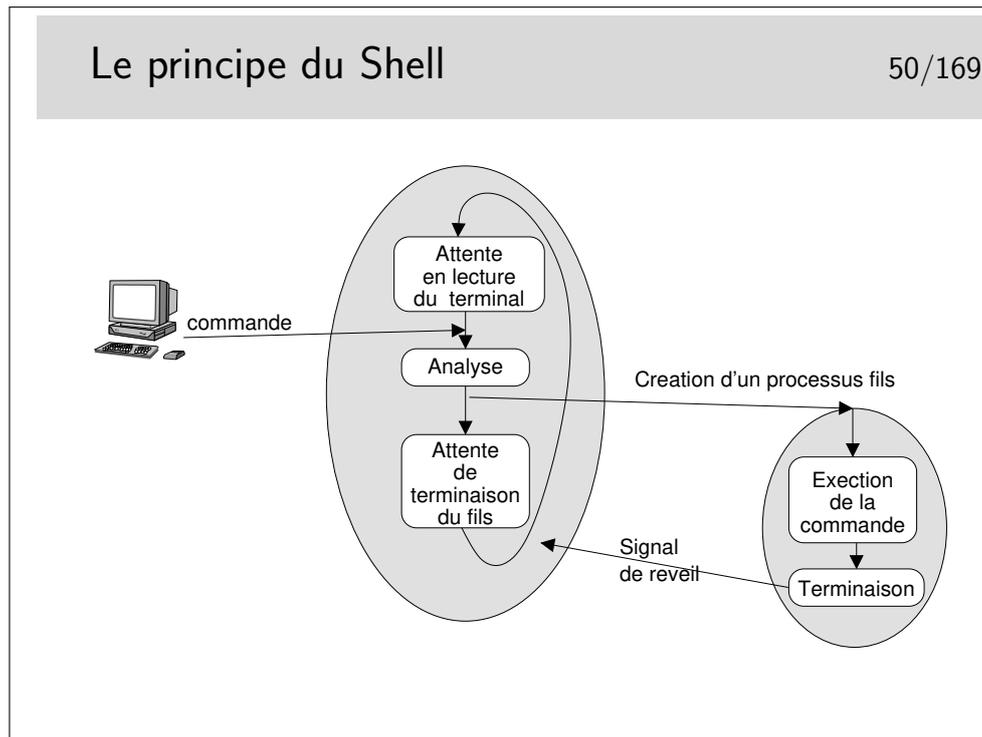
Étape III : le processus commence l'exécution de son nouveau code. Il ne peut y avoir retour à l'ancien code

On remarquera qu'un élément a résisté à l'envahisseur... Il s'agit du tableau des variables d'environnement qui n'a pas été altéré. Il est néanmoins possible de remplacer ce tableau par un nouveau.

La famille des fonctions `exec`

49/169

- ▶ Un ensemble de fonctions présentent une utilisation parfois plus aisées que `execve()`
 - ▶ `execl()` : il faut fournir les paramètres de `main()` explicitement un par un (liste)
 - ▶ `execlp()` : comme `execl()` mais prend en compte la variable `PATH`
 - ▶ `execv()` : les paramètres de `main()` sont fournis dans un tableau (vecteur)
 - ▶ `execvp()` : comme `execv()` mais prend en compte `PATH`



La phase «analyse» consiste pour le Shell à remplacer les métacaractères (*, ?, \, accents, etc.) pouvant être présent dans la ligne de commande. Il vérifie ensuite si la commande correspond à un alias, une fonction interne (une fonction définie par l'utilisateur dans le Shell lui même), ou une commande interne (c'est différent d'une fonction).

Dans le cas de la fonction ou de la commande interne, le Shell exécute directement ce qu'on lui demande sans créer de processus fils.

Sinon (la commande correspond à un fichier exécutable), un processus fils est créé pour exécuter la commande.

Le Shell reçoit le signal `SIGCLD` à la terminaison du processus fils.

Il ne reste pas en attente (`wait()`) si la ligne de commande est terminée par «&» (commande en background).

3 Les entrées sorties

3.1 Descripteurs et pointeurs de fichiers, primitives et fonctions

Communication d'un processus avec l'environnement

52/169

- ▶ Un processus communique généralement par le biais de fichiers
 - ▶ Fichiers ordinaires
 - ▶ Fichiers spéciaux : qui indiquent des périphériques
 - ▶ Pseudos fichiers : tubes de communication, sockets, pour communiquer entre processus
 - ▶ Ces fichiers sont manipulés dans le programme via :
 - ▶ un **descripteur** : petit entier positif ou nul, ou bien
 - ▶ un **pointeur** de type **FILE** (type opaque défini dans `<stdio.h>`)
- Descripteurs et pointeurs de FILE sont obtenus par des fonctions **d'ouverture** de fichier ou via des appels systèmes spécifiques lorsqu'il s'agit de tubes de communication et de sockets.

Il faut faire le choix de travailler avec les descripteurs ou avec les **FILE pointer**. Il n'y a pas de recommandation particulière, parfois il est plus aisé de travailler avec les descripteurs, parfois non.

Le type FILE, défini dans `<stdio.h>` s'utilise facilement, il cache une structure dont il est parfaitement inutile de connaître le contenu.

Les tubes de communication servent à la communication entre processus sur une même machine. Il en existe deux types : les tubes simples qui permettent la communication entre processus filliés (père-fils, fils-fils, ...) et les *tubes nommés*, plus généraux.

Les sockets cachent des mécanismes de communication en réseau mais permettent aussi la communication locale.

Pour ce qui concerne uniquement la communication entre processus sur la même machine, il existe une autre famille d'appels systèmes que l'on nomme les IPC Système V, IPC pour *Inter Process Communication* et Système V car ils sont issus de la version V du système Unix. Voir plus loin.

- ▶ Obtenus par les appels systèmes
 - ▶ `open()` : ouvrir un fichier ordinaire ou spécial
 - ▶ `pipe()` : créer un tube de communication
 - ▶ `socket()` : créer une socket de communication
 - ▶ `dup()` et `dup2()` : dupliquer un descripteurs existant
 - ▶ `fileno()` : obtenir un descripteur à partir d'un FILE pointer
- ▶ On obtient toujours le plus petit disponible
- ▶ S'utilisent uniquement avec des appels systèmes
 - ▶ `read()` pour les lectures
 - ▶ `write()` pour les écritures
 - ▶ Et d'autres...

Le fait d'obtenir le descripteur le plus petit disponible est fondamental. C'est sur lui que repose le mécanisme des redirections des fichiers standard d'entrée-sorties.

- ▶ Obtenus via les fonctions
 - ▶ `fopen()` : ouvrir un fichier ordinaire ou spécial
 - ▶ `popen()` : ouvrir un tube de communication et lancer une commande
 - ▶ `fdopen()` : obtenir un FILE pointer à partir d'un descripteur

Le «FILE pointer» est en fait une structure définie dans `<stdio.h>`. Un `typedef` en fait le type `FILE`.

Le type `FILE` est un type opaque, on sait qu'il existe, on manipule des pointeurs de ce type mais on n'a pas du tout besoin d'en examiner le contenu.

Lisez le man de `stdio` et de `stdout`.

- ▶ Le fichier standard d'entrée
 - ▶ Descripteur 0 (ou macro `STDIN_FILENO` de `<unistd.h>`)
 - ▶ FILE pointer `stdin` (défini dans `<stdio.h>`)
- ▶ Le fichier standard de sortie
 - ▶ Descripteur 1 (ou macro `STDOUT_FILENO`)
 - ▶ FILE pointer `stdout`
- ▶ Le fichier standard de sortie d'erreur
 - ▶ Descripteur 2 (ou macro `STDERR_FILENO`)
 - ▶ FILE pointer `stderr`
- ▶ Hérités du processus père et a priori toujours ouverts

- ▶ Les classiques :
 - ▶ `read()`, `write()` : lire des blocks d'octets
 - ▶ `getchar()`, `putchar()` : lire et écrire un caractère
 - ▶ `gets()`, `puts()` : lire et écrire une ligne
 - ▶ `printf()`, `scanf()` : écrire et lire du texte formaté
- ▶ En utilisant spécifiquement les FILE Pointers `stdin/stdout/stderr`
 - ▶ `fgets()`, `fputs()`
 - ▶ `fprintf()`, `fscanf()`
 - ▶ `fread()`, `fwrite()`

Notons également la fonction `getline()`, spécifique aux libc GNU, mais très souple d'utilisation pour lire du texte de taille non déterminée sur un FILE pointer.

Il y a aussi le cas de l'appel système `mmap()`, très prisée des développeurs soucieux d'optimiser leur code : cet appel système réalise une *projection en mémoire* du contenu d'un fichier. La zone de mémoire en question étant gérée par le noyau (et oui : c'est un des rares cas où du code utilisateur va accéder à une zone de mémoire noyau), on évite parfois de recopier les données entre l'espace noyau et l'espace utilisateur (cas typique du

read().

Exemple d'utilisation des descripteurs

57/169

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

#define BUFSIZE 1024

int fd, r;
char buf[BUFSIZE];
...
fd = open("mon_fichier", O_RDONLY);
if (fd == -1) {
    perror("Erreur open");
    exit(1);
}
...
r = read(fd, buf, BUFSIZE);
if (r==0) {
    /* Fin de fichier atteinte */
    ...
}
```

Exemple d'utilisation des FILE pointers

58/169

```
#include <stdio.h>

#define BUFSIZE 1024

FILE *fp; /* voir notes */
int r;
char buf[BUFSIZE]
...
fp = fopen("mon_fichier", "r");
if (fp == NULL) {
    perror("Erreur fopen");
    exit(1);
}
...
r = fgets(buf, BUFSIZE, fp);
...
}
```

Remarquer la spécification de la variable **fp** : c'est un pointeur sur un objet de type **FILE**.

On l'utilise avec les fonctions qui rendent des pointeurs de ce type, exemple ici **fopen()**.

Vérifiez en consultant le manuel de référence pour **fopen()**.

Limite du nombre de fichiers ouverts dans un processus

59/169

- ▶ Un processus ne peut pas ouvrir plus d'un «certain nombre» de fichiers
- ▶ La limite est variable d'un système à un autre
 - ▶ Il existe deux limites
 - ▶ souple (*soft*)
 - ▶ stricte (*hard*)
 - ▶ La fonction `getrlimit()` permet de connaître leur valeur
 - ▶ La fonction `setrlimit()` permet de repousser la limite *soft* vers la valeur *hard* indépassable
 - ▶ Pour ne pas se laisser surprendre par un nombre trop grand de fichiers ouverts on ferme les descripteurs ou **FILE pointers** dès qu'on en n'a plus besoin

Un descripteur ne désigne pas systématiquement un fichier réel, il peut désigner un tube de communication ou une socket. Un fichier ouvert par `fopen()` est identifié par un **FILE pointer** qui cache en réalité un descripteur.

Notez également que `getrlimit()/setrlimit()` permettent de manipuler la limitation d'utilisation de différents types de *ressources* (mémoire, cpu, nombre de processus fils, etc.). Consultez le man.

Les redirections

60/169

- ▶ Grace à `dup()` ou `dup2()`
 - ▶ Exemple de redirection de sortie standard :

```
int fd;
...
fd = open("le_fichier", O_WRONLY | O_CREAT, 0666);
if (fd < 0) {...}

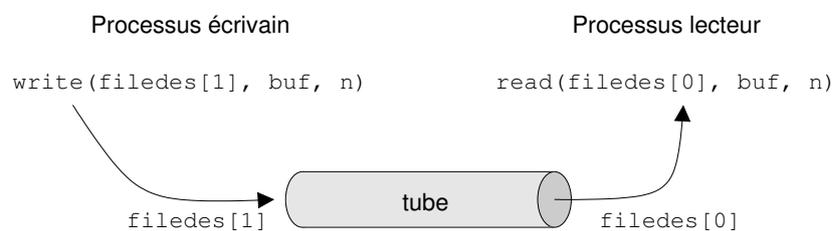
close(1);
dup(fd); /* Voilà la clé */
close(fd); /* et le tour est joué */
...
printf("xyz"); /* dans le fichier et pas sur l'écran */
```

- ▶ Un tube est ouvert par un processus père, il est hérité par ses fils qui s'en servent pour communiquer entre eux ou avec le père
- ▶ Appel système `pipe()` :

```
#include <unistd.h>

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

 - ▶ rend deux descripteurs dans le tableau `filedes[]`
 - ▶ `filedes[0]` pour la lecture
 - ▶ `filedes[1]` pour l'écriture



Une fonction intéressante : **`popen()`**

La fonction **`popen()`** permet de créer un tube de communication puis un processus fils dans lequel on exécute une commande Unix passée en premier paramètre. Le second paramètre de **`popen()`** permet d'indiquer si on veut lire ou écrire le tube.

`popen()` rend un pointeur de type **`FILE`**.

La fonction **`pclose()`** permet de fermer le tube obtenu via **`popen()`**.

Notez que l'on peut aussi faire des communications entre deux processus qui n'ont

rien à voir entre eux (pas relations de filiation ou autre), grâce à des *pipes nommés* (ou *FIFO*, voir `man 7 pipe` et `man 7 fifo`). Pour cela, il faut créer un fichier spécial de type pipe (voir la commande ou l'appel système `mkfifo`, qui est en fait une spécialisation de `mknod`). Ensuite, deux processus (seulement) doivent ouvrir ce fichier, l'un en écriture, l'autre en lecture, et communiquent comme via un pipe ordinaire.

Outils de base pour travailler avec les fichiers

63/169

- ▶ Ouvrir : `open(2)`, `fopen(3)`
- ▶ Fermer : `close(2)`, `fclose(3)`
- ▶ Lire et écrire : `read(2)`, `write(2)`, `fread(3)`, `fwrite(3)`, `fgets(3)`, `fputs(3)`, etc.
- ▶ Modifier des indicateurs associés au fichiers (flags) : `fcntl(2)`
- ▶ Se déplacer dans un fichier : `lseek(2)`
- ▶ Obtenir des informations : `stat(2)`, `lstat(2)`, `fstat(2)`
- ▶ Déterminer les droits d'accès : `access(2)`

Note : `fcntl()` permet également de poser des verrous sur des fichiers... Mais le sujet des verrous de fichiers sous Unix est éminemment complexe.

- Les verrous sont *collaboratifs* : un processus pose un "marquage" associé au fichier à l'attention des autres processus, qui, s'ils sont collaboratifs, en tiennent compte avant de réaliser des opérations de lecture écriture.
- La sémantique est assez implicite : il y a des verrous dits "exclusifs" plutôt destinés à gérer des accès en écriture, et des verrous "partagés" plutôt pour les accès en lecture, mais finalement rien ne l'impose.
- Il y a plusieurs API, implémentées différemment suivant les OS :

L'API `fcntl()` (certainement la plus souple et la plus utilisée), avec parfois (comme sous Linux) l'API `lockf()` (POSIX) implémenté par dessus. Le marquage des fichiers se fait dans une table de verrous dans la zone mémoire du noyau et très liée au fonctionnement des processus.

L'API `flock()` (BSD) utilise un marquage placé au niveau de la table des inodes, et donc liée aux données des fichiers. Cette différence implique des comportements différents lors des `dup()` `fork()` `execve()` ... Lire la documentation avec la plus grande attention.

L'API `flockfile()` ne concerne en fait que le verrouillage des flux FILE stdio à l'intérieur d'un processus, entre plusieurs threads.

- Ces verrous ont un comportement très erratique lorsqu'on les utilise sur des fichiers en réseau (c.à.d. faire en sorte qu'un processus sur une machine bloque un processus sur une autre machine via un verrou sur un fichier partagé). L'API `fcntl()` est réputée bien se comporter avec des serveurs NFS configurés judicieusement. Parfois cela fonctionne sur des fichiers CIFS ou Andrew FS, mais pas toujours...

Outils de base pour travailler avec les fichiers

64/169

- ▶ Modifier les droits d'accès : `chmod(2)`, `fchmod(2)`
- ▶ Modifier le propriétaire et groupe : `chown(2)`, `fchown(2)`
- ▶ Créer un nouveau nom : `link(2)`
- ▶ Supprimer un nom sur un fichier : `unlink(2)`
- ▶ Renommer : `rename(2)`

Note : voir également la fonction `remove(3)`, qui fait soit un `unlink(2)` si c'est un fichier, soit un `rmdir(2)` si c'est un répertoire.

Travailler avec les fichiers spéciaux

65/169

- ▶ Rappel : les fichiers spéciaux identifient des pilotes de périphériques et les périphériques eux-même
- ▶ L'appel système `ioctl()`
 - ▶ Le «couteau suisse», avec lui on fait tout ce qui n'a pas été prévu de manière standard
`ioctl(int fd, int CMD, [arg])`
 - ▶ `fd` est un descripteur obtenu lors de l'ouverture du fichier spécial
 - ▶ `CMD` identifie une commande dépendant du pilote du périphérique, il faut avoir la documentation sur le pilote
 - ▶ Il peut y avoir un argument à la commande

Travailler avec les répertoires

66/169

- ▶ Ouverture/fermeture
 - ▶ `opendir()`
 - ▶ `closedir()`
- ▶ Lecture
 - ▶ `readdir()`
 - ▶ `scandir()`
- ▶ Déplacements
 - ▶ `seekdir()`
 - ▶ `rewinddir()`
- ▶ Création, suppression, changement
 - ▶ `mkdir()` `rmdir()` `chdir()`

Rappelez-vous qu'un répertoire, même vide, contient toujours au moins deux entrées (les deux premières) qui sont «`.`» et «`..`».

4 Structure d'un logiciel Unix/Linux

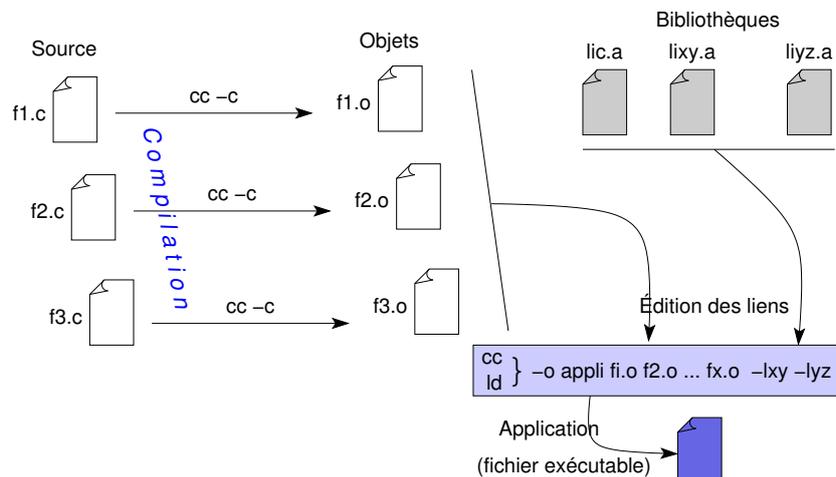
4.1 Compilation, édition de liens, bibliothèques

Logiciels - Caractéristiques

68/169

- ▶ programme ou collection de programmes et de bibliothèques
 - ▶ programmes compilés : sources en C, C++ ou autre
 - ▶ programmes interprétés : shells, scripts perl, python, Tcl, etc.
 - ▶ bibliothèques : collection de fonctions utilisés par les programmes ou les interpréteurs
 - ▶ bibliothèques système : la «libc», libX11, GTK, Qt, etc.
 - ▶ bibliothèques d'usage restreint au logiciel : libmysqlclient (par exemple)

Logiciels - interaction entre les composants 69/169



```
$ cat essai.c
#include <math.h>
main(int argc, char **argv)
{
    int x = 4;
    printf("x = %d log(x) = %f\n", x, log(x));
}
```

```
cc -S essai.c
```

```
$ less essai.s
.file "essai.c"
.version "01.01"
gcc2_compiled.: .section .rodata
.LC0:
.string "x = %d log(x) = %f\n"
.text
.align 4
.globl main
.type main,function
main:
    pushl %ebp
    movl %esp, %ebp
    subl $8, %esp
    movl $4, -4(%ebp)
    subl $8, %esp
    fldl -4(%ebp)
    leal -8(%esp), %esp
    fstpl (%esp)
    call log
    addl $16, %esp
    leal -8(%esp), %esp
    fstpl (%esp)
    pushl -4(%ebp)
    pushl $.LC0
    call printf
    addl $16, %esp
    ...
```

Cet exemple montre le résultat de la compilation proprement dite. Résultat intermédiaire car stoppé avant la production du binaire. On voit comment le code C standard est traduit, on voit surtout comment un appel à une fonction est traduit : un simple *call fonction* (avec, toutefois, auparavant la préparation du changement de contexte et ensuite la récupération du résultat, ce n'est pas si simple...).

La compilation de ce code assembleur produira un fichier objet suffixé par `.o` contenant la traduction binaire des instructions ci-dessus au format du micro-processeur.

Ce fichier objet ne sera pas exécutable bien que le code contenu sera formé d'instructions machine valides. Il faudra terminer le travail en *reliant* ce code avec celui des fonctions appelées : ce sera l'opération d'édition de liens.

▶ Continuons la compilation

▶ premier essai :

```
$ cc -o essai essai.s  
/tmp/ccx1wkM6.o: In function 'main':  
/tmp/ccx1wkM6.o(.text+0x1b): undefined reference to 'log'  
collect2: ld returned 1 exit status
```

- ▶ Erreur! pas de référence à log
- ▶ la référence à printf semble trouvée
- ▶ on voit que la commande ld a été appelée

▶ second essai :

```
$ cc -o essai essai.s -lm
```

- ▶ Pas d'erreur
- ▶ Mais on a ajouté une référence à une bibliothèque : libm.a

La première tentative de compilation provoque une erreur sur la référence manquante à la fonction `log()` appelée dans notre programme. Mais soyons plus précis, il ne s'agit plus ici de compilation mais d'édition de liens. La phase compilation est en effet terminée, un fichier objet a été produit. On recherche maintenant à résoudre les références trouvées dans ce fichier objet en recherchant, entre autre, les fonctions dans des bibliothèques.

On remarque que la fonction `printf()` ne pose pas de problème. Sa référence est donc résolue. Mais pourquoi ?

L'édition de lien par défaut prend toujours en compte la bibliothèque standard C, la `libc`, dans laquelle on trouve le code compilé des fonctions standard du langage C. Par contre il n'en va pas de même pour la bibliothèque mathématique par exemple. Il faut préciser qu'elle est nécessaire dans la ligne de compilation.

- ▶ Indication des bibliothèques et de leur emplacement
 - ▶ par défaut ld recherche les bibliothèques dans `/lib` et `/usr/lib`
 - ▶ on peut lui indiquer d'autres répertoires via l'option `-L`
`-L/local/src/appli/lib`
 - ▶ le nom des fichiers bibliothèques commence toujours par `lib` et se termine par `.a` ou `.so` ou `.so.x.y` (où `x` et `y` sont des numéros de version)
 - ▶ on indique les bibliothèques à l'aide de l'option `-l` suivie par le nom de la bibliothèque sans le préfixe `lib` et sans suffixe : `-lm` pour indiquer `/usr/lib/libm.a`

- ▶ Bibliothèques statiques
 - ▶ Fichiers suffixés par `.a` : `libm.a`
 - ▶ Lorsque ces bibliothèques sont utilisées, le code des fonctions qui en sont extraites est ajouté directement dans les exécutable au moment de l'édition de liens.
- ▶ Bibliothèques dynamiques
 - ▶ Fichiers suffixés par `.so` : `libm.so`
 - ▶ Lorsque ces bibliothèques sont utilisées, les fonctions qui en sont extraites sont simplement référencées dans les exécutable résultant de l'édition de liens. Le chargement effectif se fait lors de l'exécution.
 - ▶ Ces bibliothèques sont partageables (`.so` pour *sharable object*)

Partageable : le code de la bibliothèque n'est présent qu'une seule fois en mémoire, mais plusieurs applications peuvent l'utiliser. À l'opposé, dans le cas de bibliothèques statiques, le code est recopié dans chaque application qui l'utilise.

- ▶ Par défaut lors de l'édition de liens, les bibliothèques dynamiques sont recherchées d'abord, puis si elles ne sont pas trouvées les bibliothèques statiques sont recherchées
- ▶ Forçage du type d'édition de liens (man ld)
 - ▶ `-static`
 - ▶ `-dynamic` (option par défaut)

Édition de liens - Exemples

```
$cc -o essai essai.c -lm
$ls -l essai
-rwxrwxr-x 1 clohr clohr 13891 avr 24 11:13 essai
$ ldd essai
    libm.so.6 => /lib/i686/libm.so.6 (0x40033000)
    libc.so.6 => /lib/i686/libc.so.6 (0x40056000)
    /lib/ldlinux.so.2 => /lib/ldlinux.so.2 (0x40000000)

$ cc -o essai1 essai.c -static -lm
$ ls -l essai1
-rwxrwxr-x 1 clohr clohr 1701498 avr 24 11:13 essai1
$ ldd essai1
not a dynamic executable
```

- Remarquer la différence de taille entre les deux excutables pour une même programme

- ▶ Bibliothèques statiques
 - ▶ compilation des sources :
`cc -c *.c`
 - ▶ construction de la bibliothèque :
`ar r libxyz.a *.o`
- ▶ Bibliothèques dynamiques
 - ▶ compilation des sources :
`cc -c -fPIC *.c`
 - ▶ création de la bibliothèque :
`cc -o libxyz.so -shared -fPIC *.o`

- ▶ Ne pas confondre fichiers `.h` et ... bibliothèques...
- ▶ Les fichiers `.h` (`#include`) contiennent du code source C, des définitions de constantes, des spécifications de fonctions, des macros. Ce sont des fichiers d'entête des autres sources
- ▶ Les fichiers d'entête sont pris en compte au cours de la première phase de la compilation : **pre-processing**
- ▶ les fichiers bibliothèques (`.a` ou `.so`) sont pris en compte *après* la compilation, au moment de l'édition de liens

- ▶ Pre-processing : `/usr/bin/cpp`
 - ▶ Inclusion des fichiers `.h` spécifiés
 - ▶ substitution des `#define` par leurs valeurs
 - ▶ prise en compte des `#ifdef` et autres directives
- ▶ Compilation : `cc` (lié à `gcc`)
 - ▶ traduction du code source en instructions micro-processeur
 - ▶ le produit est placé dans un fichier objet `.o`
- ▶ Edition de liens : `ld`
 - ▶ création de l'exécutable par association de tous les fichiers objets résultant de la compilation et des fonctions des bibliothèques

```
cc -o appli f1.c f2.c ... fx.c -Ireinclude1 -Ireinclude2  
-Lreplib1 -Lreplib2 -labc -ldef ...
```

- ▶ `-o` pour indiquer le nom de l'exécutable, sans cette option il se nommera `a.out`
- ▶ `-I` pour indiquer un répertoire où trouver les fichiers d'entête si celui-ci n'est pas standard (`/usr/include`)
- ▶ `-L` pour indiquer un répertoire où trouver les bibliothèques si celles-ci ne se trouvent pas dans un répertoire standard (`/lib`, `/usr/lib`)
- ▶ `-labc` pour indiquer de prendre en compte la bibliothèque `libabc.so` si elle existe ou `libabc.a` sinon

Le profiling avec la commande `gprof`

83/169

- ▶ Le source doit être compilé avec l'option `-pg`
- ▶ L'exécutable doit se terminer par un `exit()`
- ▶ Un fichier `mon.out` est produit lors de l'exécution qui contient le résultat du profilage
- ▶ La commande `gprof` (avec le nom de l'exécutable en argument) affiche les résultat des appels aux fonctions internes du programme

L'outil `gdb`

84/169

- ▶ Un outil puissant de débogage (Gnu DeBugger)
- ▶ Les sources doivent être compilés avec l'option `-g` afin que l'exécutable contienne une table de correspondance entre les noms des variables et fonctions et leur représentation interne
- ▶ Une interface graphique existe : `ddd`
- ▶ L'outil est facilement appelable depuis emacs : `<M-x>`

Notons que les outils `strace` et `gdb` sont implémentés en utilisant l'appel système `ptrace(2)` qui permet à un processus d'accéder à l'espace mémoire d'un autre (et accessoirement de faire de l'injection de code). À l'opposé, `gprof` s'attend à ce que le processus racompte lui-même ce qu'il fait (compilé avec `-pg`).

Note : l'injection de code avec `ptrace` permet des trucs rigolo (e.g. `retty` ou `reptyr`), mais introduit des problèmes de sécurité (e.g. le recours à `prctl(PR_SET_DUMPABLE, 0)` dans `ssh-agent...`).

- ▶ Surveiller l'utilisation de `malloc()` et `free()`
- ▶ Solution glibc :

- ▶ Ajouter dans le code :

```
...
#include <mcheck.h>
...
mtrace();
/* malloc() free() à surveiller */
muntrace();
```

- ▶ Positionner la variable d'environnement :
`export MALLOC_TRACE=fichier_trace.txt`
- ▶ Compiler et exécuter :
`$ gcc -g -o prog sources.c ... ; prog`
- ▶ Analyser à l'aide du script perl :
`$ mtrace prog $MALLOC_TRACE`

Pour aller plus loin, on peut utiliser des frameworks comme *Valgrind* (<http://valgrind.org/>) ou *cmoka* (<https://cmocka.org/>).

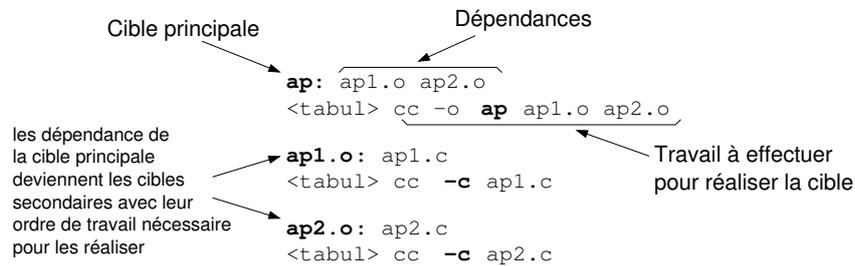
6 L'utilitaire Make

6.1 automatisation des phases de compilation

- ▶ Soient les fichiers `ap1.c`, `ap2.c` à compiler pour obtenir l'exécutable `ap`.
 - ▶ Commande normale : `cc -o ap ap1.c ap2.c`
 - ▶ Si on modifie ensuite un des fichiers il faut tout recompiler.
- ▶ L'outil `make` permettra de ne recompiler que ce qui est nécessaire :
 - ▶ compilation des fichiers sources plus récents que les fichiers objets qui leur correspondent
 - ▶ `make` utilise les directives indiquées dans un fichier `Makefile` utilisé implicitement ou dont le nom est fourni en argument. Ce fichier contient une liste de cibles à construire et les dépendances de ces cibles.

Un fichier makefile simple

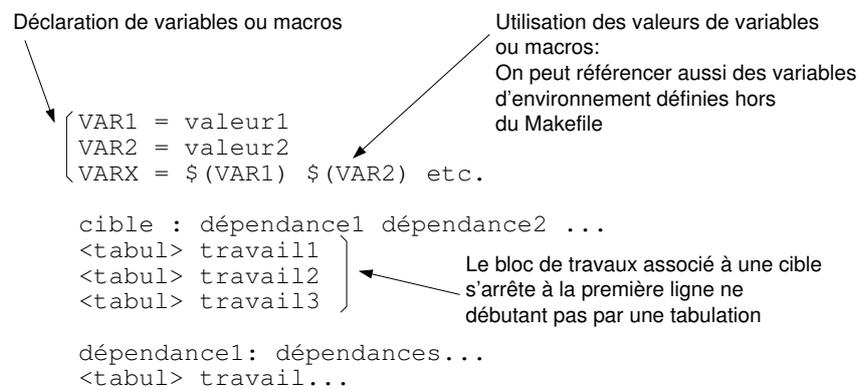
88/169



- ▶ Le travail associé à une cible n'est effectué que :
 - ▶ si une des dépendances est plus récente que la cible,
 - ▶ si le fichier de même nom que la cible n'existe pas.

Syntaxe générale d'un fichier Makefile

89/169



- ▶ Un bloc de travail typique :

```
<tabul> travail1
<tabul> travail2
<tabul> travail3
```

- ▶ Une ligne de travail peut contenir une liste de tâches séparées par des caractères «;»

```
cible: dépendance1 dépendance2 ...
<tabul> travail1; travail2; travail3
```

- ▶ Toute la ligne est exécutée à partir d'un même processus Shell, au retour de la dernière tâche de la ligne on revient dans le processus make d'origine. Ainsi on peut faire :

```
cible_x: ...
<tabul> cd rep1; premier travail ici; second travail ici; etc.
<tabul> cd rep2; premier travail là; second travail là; etc.
```

- ▶ rep2 n'est pas contenu dans rep1 mais dans le répertoire contenant ce makefile. Il y retour au pertoire d'origine à la fin de la première ligne. Il est possible de continuer une ligne sur une suivante en terminant la première par le caractère \

```
cible_x : ...
<tabul> cd rep1; \
<tabul> premier travail ici; \
<tabul> second travail ici; etc.
<tabul> cd rep2; premier travail là; second travail là; etc.
```

- ▶ Liste des variables et macros prédéfinies et des règles implicites : `make -p`
- ▶ Les variables les plus courantes :
 - `CC` indique le compilateur
 - `CPPFLAGS` fournit les options pour le préprocesseur (-I... , -D...)
 - `CFLAGS` fournit les options pour le compilateur (-g, -O, ...)
 - `LDFLAGS` fournit les options pour l'éditeur de liens (-L... , -l...)
- ▶ Ces variables peuvent ne pas être utilisées, elles ne sont pas préaffectées (sauf `CC`), si on veut les utiliser il faut leur donner une valeur dans le `Makefile`

- ▶ Pour compiler
`COMPILE.c= $(CC) $(CFLAGS) $(CPPFLAGS) -c`
- ▶ Pour réaliser la compilation et l'édition de liens
`LINK.c= $(CC) $(CFLAGS) $(CPPFLAGS) $(LDFLAGS)`

Exemple d'utilisation des macros prédéfinies 94/169

```
CPPFLAGS = -I../include -D__POSIX__
CFLAGS =
LDFLAGS = -Llib -L../share/lib -labc -lxy -lm

appli: ap1.o ap2.o
<tabul> $(LINK.c) -o appli ap1.o ap2.o

ap1.o: ap1.c
<tabul> $(COMPILE.c) ap1.c

ap2.o: ap2.c
<tabul> $(COMPILE.c) ap2.c
```

Attention à ne pas oublier un retour à la ligne en fin de fichier.

Les tabulations <tabul> sont extrêmement importantes et ne doivent pas être remplacées par des espaces, sans quoi l'outil **make** n'arrivera pas à interpréter votre fichier **Makefile**.

Les règles implicites

95/169

- ▶ Les règles implicites permettent de généraliser en un nombre restreint de directives des listes de travaux répétitifs
- ▶ Exemple : directive pour compiler de la même manière un ensemble de fichiers sources

- ▶ Pour make le nom d'un fichier est constitué par une racine suivie par un suffixe,
- ▶ Les suffixes possibles doivent être présents dans la liste implicite SUFFIXES ou dans la liste explicite (présente explicitement dans le Makefile)
.SUFFIXES
- ▶ Exemple : .SUFFIXES = \$(SUFFIXES) .z .Z
.gz
- ▶ Règles de type «.x.y» : retenir «*source* → *but*», donc «*source.but*»
- ▶ Question : .c.o ou .o.c ?

Réponse :

- La source : .c
- Le but : .o
- Donc : .c.o

- ▶ Dans le cas général d'une cible «.x.y»,
 - ▶ make recherche les fichiers dont le nom se termine par .y
 - ▶ il extrait la partie racine du nom et applique la règle sur cette racine complétée par le suffixe .x, (make vérifie la dépendance du *racine.y* par rapport au *racine.x*)
 - ▶ tous les fichiers suffixés .x sont pris en compte l'un après l'autre
 - ▶ si un fichier .y n'existe pas mais que le fichier correspondant .x existe, alors la règle est appliquée et le fichier .y est construit.

Exemple de travail avec les règles implicites 98/169

- ▶ Soit l'exemple suivant :
ap: ap1.o ap2.o
<tabul> \$(LINK.c) -o appli ap1.o ap2.o
.c.o:
<tabul> \$(COMPILE.c) \$<
- ▶ Le Makefile va être traité comme suit :
 - ▶ Première dépendance : ap1.o
 - ▶ Existe-t'il une règle pour construire ap1.o? → oui, c'est la règle .c.o
 - ▶ Cette règle implique qu'il faut rechercher un fichier de même racine mais avec le suffixe .c. Existe-t'il un fichier ap1.c? → oui, alors appliquons lui la suite de la règle : ap1.c est-il plus récent que ap1.o? (réponse OUI si ap1.o n'existe pas encore)
 - ▶ Si la réponse est OUI, alors la partie travail est effectuée, le \$< est remplacé par le nom de la dépendance : ap.c
 - ▶ Puis le contrôle remonte sur la cible principale et make s'occupe de la suite des dépendances, donc de ap2.o et le cycle recommence.
- ▶ Pour terminer la cible principale, les dates des dépendances sont vérifiées par rapport à la date de la cible et le travail associé est éventuellement exécuté.

Lisez lentement... :-)

Notation pour les règles implicites

99/169

- \$< le nom du fichier dépendant déterminé par make
- \$@ le nom de la cible courante
- \$? la liste des dépendances plus récentes que la cible
- * le nom de la cible sans son suffixe
- % lors de la construction de bibliothèques, désigne l'élément à traiter

- ▶ `prefixe%suffixe` :
 - ▶ toute référence, commençant par «`prefixe`» et se terminant par «`suffixe`» et contenant entre les deux un nombre quelconque de caractères, correspond à la règle.
 - ▶ Exemple :

```
ap%.o: ap%.c
```

`<tabul> $(CC) -c $<`

À la cible `apxyz.o` correspond la dépendance `apxyz.c`.
(Les caractères `xyz` sont les mêmes des deux cotés).

Compléments sur les macros et les règles implicites

- ▶ Exemple :

```
SOURCES = ap1.c ap2.c ap3.c
```

```
OBJETS = $(SOURCES:.c=.o)
```

équivalent à `OBJETS = ap1.o ap2.o ap3.o`
- ▶ `make -p` montre que `.c.o` est une règle implicite, donc connue de `make`, donc il est généralement inutile de la faire figurer explicitement. Vérifier malgré tout que les macros implicites associées soient en accord avec ce que l'on veut faire (`CCPFLAGS`, `CFLAGS`, ...)
- ▶ `make -p` montre que `.c` est aussi une règle implicite. On peut donc utiliser `make` sur un fichier `.c` (`make truc` si `truc.c` existe) et l'exécutable correspondant est construit. Pas besoin de `Makefile` pour cela.

Difficile de gérer les modifications possibles des fichiers d'entête (`xxx.h`)

► Utilisation du préprocesseur : `gcc -MM`

- Cette commande recherche récursivement tous les fichiers d'entête pouvant être inclus dans des fichiers sources dont la liste est passée en paramètre. Généralement on sauve le résultat dans un fichier annexe (p.ex. `Makefile.dep`), que l'on référence dans le `Makefile` avec une clause `include`

- En général il est généré via une cible spécifique dans le `Makefile` :

```
Makefile.dep: $(SOURCES)
<tabul> $(CC) $(CFLAGS) -MM $^ > $@
include Makefile.dep
```

Note : la clause `include` est une spécificité du `make` de GNU (qui est tout de même largement répandu). S'il existe une règle pour fabriquer le(s) fichier(s) indiqué par ce `include`, `make` l'applique (après avoir comparé les dates suivant la mécanique habituelle).

L'exemple donné ici convient bien pour les projets de petite ou moyenne taille : on recalcule les dépendances pour toutes les `$(SOURCES)` à la fois. Si le projet est gros, on peut vouloir recalculer les dépendances pour chacun des fichiers `.c` indépendamment des autres. Voir l'option `gcc -MMD` pour cela : le compilateur compile tout (il ne s'arrête pas au pré-processeur comme `gcc -MM`), et en même temps fabrique un fichier de dépendance `fichier.d` pour chaque `fichier.c`, qui pourra être utilisé pour le coup d'après à l'aide de clauses `include`. Par contre on n'ajoute pas de règle spécifique pour les générer dans le `Makefile`. Cela implique donc que les fichiers `.h` ne doivent pas trop changer d'une fois sur l'autre, puisque les dépendances ne seront considérées que lors d'un `make` ultérieur...

Historiquement on utilisait la commande `makedepend` qui recherche également les dépendances des fichiers `.h`, et place le résultat à la fin du fichier `Makefile` lui-même (pratique si `make` ne comprend pas les `include`). On avait alors l'habitude d'ajouter une cible spéciale `make depend` pour l'appeler. (Donc un appelait deux fois `make` : `make depend`; `make all`.) Cette façon de faire n'est plus trop utilisée.

Les cibles indépendantes

103/169

- ▶ `clean` : pour nettoyer l'arborescence des fichiers produits par un `make` normal précédent. Exemple :

```
clean:  
<tabul> rm *.o ....
```

- ▶ `install` : pour installer le produit de la construction. Le travail à effectuer est décrit par un script shell ad-hoc, ou par la commande `install`.
- ▶ `all` : mot conventionnel, désigne souvent la première cible (celle considérée par défaut), cible à laquelle sont associées des dépendances (elles mêmes cibles) permettant de tout construire.
- ▶ En toute rigueur on se doit de lister ces cibles indépendantes dans un règle : `.PHONY: all clean install ...`

Les options de make

104/169

- d affiche les dépendances et indique les raisons de la reconstruction d'une cible
- f `fichier_makefile` pour indiquer un fichier de type Makefile mais portant un autre nom (sans cette option `make` recherche d'abord le fichier `makefile`, puis Makefile)
- k permet de continuer le traitement pour les autres cibles si le traitement de l'une d'elle se termine en erreur
- n pour afficher les actions sans les exécuter
- p affiche toutes les macros (les règles implicites) et cibles
- q retourne un *status* 0 ou non selon que les cibles sont à jour ou non (testable en Shell avec la commande `echo $status` (csh) ou `echo $?` (Bourne Shell))
- s exécution silencieuse

7 Paquetages logiciels : rpm, debian, Gnu tar

7.1 gnu tar, debian, red hat, etc.

Logiciels sources au format général GNU 106/169

- ▶ Téléchargeables sous forme de fichier de type archives tar compressées avec gzip (.tgz, .tar.gz) ou bzip2 (.bz2)
 - ▶ `tar xvf paquetage`
- ▶ Contiennent un script de configuration et de création des Makefiles adaptés à l'architecture et à la version du système : `configure`
- ▶ Configuration, compilation, installation
 - ▶ `[bash]$./configure [--options]`
 - ▶ `[bash]$ make`
 - ▶ `[bash]$ make install`

Les paquetages logiciels Debian 107/169

- ▶ Trois niveaux d'utilitaires : `aptitude`, `apt`, `dpkg`
 - ▶ `dselect`/`aptitude`/`synaptic` offrent une interface texte ou graphique et permet de configurer les moyens de recherche des paquetages, de faire des suggestions, de les installer, les mettre à jour et les désinstaller
 - ▶ lorsque l'on connaît très exactement ce que l'on veut installer/désinstaller il est plus rapide d'utiliser les commandes `apt` : `apt-get`, `apt-cache`, ...
 - ▶ `dpkg` pour manipuler un fichier de paquetage déjà sur le disque, ex. : lister le contenu d'un paquetage : `dpkg -I nomDuPackage`

▶ Exemples :

```
linux# apt-cache search linuxconf
linuxconf - a powerful Linux administration kit
linuxconf-x - X11 GUI for Linuxconf
linuxconf-dev - Development files for Linuxconf
linuxconf-i18n - international language files for Linuxcon
linux#
```

Packages Debian - Installation d'un logiciel 109/169

```
linux# apt-get install linuxconf-x
Reading Package Lists... Done
Building Dependency Tree... Done
The following extra packages will be installed:
libwxxt1
The following NEW packages will be installed:
libwxxt1 linuxconf-x
0 packages upgraded, 2 newly installed, 0 to remove and 0 not upgraded.
Need to get 532kB of archives. After unpacking 1438kB will be used.
Do you want to continue? [Y/n] Y
Get:1 ftp://172.16.19.2 stable/main libwxxt1 1.67c-6 [486kB]
Get:2 ftp://172.16.19.2 stable/main linuxconf-x 1.17r5-2 [45.8kB]
Fetched 532kB in 1s (499kB/s)
Selecting previously deselected package libwxxt1.
(Reading database ... 30948 files and directories currently installed.)
Unpacking libwxxt1 (from ../libwxxt1_1.67c-6_i386.deb) ...
Selecting previously deselected package linuxconf-x.
Unpacking linuxconf-x (from ../linuxconf-x_1.17r5-2_i386.deb) ...
Setting up libwxxt1 (1.67c-6) ...
Setting up linuxconf-x (1.17r5-2) ...
linux#
```

- ▶ La commande `rpm`
- ▶ Permet d'installer (`-i`) ou de supprimer (`-e`) des logiciels :
 - ▶ `rpm -ivh nom_du_package`
 - ▶ Le nom du package peut être une URL
- ▶ Gère les dépendances entre logiciels (entre bibliothèques) : refuse d'installer si une dépendances n'existe pas (forçage possible mais dangereux)
- ▶ Permet de s'informer sur un *package*, savoir ce qu'il contient, de retrouver à quel *package* appartient tel fichier, de connaître les *packages* installés

- ...
- ▶ Permet de créer un *package* à partir d'une arborescence source compilée
- ▶ gestion de la base installée : `/var/lib/{rpm | rpm.rpmsave}`
- ▶ Commande `yum` recherche, télécharge et installe un paquetage

Exemples :

— À quel paquetage appartient la commande `ls` ?

```
[linux]# rpm -qf /bin/ls
```

```
fileutils-4.0-1
```

- Le package NFS est-il installé?

```
[linux]# rpm -q nfs
```

```
package nfs is not installed
```

En fait, le paquetage NFS porte un nom plus complexe et est peut être installé malgré tout. Essayons avec l'option `-a` qui permet, en mode *query* (option `-q`) de lister tous les paquetages installés.

```
[linux] rpm -qa | grep nfs
```

```
nfs-utils-clients-0.2.1-2mdk
```

```
nfs-utils-0.2.1-2mdk
```

- Listons le contenu du paquetage `nfs-utils-clients-0.2.1-2mdk`

```
[linux]# rpm -ql nfs-utils-clients-0.2.1-2mdk
```

```
/etc/rc.d/init.d/nfslock
```

```
/usr/sbin/rpc.lockd
```

```
...
```

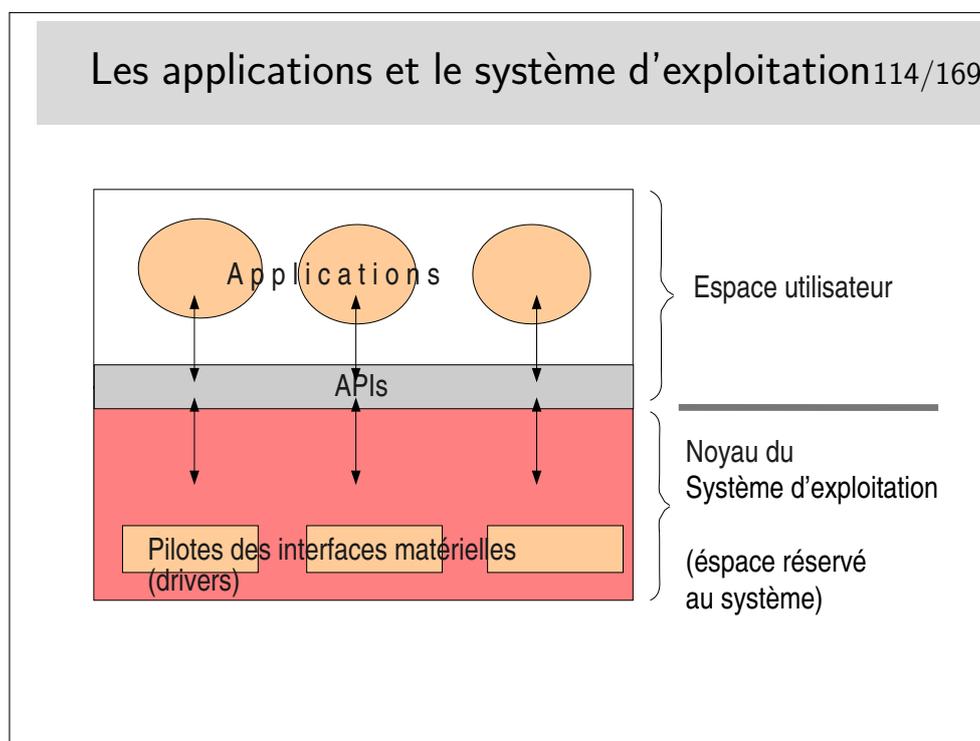
Où trouver les paquetages RedHat

112/169

- ▶ Sur les CD-ROM d'installation
 - ▶ si monté à l'endroit standard :
 - ▶ `/mnt/cdrom/Redhat/RPMS`
- ▶ Sur le web
 - ▶ `http://www.rpmfind.com`
- ▶ Outils systèmes
 - ▶ `gnorpm`
 - ▶ `yum`
 - ▶ ...

8 Programmation d'applications Réseau

8.1 Concepts généraux



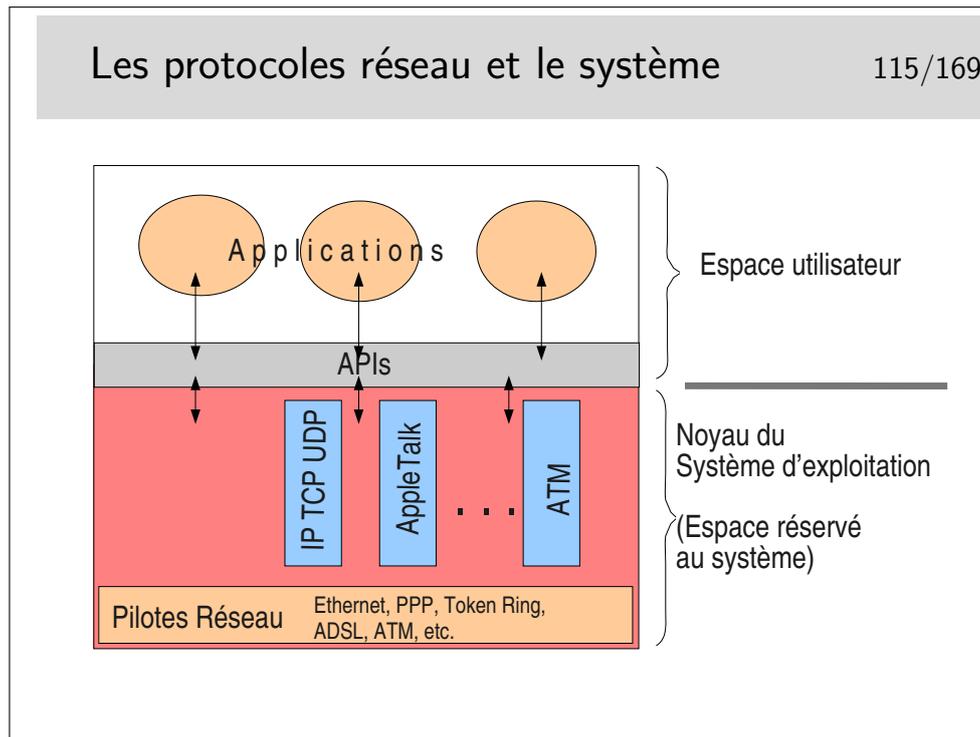
Une application informatique s'exécute dans la mémoire centrale de l'ordinateur (la RAM), sous contrôle du système d'exploitation, lui aussi chargé dans la mémoire centrale. Laissons nous imaginer que l'ordinateur ne soit représenté que par sa mémoire centrale. C'est ce que veut représenter la figure ci-dessus : imaginez que le rectangle extérieur englobant est cette mémoire RAM.

L'espace mémoire RAM disponible est divisé en deux parties : d'une part la partie occupée par le système d'exploitation lui-même, l'espace «noyau» (qu'on appellera pour faire simple le «noyau»), d'autre part la partie où peuvent s'exécuter les applications qu'on appellera l'espace utilisateur.

Pour accéder aux services des divers matériels périphériques (disques, clavier, souris, écran, et pour notre propos, réseau) le noyau possède des modules logiciels spécifiques de ces périphériques, des modules pouvant dialoguer directement avec eux qu'on appelle des pilotes de périphériques, en anglais des «drivers».

Les applications ne communiquent pas directement avec les pilotes de périphériques, elles communiquent avec le noyau, à l'aide de fonctions spécifiques (qu'on nomme des «appels système» ou parfois des primitives). Ces fonctions sont regroupées dans des bibliothèques de fonctions qu'on nomme des «APIs» (*Application Programmer's Interface*).

Ces fonctions sont «génériques» dans le sens où elles masquent la réalité matérielle. Le matériel devient une abstraction. L'API est une couche d'abstraction. Pour ce qui nous concerne en réseau et pour faire simple, l'API Réseau que nous utiliserons nous présentera le réseau comme un fichier et nous enverrons des données vers un destinataire comme nous écrivons dans un fichier (le réseau est devenu un concept abstrait).



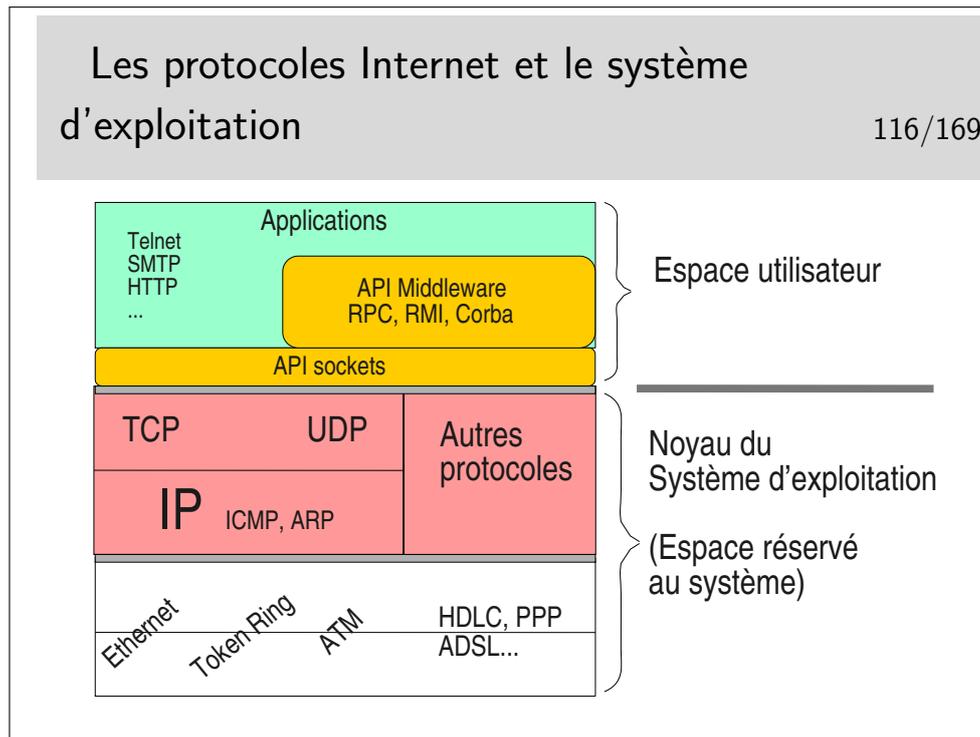
Pour ce qui nous concerne, en réseau, il est fort heureux que les protocoles majeurs soient directement implémentés au cœur des noyaux des systèmes d'exploitation. Les APIs sont là pour nous offrir une couche d'abstraction de ces protocoles. Rassurez vous, pour communiquer avec des protocoles standards tels que TCP-IP vous n'aurez pas à créer de toute pièce les datagrammes IP ni les segments TCP, ni à contrôler le fonctionnement de ces protocoles (pensez à la machine d'états finis TCP, en fait, non, n'y pensez plus). Il vous suffira d'utiliser une fonction de l'API pour ouvrir un point d'accès, de le paramétrer correctement (là il vous faudra connaître l'existence de quelques aspects du protocole utilisé mais pas beaucoup), ensuite ce point d'accès sera banalisé et vous le traiterez comme un fichier...

Magique... Presque...

Il vous faudra quand même connaître les arcanes de la programmation système en langage C ou C++ sous Unix/Linux ou Windows ou MacOS. Vous pourrez négliger ces derniers aspects si vous programmez en Java, mais alors il vous faudra maîtriser Java. Si Java ne vous plaît pas il vous reste Perl, Python, et bien d'autres langages de haut niveau comprenant de manière intrinsèques des fonctionnalités Réseau.

Les protocoles Internet et le système d'exploitation

116/169



Pour ce qui est des protocoles Internet en particulier ce qui a été dit précédemment s'applique tout à fait aux protocoles des niveaux 3 et 4 OSI, donc à IP et TCP et UDP.

Ces derniers sont inclus dans le système d'exploitation et sont accessibles via des fonctions de la bibliothèque socket (inclue dans la bibliothèque standard du langage C sous Linux et bibliothèque standard WinSock (ws32) sous Windows). Les «sockets» vont nous donner une abstraction du réseau en nous le présentant comme un fichier (il est facile d'écrire ou de lire un fichier, il sera facile d'écrire ou de lire le réseau, c'est à dire, en pratique, d'envoyer ou de recevoir des octets via le réseau).

Les sockets restent cependant des outils de bas niveau, adaptées au langage C ou C++ de base et il faut construire «à la main» les protocoles applicatifs que l'on veut mettre en œuvre (les protocoles des applications, pensez au mail (smtp), au web (http), etc.). Ces protocoles s'appuient sur TCP-IP pour véhiculer leurs PDUs, il faut cependant construire ces PDUs, ces fonctionnalités ne sont pas dans le noyau.

Les informaticiens ont cherché à renforcer l'abstraction réseau offerte par l'interface programmatique socket en créant tout d'abord le concept de procédure distante c'est à dire de fonction qu'on appelle en local mais qui s'exécute à distance : ce sont les RPC (Remote Procedure Call) dont on trouve un standard sous Unix (origine Sun) et un autre sous Windows (incompatible). Dans le même esprit on trouve les RMI en Java (*Remote Method Invocation*).

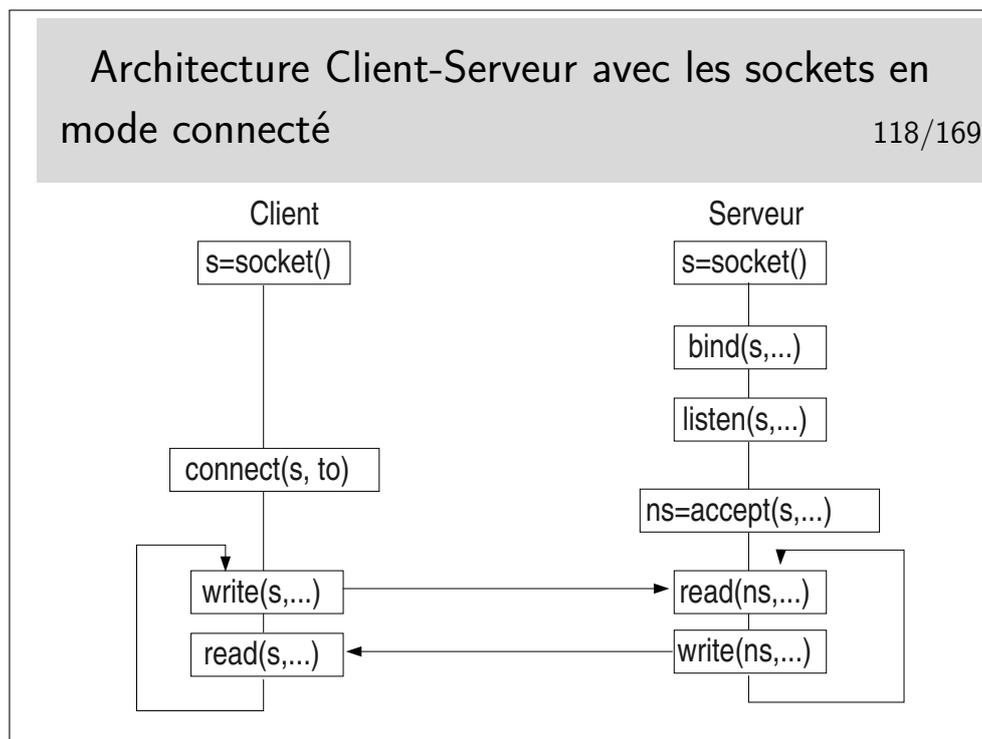
En poussant plus loin le concept de RPC, les informaticiens ont développé le concept de méthode distante (au sens méthode de langage objet). Cela a donné lieu à l'architecture d'informatique répartie CORBA (*Common Object Broker Architecture*), on invoque une méthode sur un objet quelque part, qu'on sait exister et que l'architecture sait situer. RMI Java et CORBA sont des concepts différents (RMI est plus proche de RPC) et il existe des implémentations CORBA pour Java.

On peut également mentionner (pour la culture générale) l'API réseau définie par le consortium X/Open : *The X/Open Transport Interface (XTI)*. Cette API se veut plus

générale que l'API socket BSD (qui est tout de même très connue TCP/IP), et plus proche conceptuellement du modèle OSI. Mais fonctionnellement, ça fait un peu la même chose. Cette API est disponible sur la plupart des Unix (Linux, Solaris, BSD, etc.), mais est rarement employée dans les programmes. http://en.wikipedia.org/wiki/X/Open_Transport_Interface.

Notons le travail de l'IETF dans le RFC 8303 «On the Usage of Transport Features Provided by IETF Transport Protocols» qui dresse un panorama des différents services fournis par les différents protocoles de transport normalisés à l'IETF (car il n'y a pas que TCP et UDP dans la vie)... Un travail qui aboutira peut être à la définition d'API de programmation réseau plus générique que l'API socket actuelle.

8.2 L'API Socket



— Coté serveur :

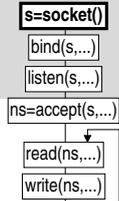
- création de la socket : fonction **socket()**, la valeur rendue, **s**, «est» la socket. Sous Unix/Linux c'est ce qu'on appelle un descripteur de fichier (donc un pseudo fichier), sous Windows c'est un type spécifique **SOCKET**.
- fonction **bind()** : permet d'associer (to bind = lier) une adresse à la socket précédemment créée (en IP on associera une adresse IP en un port TCP ou UDP)
- fonction **listen()** : positionne la couche protocolaire immédiatement inférieure en mode serveur (dans le cas de TCP, la machine d'états finis passe en état LISTEN)
- fonction **accept()** : on attend des requêtes de connexion
- **read()**, **write()** : les fonctions de dialogue avec le client

- Coté client :
 - création de la socket... Comme dans le serveur
 - fonction **connect()** : on se connecte au serveur. Ici on résume car bien évidemment il faut passer l'adresse du serveur en argument. Nous verrons les détails plus loin.
 - **write()** et **read()** : on parle au serveur (et on écoute ensuite ce qu'il dit...)

Remarque : pas de **bind()** coté client, ce n'est pas utile. Lors du **connect()** le système attribuera une adresse à la socket. Il n'est pas cependant pas interdit de faire un **bind()**.

Communication : le client écrit sur la socket, le serveur lit la socket... Ou l'inverse. C'est au programmeur de décider qui parle le premier. C'est au programmeur d'écrire le dialogue, mais ce n'est pas un dialogue de pièce de théâtre ou de film, il s'agit d'un **protocole** applicatif (et c'est moins drôle).

La fonction socket()



```

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

int socket(int domain, int type, int protocol);
  
```

Permet d'obtenir un point d'accès aux couches protocolaires de communication désirées

- ▶ La valeur renvoyée (**int**) est l'identificateur de ce point d'accès, c'est «la socket»
- ▶ **domain** : PF_UNIX, PF_INET, PF_INET6
- ▶ **type** : SOCK_STREAM, SOCK_DGRAM, SOCK_RAW
- ▶ **protocol** : permet d'indiquer le protocole à utiliser s'il n'est pas implicite avec le domaine et le type

Le préfixe PF (*Protocol Family*) du paramètre domain peut être remplacé par le préfixe AF (*Address Family*) lorsque l'on définit l'adresse de la socket (notons que ce sont ces macros ont les mêmes valeurs numériques dans AF et dans PF).

- Le domaine PF_UNIX est utilisé pour ouvrir des sockets qui serviront à des communications entre des processus locaux à une machine Unix.
- Le domaine PF_INET indique des sockets IPv4.
- Le domaine PF_INET6 indique une utilisation du protocole IPv6.
- Le domaine PC_NETLINK pour dialoguer avec la couche réseau interne à Linux (p.ex. éditer la table de routage, le firewall, etc.)
- Et bien d'autres domaines encore PF_BLUETOOTH PF_IRDA PF_MPLS etc.

Le type `SOCK_DGRAM` indique que le mode de communication sera sans connexion, et que les messages seront des datagrammes.

Le type `SOCK_STREAM` indique un mode de communication orienté flot d'octet, avec connexion.

Le type `SOCK_RAW` indique que l'on court-circuite la couche immédiatement inférieure.

Le champ protocole est la plupart du temps mis à 0 car il est implicite, `PF_INET` et `SOCK_STREAM` impliquent TCP, `PF_INET` et `SOCK_DGRAM` impliquent UDP.

On peut cependant utiliser `IPPROTO_TCP` ou `IPPROTO_UDP` pour être explicite (il faut inclure `<netinet/in.h>`).

Informations complémentaires sous linux : `man 7 ip`, et `man 7 ipv6`

La valeur rendue par `socket()` est un descripteur de fichier (sous Unix/Linux) et un type `SOCKET` sous Windows.

La fonction `bind()`

`s=socket()`
`bind(s,...)`
`listen(s,...)`
`ns=accept(s,...)`
`read(ns,...)`
`write(ns,...)`

```
int bind(int sockfd, struct sockaddr *my_addr, socklen_t addrlen);
```

- ▶ Permet d'associer (lier) une adresse à la socket indiquée en premier paramètre
- ▶ Le format de l'adresse dépend du domaine de la socket, il diffère entre `AF_UNIX`, `AF_INET` et `AF_INET6` par exemple
- ▶ Le type du second argument doit être adapté à celui du domaine utilisé (voir exemple suivant)

```

struct sockaddr_in sin;
int sfd, port, r;
...
sfd = socket(PF_INET, SOCK_STREAM, 0);
port = 7890;
sin.sin_family=AF_INET;
sin.sin_addr.s_addr=INADDR_ANY;
sin.sin_port=htons(port);
r = bind(sfd, (struct sockaddr *)&sin, sizeof(sin));
if (r < 0) {
    perror("bind");
    ...
}

```

force le type

La socket est ouverte en `PF_INET` (IPv4), sa structure d'adresse sera de type `struct sockaddr_in`. Pour de l'IPv6 on aurait `PF_INET6` et `sockaddr_in6`.

La fonction `bind()` n'est pas réservée aux protocoles de la famille IP, elle peut être utilisée avec d'autres. Sa spécification indique que le second paramètre est de type `struct sockaddr *`. En IP nous devons utiliser le type `struct sockaddr_in` ou `sockaddr_in6`. Le compilateur ne sera pas content et nous donnera une alerte (un Warning s'il parle anglais). Pour contenter le compilateur on fait un forçage de type (un «cast» en anglais).

Cette structure contient trois champs. Le premier, `sin_family` doit contenir `AF_INET` pour IPv4. Pour IPv6, le champ `sin6_family` contient `AF_INET6`.

Le second contient l'adresse IP à laquelle on associe la socket. Les connexions devront être adressées à cette adresse particulière. Ce point pose problème. En effet, on doit placer ici une adresse d'une des interfaces de la machine, or il peut en exister plusieurs (il en existe en général au moins deux, une adresse associée à l'interface physique et une adresse de boucle locale, 127.0.0.1). De plus, en IPv6 on a potentiellement plusieurs adresses pour une interface : une adresse qui a un *scope* de niveau lien (i.e. valable que sur le bus Ethernet), et un *scope* de niveau global (i.e. valable à travers le grand Internet).

Quelle adresse doit on indiquer ici ? Si on prend une des adresses de la machine, seuls les paquets à destination de l'interface portant cette adresse exacte seront reçus par cette socket et pas les autres. Par ailleurs, il n'est pas portable de fixer en dur dans le programme une adresse à la socket (mais on peut passer cette adresse dynamiquement par argument au programme par exemple).

Pour passer outre tous ces problèmes on peut utiliser la «méta-adresse», appelée adresse *joker*. C'est l'adresse `INADDR_ANY` en IPv4 (en fait `0.0.0.0`), ou bien `IN6ADDR_ANY_INIT` (en fait `:::`). Une socket «bindée» sur cette adresse acceptera toutes les connexions (en TCP) ou tous les messages (en UDP).

Le dernier membre important de la structure d'adresse est le numéro de port (TCP ou UDP). Remarquez ici, qu'il est indiqué via la fonction `htons()` pour l'indiquer dans un format «réseau» indépendant de l'architecture matérielle de la machine (problème des architectures *big endian* versus *little endian*, voir plus loin).

On ne peut pas attribuer un port déjà attribué.

Sous Unix/Linux, un utilisateur normal ne peut pas attribuer un port inférieur à 1024. Seul l'administrateur peut le faire (l'utilisateur *root*).

La structure d'adresse `sockaddr_in`

122/169

```
typedef uint32_t in_addr_t;
struct in_addr
{
    in_addr_t s_addr;
};

struct sockaddr_in
{
    sa_family_t sin_family;
    in_port_t sin_port;      /* Port number. */
    struct in_addr sin_addr; /* Internet address. */
};
```

La structure `sockaddr_in` est ici résumée à l'essentiel, voir sa définition complète dans `/usr/include/netinet/in.h` sous Unix/Linux (le champ `sin_family` y est défini, de manière complexe, via une macro, nous l'avons «traduit en clair» ci-dessus).

On remarquera que son champ `sin_addr` est en fait une structure ne contenant qu'un seul membre : `s_addr` qui est en fait un entier 32 bits non signé.

```
struct sockaddr_in6 {
    sa_family_t    sin6_family;    /* AF_INET6 */
    in_port_t      sin6_port;      /* numero de port */
    uint32_t       sin6_flowinfo;  /* flux IPv6 */
    struct in6_addr sin6_addr;     /* adresse IPv6 */
    uint32_t       sin6_scope_id;  /* Scope ID */
};

struct in6_addr {
    unsigned char  s6_addr[16];    /* adresse IPv6 */
};
```

Là aussi, la structure `sockaddr_in6` est ici résumée à l'essentiel. Consultez le `man ipv6` pour plus de détails.

On retrouve les mêmes champs que pour la structure `sockaddr_in`, excepté le champ `sin6_flowinfo` maintenant obsolète et laissé à 0, ainsi que le champ `sin6_scope_id` qui n'a d'intérêt que pour les les adresses locales de lien et qui est fréquemment laissé à 0.

Notez toute fois un problème de taille : la structure d'adresse générique `sockaddr` prend généralement 16 octets (comme la structure IPv4 `sockaddr_in`), alors que la la structure d'adresse IPv6 `sockaddr_in6` en consomme 28... Tant que l'on se passe des pointeurs, ce n'est pas un problème. Cependant, lorsque l'on veut stocker une adresse, on a un souci. Ainsi la norme POSIX a introduit une nouvelle structure : `sockaddr_storage`, qui peut donc être utilisée pour stocker indifféremment des adresses IPv4, IPv6, de socket Unix, etc.

La structure d'adresse sockaddr_un

124/169

```
#define UNIX_PATH_MAX    108

struct sockaddr_un {
    sa_family_t sun_family;           /* AF_UNIX */
    char        sun_path[UNIX_PATH_MAX]; /* chemin */
};                                     /* d'accès */
```

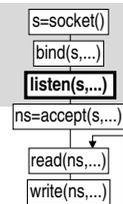
La structure d'adresse d'une socket Unix. (Ne pas oublier qu'il n'y a pas que des socket réseau dans la vie!)

Le `sun_path` est le nom du fichier spécial de type socket, qui est créé dans l'espace de fichier de la machine (c'est un vrai nom de fichier, avec un propriétaire, des permissions, une date, mais pas de taille!). Dans certains cas, on peut avoir des sockets *anonymes*, c'est à dire sans nom explicite (typiquement lors de l'utilisation de `socketpair()`).

La fonction listen()

```
int listen(int sfd, int backlog);
```

- ▶ Place la socket `s` en mode serveur
 - ▶ Si la socket est de type TCP, la machine d'états finis, associée à la socket dans la couche TCP, est placée dans l'état LISTEN
 - ▶ Le paramètre `backlog` indique la taille de la file d'attente des requêtes de connexion
 - ▶ Cette fonction n'est pas bloquante (comme son nom pourrait le faire penser)
- ▶ La socket ne pourra plus servir qu'à accepter des requêtes de communication, elle ne pourra pas servir pour les échanges de données



Le paramètre `backlog` ne limite pas le nombre de communications simultanées, il limite

le nombre de requêtes simultanées. Si n requêtes arrivent au même instant elles seront servies les unes après les autres, dans un temps de service non nul. Si $n < backlog$, tout va bien, toutes seront servies. Si $n > backlog$, alors seulement $backlog$ requêtes seront servies.

Si n requêtes arrivent ($n > backlog$) dans un intervalle de temps bien supérieur au temps de traitement d'une requête, la file d'attente ne se remplira pas complètement et le système aura le temps de traiter toutes les requêtes.

La question principale après ce discours est de savoir ce qu'il est bon d'indiquer comme valeur et existe-t'il une règle pour déterminer celle-ci ?

La réponse est : « euh !... »

...Donc, si vous pensez que votre serveur sera très sollicité vous mettrez 20, sinon vous mettrez 5 et vous testerez... (Notez que Linux utilise un minimum de 3.)

En toute rigueur on peut distinguer deux files d'attente : l'une pour le nombre de demandes de connexions incomplètes (TCP n'a pas terminé sa poignée de main en trois coups), et l'autre pour le nombre de demandes de connexions établies. C'est le cas dans les implémentations dans Linux, BSD et quelques autres OS. Dans ce cas, la file d'attente des connexions incomplètes (dont le comportement dépend en fait de l'état du réseau) est paramétrée par l'administrateur du système (p.ex. `/proc/sys/net/ipv4/tcp_max_syn_backlog`). Par contre, la file d'attente des connexions établies, et donc en attente que le programme les prennent par un `accept()`, dépend du programmeur qui paramètre la taille avec le `backlog`.

La fonction `accept()` |

`s=socket()`
`bind(s,...)`
`listen(s,...)`
`ns=accept(s,...)`
`read(ns,...)`
`write(ns,...)`

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

- ▶ Accepte des requêtes de connexion sur la socket s
 - ▶ Bloquante
 - ▶ Le paramètre `addr` est un pointeur sur la structure d'adresse de la socket distante (la socket appelante)
 - ▶ Le paramètre `addrlen` est un pointeur sur la longueur de cette structure d'adresse
- ▶ `accept()` rend une nouvelle socket, presque clone de la précédente qui servira à la communication

Cette fonction est bloquante, c'est à dire que le processus d'exécution du programme contenant l'appel `accept()` va bloquer sur cet appel. Le déblocage interviendra lorsqu'une requête de connexion sera reçue.

Il est possible de faire en sorte que `accept()` ne soit pas bloquant en agissant sur la socket via l'appel système (une fonction du système d'exploitation) `fcntl()` sous

Unix/linux. Dans ce cas, le `accept()` ne bloque pas et retourne immédiatement. S'il n'y a pas de requête à traiter, la valeur rendue est `-1` comme en cas d'erreur. Il faut alors examiner la variable externe `errno` pour vérifier s'il s'agit vraiment d'une erreur, s'il s'agit simplement du fait qu'il n'y a pas d'appel, `errno` vaudra la valeur `EAGAIN` ou `EWOULDBLOCK` (valeurs identiques en fait).

On ne peut accepter que sur une seule socket.

La fonction `accept()` II127/169

- ▶ Exemple d'utilisation en TCP-IP

```
int sfd, nsfd, fromlen;
struct sockaddr_in from;
...
fromlen = sizeof(from);
nsfd = accept(sfd, (struct sockaddr *)&from, &fromlen;
```

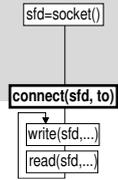
force le type
- ▶ `accept()` nous rend une nouvelle socket dont nous mémorisons la valeur dans la variable `ns`
- ▶ `nsfd` servira aux échanges de données

Dans l'exemple ci-dessus, on récupère l'identité de l'appelant dans la structure `from`. On pourra connaître ainsi l'adresse IP de la machine appelante et le numéro de port de l'application cliente.

La fonction connect() |

```
int connect(int sfd,  
            const struct sockaddr *serv_addr,  
            socklen_t addrlen);
```

- ▶ Établit la connexion d'une socket cliente *s* vers une socket serveur dont on passe l'adresse à l'aide du second argument



Dans l'architecture que nous donnons du client, on voit que nous ne faisons pas de `bind()` pour affecter une adresse à la socket. Or, si on travaille en TCP par exemple, on sait que le client doit être identifié par un port. Il faudrait donc en toute rigueur faire un `bind()`. On préfère laisser cette affaire au noyau du système d'exploitation. Lors du `connect()`, le système voit que la socket n'est pas liée (*binded*) à une adresse, alors il le fait, il attribue un port. Cela a l'immense avantage d'attribuer un port libre. Si nous faisons un `bind()` explicite il y a de forte chances qu'il faudrait de multiples tentatives avant un succès (on ne pas attribuer un port non libre).

```
1 int sfd, r;
2 struct addrinfo hints, *result;
3
4 memset(&hints, 0, sizeof(struct addrinfo));
5 hints.ai_family = AF_UNSPEC;
6 hints.ai_socktype = SOCK_STREAM;
7 hints.ai_protocol = 0;
8 hints.ai_flags = 0;
9
10 r = getaddrinfo(argv[1], argv[2], &hints, &result);
11 if (r != 0) { ... }
12 ...
13 r = connect(sfd, result->ai_addr, result->ai_addrlen);
14 freeaddrinfo(result);
```

Ce n'est qu'un exemple... Mais il marche. Et les structures à manipuler sont complexes... Faites donc comme tout le monde... Copiez/Collez (le **man getaddrinfo**) et adaptez ensuite à votre cas.

Ligne 4 : Tornade blanche sur la structure **hints** de type **struct socakadr**. On nettoie son contenu en y mettant des «0» (voir le manuel de référence pour **memset()**).

Ligne 5 : La structure **hints** paramètre la demande que l'on fait à la fonction **getaddrinfo()** utilisée juste après (qui effectue la résolution des noms de machine et de port). La valeur **AF_UNSPEC** indique que l'on veut aussi bien de l'IPv4 que de l'IPv6.

Ligne 6 : La valeur **SOCK_STREAM**, que ce soit en IPv4 ou en IPv6, c'est offert par le protocole TCP. A priori on veut donc une socket TCP...

Ligne 7 : Si l'on voulait forcer un autre protocole (à supposer qu'un autre soit possible), on le préciserait ici. Bon, on laisse 0.

Ligne 8 : Éventuellement on pourrait préciser notre demande. Voir le **man getaddrinfo**.

Ligne 10 : On récupère l'adresse IP du serveur sur lequel on veut se connecter. Le paramètre **argv[1]** correspond au premier argument de la ligne de commande qui sert à lancer le programme. On peut utiliser un autre paramètre, de toute manière ce sera une chaîne de caractères du type «www.quelquepart.com», «192.108.117.241» (IPv4), ou «2001:660:7302:2::11» (IPv6). L'adresse IP, sous forme numérique, sera accessible via le pointeur **result** rempli par la fonction **getaddrinfo()**. Le second paramètre **argv[2]** devra être une chaîne de caractère décrivant le *service* (ou numéro de port), par exemple «80» ou «http».

En fait, cette fonction **getaddrinfo()** nous retourne une liste de structures d'adresses possibles : une machine (ici le serveur) peut avoir plusieurs adresse tant en IPv4 qu'en IPv6. Idéalement, il faudrait donc parcourir cette liste et les essayer les unes après les autres.

Ligne 13 : Et hop, on se connecte en utilisant la structure d'adresse retournée par

`getaddrinfo()`. (Attention : dans cet exemple on utilise la première adresse retournée, potentiellement il y en a plusieurs à essayer.)

Ligne 14 : On libère la mémoire occupée par la liste chaînée `result` créée par `getaddrinfo()`.

La communication en mode connecté| 130/169

- ▶ Sous Unix/Linux : les fonctions `read()` et `write()`
- ▶ Comme pour écrire et lire des fichiers

```
graph TD; subgraph Box1; direction LR; W1[write(s,...)]; R1[read(ns,...)]; end; subgraph Box2; direction LR; R2[read(s,...)]; W2[write(ns,...)]; end; W1 --> R1; R1 --> W2; W1 --> R2; W2 --> R2;
```

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

En Unix/Linux les sockets **sont** des «file descriptors», des descripteurs de fichiers. Un descripteur de fichier est normalement obtenu via la fonction `open()` qui «ouvre» un fichier. Le descripteur sert ensuite de référence au fichier ouvert dans les fonctions de lecture et d'écriture que sont `read()` et `write()`.

La socket est donc un descripteur, le réseau (plus exactement le point d'accès au réseau qu'on se crée en créant la socket) est donc assimilé à un fichier. Lorsque la socket est connectée (coté client) ou lorsqu'une acceptation a eu lieu (coté serveur) il suffit d'écrire sur la socket pour envoyer des données et de la lire pour en recevoir.

Le premier argument de ces deux fonctions (ce sont des «appels systèmes») est le descripteur du fichier précédemment ouvert. Pour nous c'est donc la socket. Le second un un tampon mémoire (un «buffer» dans notre jargon) qui contient les octets à écrire (pour `write()`) ou qui va contenir les octets lus (pour `read()`). Le troisième argument est la longueur maximale que l'on écrira (pour `write()`) ou qu'on lira (pour `read()`).

Par défaut `write()` tente d'envoyer le nombre d'octets qu'on lui indique dans son troisième argument. Avec les sockets en mode connecté il se peut que le protocole mette en œuvre des mécanismes de contrôle de flux qui bloquent la transmission si le récepteur ne peut traiter les octets reçus au rythme soutenu auquel il les reçoit... C'est, par exemple, le cas de TCP. Si le protocole sous-jacent bloque la transmission avant que le nombre d'octets soumis par `write()` ne soit réellement envoyé, alors `write()` va bloquer.

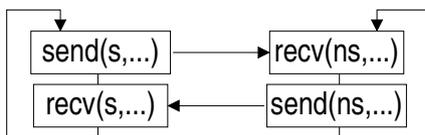
Par défaut, avec les sockets, `read()` bloque tant qu'on ne reçoit pas de données. Plus exactement `read()` bloque tant que la couche protocolaire sous-jacente ne nous remet pas les données qu'elle a reçu. Lorsque `read()` débloque on retrouve les données reçues

dans le tampon dont l'adresse constitue le deuxième argument.

Il est possible avec `fcntl()` de rendre les sockets non bloquantes.

La communication en mode connecté II 131/169

- ▶ Sous Windows et sous Unix/Linux les fonctions `send()` et `recv()`



```
ssize_t send(int sfd, const void *buf, size_t len,
             int flags);
ssize_t recv(int sfd, void *buf, size_t len, int flags);
```

- ▶ Fonctions identiques à `write()` et `read()`, avec en plus un flag spécifique
- ▶ Flag : `MSG_OOB`, `MSG_PEEK`, ...

Ces fonctions ont le même comportement que `read()` et `write()`. Le flag permet de préciser des spécificités réseau, ce que ne permettent pas les fonctions `read()` et `write()`.

Le flag `MSG_OOB` est utilisé pour l'envoi de données urgentes en TCP. `OOB` signifie « *Out Of Band* », en français : « données hors bande ». Les données ainsi envoyées en TCP ne sont nullement « hors bande », cette notion n'existant pas en TCP, mais elles sont « urgentes » (TCP connaît cette fonctionnalité).

Pour les recevoir il faut utiliser un `recv()` muni du même flag. À noter que ce n'est pas si simple, il faut que le processus récepteur ait averti le noyau du système d'exploitation qu'il s'attend à recevoir ce type de données urgentes. Lorsque ces données arriveront, alors il recevra un signal (une sorte d'interruption logicielle) qui lui commandera de se dérouter vers une routine de traitement dans laquelle il fera le `recv()`.

Complicé, non ? Rassurez vous (?) nous reverrons cela plus loin.

Le flag `MSG_PEEK`, en lecture, permet de lire sans que les données soient retirées du tampon de réception. On peut donc les retrouver ensuite avec un nouveau `read()` ou `recv()`.

Les flags décrits ici sont au standard POSIX. En linux ou Unix de type BSD il en existe d'autres (voir `man 2 recv` sur ces systèmes).

Sous Windows (bibliothèque `winsock-2`), les sockets ne sont pas des descripteurs de fichiers mais des types `SOCKET`. On ne peut donc pas utiliser les fonctions standards du C `read()` et `write()`.

Notez également les appels systèmes `sendmsg()` et `rrecvmsg()`, des appels systèmes de plus bas niveau qui permettent de manipuler des vecteurs de buffers de message, ainsi que des *données auxiliaires* (ou *données de contrôle*). Vous trouverez dans le `man cmsg` des exemples rigolos pour récupérer le TTL sur une socket IP, s'échanger des descripteurs de

fichiers entre processus via une socket Unix, etc.

D'autres appels système "optimisés" (i.e. selon le concept de *zero-copy*) peuvent également vous intéresser : `sendfile()` `splice()` ...

La fermeture des connexions

132/169

▶ Fonction `close()`

- ▶ Ferme le descripteur passé en argument, donc la socket si le descripteur référence une socket. (Le processus informe le noyau qu'il n'en n'a plus besoin.)
- ▶ La socket n'est vraiment fermée que lorsqu'un `close` a été fait dans tous les processus où elle est visible (processus fils par exemple)

▶ Fonction `shutdown()`

```
int shutdown(int sfd, int how);
```

▶ Paramètre `how`

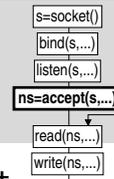
- ▶ `SHUT_RD` : fermeture en lecture
- ▶ `SHUT_WR` : fermeture en écriture
- ▶ `SHUT_RDWR` : fermeture en lecture/écriture

La manière simple de fermer une socket est d'utiliser `close()`, il faut être certain qu'il ne reste pas d'octets à recevoir.

On a plus de souplesse avec `shutdown()`, mais aussi plus de complexité. On peut fermer une socket en écriture (cela provoquera un échange protocolaire sous-jacent avec TCP par exemple (envoi d'un paquet avec le bit de contrôle **FIN**)) et continuer à pouvoir lire cette socket.

Lorsqu'une socket en mode connecté est fermée, une lecture sur la socket à l'autre extrémité de la connexion renvoie `0` indiquant ainsi la fermeture de la communication (le `read()` ou `recv()` renvoie `0`). À noter que le renvoi de `0` n'est effectué que lorsque toutes les données encore à lire ont été lues.

Comment un serveur peut gérer plusieurs connexions simultanées



- ▶ `accept()` est bloquant et c'est bien embêtant...
- ▶ `accept()` ne peut «accepter» que sur une seule socket
- ▶ Il faudrait que le serveur puisse se dupliquer après le `accept()` pour d'une part revenir sur le `accept()` et d'autre part traiter la communication
- ▶ Solutions :
 - ▶ Générer un nouveau **processus**
 - ▶ Générer un nouveau **thread**

Définition sommaire de ce qu'est un processus : *un processus est un programme en cours d'exécution dans la mémoire centrale de la machine, dans un environnement donné.*

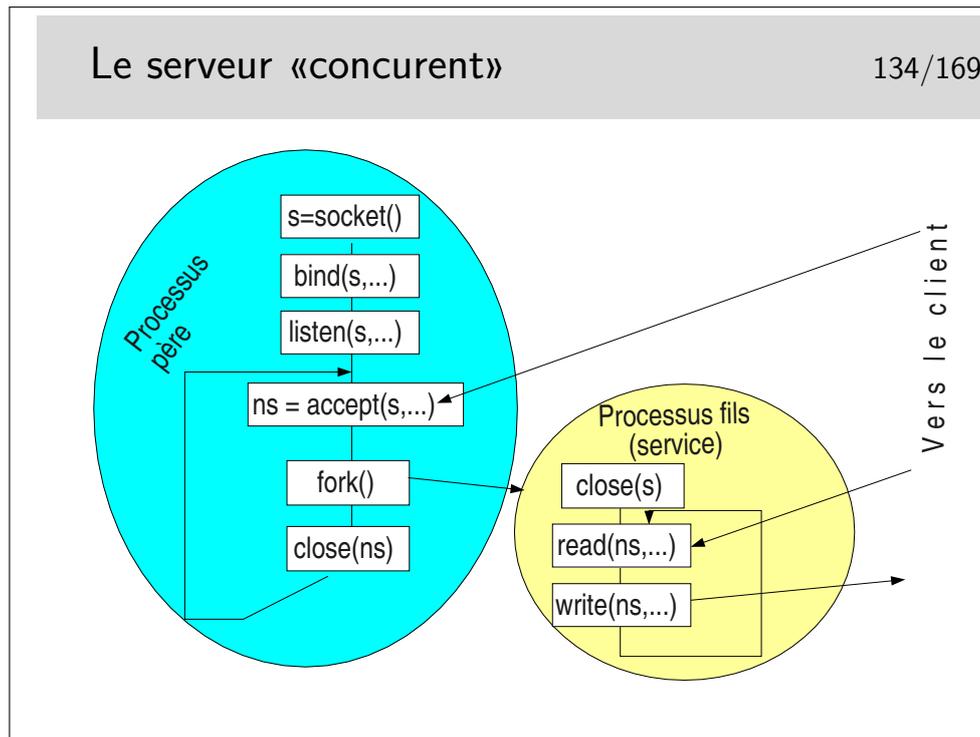
Lorsqu'on exécute un programme, celui-ci est chargé en mémoire centrale, dans un espace qui lui est propre. Dans cet espace il y a le programme lui-même ainsi que les structures de données nécessaires à l'exécution ainsi que des informations complémentaires telles que l'identité de l'utilisateur au nom duquel s'exécute le programme. Prenons l'exemple du programme qui réalise l'effacement des fichiers. Lorsque **je** l'exécute sur **mes** fichiers, il marche. Si **vous** l'exécutez sur **mes** fichiers il ne marche pas (enfin, bon, on espère) ; Pourtant il ne me connaît pas et il ne vous connaît pas. Mais lorsqu'on l'invoque, il est chargé en mémoire pour exécution en notre nom, il est muni de notre identité, ainsi il pourra exécuter les ordres d'effacement en connaissance de cause. Donc un processus, c'est le programme en mémoire plus des informations environnementales.

Un processus possède son propre espace mémoire qu'il ne partage en aucune manière avec les autres processus.

Un thread (en français, un «fil d'exécution», vous comprendrez qu'on ne cherchera plus à traduire le mot) est une sorte de processus à l'intérieur d'un processus. Il peut y avoir plusieurs threads dans un processus. Les threads partagent l'espace mémoire commun du processus. Un thread possède cependant sa propre pile.

Un thread est plus «léger» qu'un processus mais comme il y a un partage de mémoire dans un thread il y a un risque d'accès simultané en lecture ou en écriture sur une même zone mémoire, donc problèmes d'exclusion mutuelle, problèmes jamais très simples...

Sous Unix/linux, traditionnellement on travaille avec les processus. Toutefois on trouve depuis longtemps une implémentation des threads. Sous Windows il existe les deux nativement et on travaille plutôt avec les threads.



La structure du serveur est presque similaire à ce que nous avons déjà vu, à la différence près qu'on rajoute après le `accept()` la fonction de création de processus. Le processus serveur (le père) est dupliqué (on pourrait presque dire cloné) en mémoire. Il est en fait recopié dans une nouvelle zone mémoire et forme ainsi un nouveau processus, dit «processus fils». Comme il est la copie de son père, il exécute le même code. Mais il en va du clonage en informatique comme du clonage en biologie, il existe des mutations génétiques, en tous cas une en ce qui nous concerne en Unix/Linux. Il s'agit du code de retour de la fonction de création de processus : `fork()`. Dans le processus père, le `fork()` retourne une valeur différente de 0 (en fait, il s'agit du numéro de processus du fils). Dans le fils, le code retourné vaut 0. Le programmeur d'application peut ainsi dire précisément quel code sera exécuté par le père et quel code sera exécuté par le fils.

Dans notre cas, le fils exécute les fonctions de communication tandis que le père retourne bien vite bloquer sur le `accept()`.

Notez les appels à `close()` dans le père et dans le fils. Le processus père ferme la socket qu'il a reçu du `accept()` car il n'en a plus besoin. Cette fermeture dans le père n'a pas d'effet dans le fils. Ce n'est pas une fermeture totale système. La socket `ns`, reçue du `accept()` a été dupliquée dans le fils lors de sa création comme tout le code du père. Elle existe donc dans le fils et est active. La fermer dans le père est sans effet dans le fils. Mais pourquoi s'embêter avec cette fermeture ? Un processus ne peut avoir une infinité de descripteurs ouverts, la limite dépend du système d'exploitation, cela peut atteindre 1024 dans les noyaux linux récents. Dans notre cas, à chaque `accept()` on obtient un nouveau descripteur, donc à chaque nouvelle communication. Si on ne ferme pas le descripteur obtenu dans le père, on ne pourra pas servir plus de 1024 connexion (en réalité un petit peu moins). Et ceci interviendra dans... 3 jours... 3 heures... 3 semaines... Cela dépendra de la charge de notre serveur, et un blocage apparaîtra ainsi, «au bout d'un moment», de manière aléatoire... Allez déboguer cela sereinement...

Dans le fils, on ferme la socket `s` car elle ne nous sert pas, et «c'est bien» de fermer

tout de suite un descripteur devenu inutile.

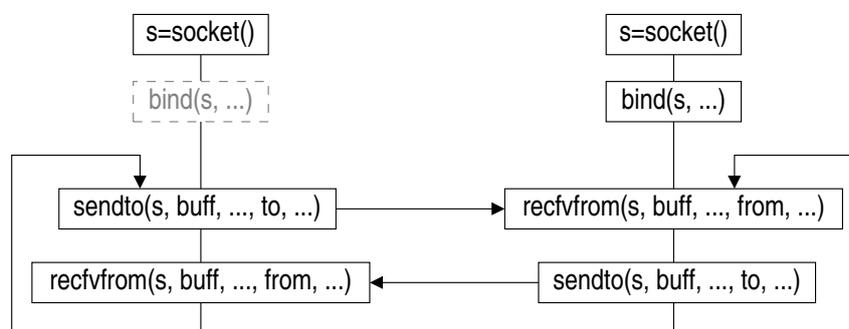
Exemple de création de processus sous Unix 135/169

```
int pid
...
pid = fork();
switch(pid) {
  case -1: /* erreur, creation impossible */
    ... traitement qui convient...
  case 0: /* nous sommes dans le fils */
    ... code du fils...
  default: /* nous sommes dans le pere */
    ... code du pere...
}
```

L'appel système `fork()` peut échouer si le nombre maximal de processus pour le système est atteint (certains systèmes ne permettent pas de dépasser un certain nombre de processus). L'appel peut aussi échouer s'il n'y a plus assez de mémoire centrale disponible (y compris la mémoire d'échange, le swap sous Unix). Dans tous ces cas, le `fork()` rend `-1` et il faut traiter ce cas d'erreur en fonction du contexte (c'est grave, alors on sort par un `exit()`, c'est moins grave alors on sort du switch et on continue)

Les sockets en mode non connecté

136/169



Il n'y a pas de connexion. On ouvre les sockets de part et d'autre, d'un coté, obligatoirement, on affecte une adresse avec `bind()` (ce sera le «serveur» en quelque sorte), on est alors prêt à recevoir de ce coté et à envoyer de l'autre coté. Le coté qui envoie en premier n'est pas obligé de faire un `bind()`, le `sendto()` affectera l'adresse qui convient (le numéro de port en UDP par exemple) si cela n'a pas été fait auparavant.

Le `sendto()` envoie à l'adresse `to`.

Le `recvfrom()` mémorise l'adresse de la socket émettrice dans le paramètre `from`.

À noter qu'il est possible d'utiliser `connect()` coté «client», comme en mode connecté. Ce ne sera pas une vraie connexion, en UDP par exemple le protocole ne prévoit rien de ce genre. Donc le `connect()` sera «virtuel» mais il aura mémorisé un contexte contenant l'adresse de la socket distante et ce contexte sera associé à la socket. Il sera alors possible d'utiliser `read()/write()` ou `recv()/send()` pour réaliser l'échange de données.

sendto() et recvfrom()

137/169

```
ssize_t sendto(int sfd,
               const void *buf,
               size_t len,
               int flags,
               const struct sockaddr *to,
               socklen_t tolen);

ssize_t recvfrom(int sfd,
                void *buf,
                size_t len,
                int flags,
                struct sockaddr *from,
                socklen_t *fromlen);
```

Les trois premiers arguments de ces fonctions sont semblables à ceux de `write()` et `read()` : la socket le tampon mémoire contenant les octets à envoyer (`sendto()`) ou à recevoir (`recvfrom()`), la longueur max d'octets à envoyer ou à lire.

Le flag en quatrième argument est très peu utilisé et sera toujours à 0.

L'argument `to` pour `sendto()` pointe sur une structure d'adresse contenant l'adresse de la socket distante à laquelle on veut envoyer les données.

L'argument `from` pour `recvfrom()` pointe sur une structure d'adresse qui contient l'adresse de la socket qui nous a envoyé les données (dont on peut prendre connaissance au retour de `recvfrom()`).

- ▶ Obtenir l'adresse ou le nom d'une machine
 - ▶ `getaddrinfo()`
 - ▶ Permet de récupérer une liste chaînée de structures d'information contenant, en particulier, les structures d'adresses de la machine dont on a passé le nom ou l'adresse sous la forme de chaîne de caractères (même notée a.b.c.d)
 - ▶ `getnameinfo()`
 - ▶ Renvoie sous forme de chaîne de caractères le nom et le numéro de port (ou service) associés à une structure d'adresse.

La fonction `getnameinfo()` permet, par exemple, facilement de récupérer l'identité de la socket distante en lui passant en argument la structure `from`, argument de `accept()` ou de `recvfrom()`.

Note : historiquement on utilisait les fonctions `gethostbyname()` et `gethostbyaddr()` pour cela (ainsi que `gethostbyname2()`). Ces fonctions sont maintenant obsolètes (dédiées à IPv4) et ne doivent plus être utilisées dans de nouveaux programmes.

La fonctions `getaddrinfo()`

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

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

void freeaddrinfo(struct addrinfo *res);

const char *gai_strerror(int errcode);
```

La fonction `getaddrinfo()` permet de faire la résolution de noms. On lui fournit un nom de *node* (i.e. un nom de machine) et/ou un nom de *service* (i.e. un port), et la fonction fait ce qu'il faut pour nous retourner la liste de structure d'adresses correspondantes. (Consulte les fichiers `/etc/hosts` `/etc/services`, le système NIS, ou le DNS.)

Notez qu'il est normal pour un nom de machine d'avoir plusieurs adresses, tant IPv4 qu'IPv6. C'est pourquoi cette fonction retourne une liste chaînée `res`, qu'il faudra prendre soin de vider de la mémoire après usage avec un `freeaddrinfo()`.

En cas de succès, la fonction retourne 0. En cas d'erreur, la fonction retourne un code d'erreur que l'on peut afficher à l'aide de `gai_strerror()`.

La fonction `getaddrinfo()` attend en paramètre une structure d'adresse `hints`, qui est en quelque sorte le *patron* des structures d'adresse que l'on souhaite en retour : on précise si l'on veut des adresse IPv4 ou bien IPv6, ou bien les deux. On peut préciser le protocole utilisé par la suite par la socket (`SOCK_STREAM` `SOCK_DGRAM` ou 0). On peut préciser que l'on veut un adresse *joker*, dans le cas où l'on va utiliser cette socket pour faire une socket serveur. On peut demander des comportement particuliers : par exemple utiliser des adresses IPv4 mappées en IPv6, etc.

Bref, au fil du temps, cette fonction est devenue centrale dans les programmes utilisant des sockets réseau. Si l'on se débrouille bien, il devient facile de coder des programmes qui fonctionnent indifféremment en IPv4 et en IPv6. Le `man getaddrinfo()` fournit des exemples de code C, qu'il suffit de copier et d'adapter. (Les personnes attentives noteront dans l'exemple en question l'utilisation d'un connect sur une socket UDP.)

La démarche générale est la suivante : puisque cette fonction nous retourne une liste de structures d'adresses potentiellement intéressantes, on parcourt cette liste en essayant chaque adresse pour notre socket, en faisant un appel système à `socket()` puis à `bind()` (dans le cas d'un serveur) ou à `connect()` (dans le cas d'un client).

La fonction `getnameinfo()`

140/169

```
#include <sys/socket.h>
#include <netdb.h>

int getnameinfo(const struct sockaddr *sa, socklen_t s
                char *host, size_t hostlen,
                char *serv, size_t servlen, int flags)

/* A titre indicatif: */
#define NI_MAXHOST    1025
#define NI_MAXSERV    32

char host[NI_MAXHOST], serv[NI_MAXSERV];
```

La fonction `getnameinfo()` retourne sous forme de chaîne de caractères le nom de

machine et le service (port) de la structure d'adresse passée en paramètre (typiquement après un `accept()` ou un `recvfrom()`).

Le code d'erreur éventuel peut être affiché avec un `gai_strerror()`.

La services pour résolution d'adresse

141/169

- ▶ Les fonctions vues précédemment utilisent les services de résolution d'adresse du système d'exploitation (enfin, dans la libc).
 - ▶ Exemple sous Unix/linux, sous contrôle des fichiers `/etc/nsswitch.conf` `/etc/gai.conf`
 - ▶ Le fichier `/etc/hosts`
 - ▶ Le service NIS
 - ▶ Le DNS
 - ▶ Ce n'est pas au programmeur de décider quel service il va prendre, c'est le rôle de l'administrateur du système sur lequel le programme va s'exécuter.

On simplifie ainsi grandement les choses. Les fonctions `getxyzinfo()` font le travail pour vous, de manière transparente. Une ligne de code (plus le test de réussite) et le tour est joué, même si en fait il faut consulter des services complexes comme le DNS (donc en fait quelques milliers de lignes de codes mises en jeu, de votre côté comme du côté des serveurs, un réseau d'envergure mondiale consulté, etc. Le tout en une seule ligne de code pour vous... magique!)

Les fichiers `/etc/gai.conf` `/etc/nsswitch.conf` (ou `/etc/host.conf`) permet d'indiquer l'ordre dans lequel les sources d'informations seront consultées.

Sous Windows, les mêmes fonctions donnent les mêmes résultats.

- ▶ Obtenir des informations sur la socket locale ou la socket distante

- ▶ Fonction `getsockname()`

```
int getsockname(int sfd,
                struct sockaddr *name,
                socklen_t *namelen);
```

- ▶ Fonction `getpeername()`

```
int getpeername(int sfd,
                struct sockaddr *name,
                socklen_t *namelen);
```

Ces fonctions retournent, à l'adresse pointée par leur paramètre `name`, l'identité de la socket locale pour la première et l'identité de la socket distante pour la seconde. En IP on pourra ainsi récupérer la structure d'adresse `struct sockaddr_in` de la socket locale ou distante.

La fonction `getsockname()` peut, par exemple, être utilisée pour récupérer le numéro de port affecté à une connexion coté client, alors qu'on n'a pas fait de `bind()`. Mais on peut imaginer d'autres utilisations.

Etes vous *big endian* ou *little endian* ?

- ▶ Ou le problème de la représentation des nombres en machine

- ▶ Prenons un exemple, l'entier 2048 (2^{11}), il s'écrit 0800 en hexadécimal (en notation C on écrirait `0x0800`)

- ▶ Il se range en mémoire comme ceci :

	08	00	
Adresse mémoire	n	n+1	Big Endian (Motorola (PowerPC) Sun (Sparc))

- ▶ Ou... Comme ceci :

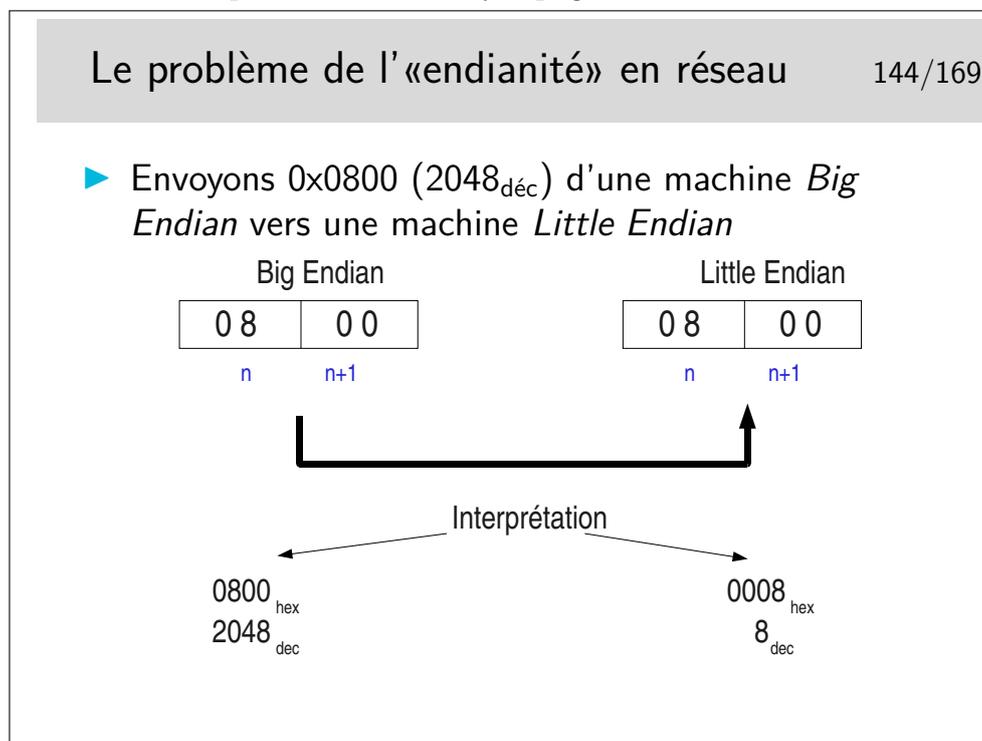
	00	08	
Adresse mémoire	n	n+1	Little Endian (Intel, ...)

Il est tout à fait naturel de ranger en mémoire les octets dans le sens où ils sont écrits sur le papier. Et lorsqu'on range des octets on commence par l'adresse mémoire la plus faible, donc il est tout à fait naturel de placer le poids fort à l'adresse de poids faible et l'octet de poids faible à suivre. Les gens *Big Endian* pensent comme cela (Motorola et IBM avec l'architecture PowerPC et donc Apple, Sun avec SPARC et bien d'autres)

Il est tout aussi naturel de penser que le poids faible du nombre tombe tout naturellement dans le poids faible de la mémoire. Les gens *Little Endian* pensent comme cela (Intel avec l'architecture générique x86 et bien d'autres).

Notez bien que ce problème est dû à l'architecture matérielle et non logicielle. Ainsi, le système d'exploitation unix Solaris de Sun fonctionne sur des machines Big Endian (Sun Sparc) et Little endian (PC type x86 tel que les pentium), donc le même système d'exploitation avec les mêmes outils et bibliothèques de programmation peut fonctionner dans un environnement ou dans un autre et poser des problèmes d'interopérabilité en communication.

Car l'endian a un impact en réseau. Voyez page suivante.



Voyez vous le problème ?

Lorsqu'on envoie des octets dans le réseau, on commence l'envoi par l'octet présent à l'adresse de poids faible de la mémoire et on progresse vers les poids forts. Ainsi dans notre exemple, on enverra 08 d'abord et ensuite 00.

Lorsqu'on reçoit, on le fait évidemment dans l'ordre où les données sont émises et on les range en mémoire en commençant par l'octet de poids faible de la mémoire. On recevra donc ici 08 puis 00 et on rangera dans cet ordre.

Tout va bien... Où est le problème ? Il est dans l'interprétation que va faire la machine réceptrice. C'est une little endian, elle va prendre 00 comme poids fort, elle va donc lire 0008 soit 8 en décimal alors que la machine émission a envoyé 2048 en décimal.

Dans le modèle OSI, le problème évoqué ici est adressé dans la couche 6, la couche

«présentation». L'OSI définit un langage permettant de typer l'information et un mécanisme permettant de coder les différents types définis, il s'agit du langage ASN.1 (*Abstract Syntax Notation One*) et du mécanisme de codage BER (*Basic Encoding Rules*). Le «modèle Internet» n'intègre pas de solutions pour ce problème. Les développeurs doivent penser à le gérer eux mêmes. Des solutions se voulant standard ont été développées, par exemple la couche XDR (External Data Representation), développée par Sun pour ses Remote Procedure Call sous Unix. RPC/XDR est effectivement devenu un standard, pour Unix, les services NFS et NIS ont été développés avec. Mais ce «standard» n'a pas été suivi par tout le monde et finalement les autres (tel Windows) ont leurs propres solutions.

Des informations plus complète existent sur le réseau : le mot clé «Endian» (attention, pas «*indian*») vous guidera vers de véritables cours sur le sujet.

Les fonctions auxiliaires	III	145/169
<ul style="list-style-type: none">▶ Les conversions Machine/Réseau<ul style="list-style-type: none">▶ Le réseau est «Big Endian»▶ Les fonctions de conversion :<ul style="list-style-type: none">▶ Host to network : <code>htons()</code>, <code>htonl()</code>▶ Netwok to host : <code>ntohs()</code>, <code>ntohl()</code>▶ Exemple : <code>sin.sin_port = htons(1234);</code>▶ Les fonctions <code>getaddrinfo()</code> et <code>getnameinfo()</code> gèrent cela elles-mêmes (une autre bonne raison de les utiliser)		

Le «s» ou le «l» indiquent une conversion *short* soit 16 bits ou *long* soit 32 bits. Pour les ports TCP ou UDP on utilisera par exemple `htons()` comme dans l'exemple ci-dessus.

Les paramètres et valeurs rendues sont tous de type *unsigned*

Contournement du problème des blocages dus aux fonctions

147/169

- ▶ Rendre les sockets non bloquantes
 - ▶ Avec l'appel système `fcntl()`
 - ▶ Dangereux si on fait ensuite des lectures en boucle (surcharge CPU)
- ▶ Utiliser la fonction `select()`
- ▶ Utiliser les mécanismes asynchrones

Quand un descripteur de fichier (une socket est un descripteur de fichier) est positionné en mode non bloquant le `read()` rend `0` immédiatement (et la variable externe `errno` est positionnée à la valeur `EAGAIN`) s'il n'y a rien à lire. Si la lecture est dans une boucle de programme et que l'on revient trop rapidement sur elle alors on sollicite très souvent le noyau du système d'exploitation. La charge CPU monte au détriment des autres applications sur la machine.

Exemple de positionnement de socket en mode non bloquant :

```
#define BUFSIZE 2048
extern int errno;
int s, flags;
char buf[BUFSIZE];
...
s = socket(...);
...
flags = fcntl(s, F_GETFL);
r = fcntl(s, F_SETFL, flags | O_NONBLOCK);
if(r < 0) {
    perror("Erreur_fcntl");
    ...
}
...
r = read(s, buf, BUFSIZE);
if ( r == 0 && errno == EAGAIN) {
    /* rien a lire */
    ...
}
```

```
int select(int n,          /* nb descripteurs +1 */
           fd_set *readfds, /* masque lecture */
           fd_set *writefds, /* masque ecriture */
           fd_set *exceptfds, /* masque evt_execp */
           struct timeval *timeout);
```

- ▶ Permet d'attendre l'arrivée d'une lecture, une écriture, un événement exceptionnel sur $n - 1$ descripteurs
- ▶ Permet de paramétrer le temps d'attente
- ▶ Les descripteurs sur lesquels on attend les événements sont indiqués dans des masques
- ▶ Des macros sont disponibles pour préparer les masques

Les masques sont en fait des nombres entiers non signés dans lesquels les rangs des bits à «1» sont significatifs du descripteur : bit 0 : descripteur 0, bit 1 : descripteur 1, etc.

`select()` bloque tant qu'un événement n'arrive pas sur un des descripteurs, par exemple si on n'a que le descripteur 1 dans le masque `readfds`, alors `select()` bloquera tant qu'il n'y aura pas d'octets à lire sur le fichier correspondant au descripteur 1. La durée du blocage peut être paramétrée par le paramètre `timeout`. Si celui-ci est à `NULL` alors le blocage est permanent jusqu'à un des événements attendus sur les descripteurs.

Si on a les descripteurs 1 et 4 dans le masque `readfds` et que des octets sont à lire sur le descripteur 4 alors `select()` débloque, mais le masque `readfds` est modifié, il ne reste plus dedans que le descripteur 4, le 0 est effacé. Il faut tester le masque pour voir quel sont les descripteurs pour lesquels une action est à entreprendre.

Il est possible d'utiliser `select()` pour gérer plusieurs sockets en mode serveur (état `LISTEN` si on est en `TCP`) et attendre ainsi des requêtes de connexion, grâce au masque pour la lecture (`readfds` ci-dessus).

Le masque `exceptfds` contient des descripteurs sur lesquels on attend des événements exceptionnels. En pratique, les seuls événements exceptionnels gérables sont les arrivées de données urgentes `TCP` (nous verrons plus loin une autre méthode pour gérer la réception de ce type de données).

La structure de type `timeval` contient deux champs `tv_sec` et `tv_usec` pour indiquer des secondes et des microsecondes.

Variantes : l'appel système `pselect()` propose en plus une gestion des masques de signaux qui pourraient survenir pendant l'attente.

Les appels systèmes `poll()` et `ppoll()` font à peu près la même chose, mais avec une api un peu différente...

La fonction select() II

Les macros

149/169

- ▶ `FD_CLR(int fd, fd_set *set);`
 - ▶ Enlève le descripteur `fd` du masque `set`
- ▶ `FD_ISSET(int fd, fd_set *set);`
 - ▶ Teste si le descripteur `fd` est dans le masque `set` (utile au retour de `select()` pour voir quels descripteurs ont des événements en attente)
- ▶ `FD_SET(int fd, fd_set *set);`
 - ▶ Place le descripteur `fd` dans le masque `set`
- ▶ `FD_ZERO(fd_set *set);`
 - ▶ Nettoie le masque `set`

La fonction select() III

Exemple d'utilisation

150/169

```
1 int fd1, fd2, max, r;
2 ...
3 fd_set r_msq, tr_msq
4 ...
5 FD_ZERO(&r_msq);
6 FD_SET(fd1, &r_msq); FD_SET(fd2, &r_msq);
7 max = fd1 > fd2 ? fd1 : fd2;
8 for (;;) {
9     tr_msq = r_msq;
10    select(max+1, &tr_msq, 0, 0, 0);
11    if (FD_ISSET(fd1, &tr_msq) {
12        r = read(fd1, ...
13    }
14    if (FD_ISSET(fd2, &tr_msq) {
15        r = read(fd2, ...
16    }
17 }
```

Ligne 1 : on déclare les descripteurs et diverses variables utiles...

Ligne 3 : on déclare deux masques, un de référence (`r_msq`) et un de travail (`tr_msq`)

Ligne 5 : On nettoie le masque de référence

Ligne 6 : On place les deux descripteurs dans le masque de référence. Il se peut que les descripteurs soient des sockets ou d'autres canaux de communication comme des tubes. L'exemple est valable quelque soient les descripteurs...

Ligne 7 : on détermine quel est le descripteur le plus grand et on mémorise sa valeur

dans max.

Ligne 8 : on entre dans une boucle sans fin

Ligne 9 : on copie le masque de référence dans le masque de travail

Ligne 10 : appel à `select()` (enfin). Le masque de travail est passé dans l'argument indiquant que l'on attend des événements en lecture.

Au retour du `select()` le masque est modifié, il ne reste dedans que les descripteurs pour lesquels l'événement attendu a eu lieu.

Ligne 11 : on teste si `fd1` est dans le masque

Ligne 12 : si oui, alors on le lit

Lignes 14 et 15 : idem pour `fd2`.

Notez que le masque est modifié. Donc lorsque l'on retourne en début de boucle, on le réinitialise en recopiant dedans le masque de référence (ligne 9).

En toute rigueur il faudrait tester le code de retour de `select()` pour voir s'il vaut -1 auquel cas il aurait une erreur.

Les lectures asynchrones

I

151/169

- ▶ Le processus applicatif fait un certain travail, mais pas de lecture pour ne pas rester bloqué
- ▶ Si des données de communications arrivent il reçoit un signal, sorte d'interruption logicielle
- ▶ Il se dérouté vers une routine de traitement spécifique dans laquelle il fait la lecture
- ▶ Le processus doit prévoir être dérouté, il doit demander au noyau que celui-ci lui envoie le signal

Les lectures asynchrones II

Exemple type

152/169

```
1 void gestionnaire() {
2     int r;
3
4     r = read(sock, buf, BUFSIZE);
5 }
6
7 int main() {
8     ...
9     fcntl(sock, F_SETOWN, getpid());
10    signal(SIGIO, gestionnaire);
11    ... travail ...
12    .....
```

Ligne 1 à 5 : la fonction `gestionnaire` qui sera appelée lors de la réception du signal. On y fait la lecture de la socket.

Ligne 10 : on indique au noyau du système d'exploitation que l'on désire recevoir le signal `SIGIO` (oui, c'est certainement peu évident.... mais...). La fonction `getpid()` renvoie le numéro du processus courant..

Ligne 11. On indique au processus quelle sera la fonction à appeler lors de la réception du signal `SIGIO`.

La variable `sock` doit être une variable globale (tout bon professeur d'informatique vous dira que c'est mal d'utiliser des variables globales...)

► Envoie

```
send(sock, buf, n, MSG_OOB)
```

► Réception

```
1 void gestionnaire() {
2     int r;
3     r = recv(sock, buf, BUFSIZE, MSG_OOB);
4 }
5
6 int main() {
7     ...
8     fcntl(sock, F_SETOWN, getpid());
9     signal(SIGURG, gestionnaire);
10    ... travail ...
11    .....
```

Comme pour les données asynchrones... On ne s'attend pas à recevoir des données urgentes à un instant bien précis, alors on se prépare à les recevoir n'importe quand (voire jamais). On dit au noyau que l'on veut recevoir le signal **SIGURG** (comme pour **SIGIO**) avec **fcntl()**. On indique quelle fonction sera appelée lors de la réception du signal. Lorsque la couche TCP (dans le noyau) recevra une donnée urgente, le noyau enverra le signal au processus qui se déroutera vers la fonction gestionnaire dans laquelle on fera la lecture, via **recv()** muni du *flag* idoine.

Note : À l'usage on se rend compte qu'au final il n'y a qu'un seul octet (le dernier du message hors bande) qui est considéré comme urgent par le récepteur. En effet, TCP définit un mécanisme (pointeur de donnée urgente) qui indique *la fin* du flux d'octets urgents, mais c'est à l'application de se débrouiller pour définir le début (mais comment?)...

Le RFC 793 (qui décrit TCP) comporte quelques ambiguïtés sur le sujet. Le RFC 6093 "*On the Implementation of the TCP Urgent Mechanism*", janvier 2011, rectifie le tir, mais le mal était déjà fait...

C'est ainsi que le pluspart des applications existantes qui utilisent ce mécanisme se contentent de n'envoyer qu'un seul octet urgent (car c'est pris en charge entièrement par la couche transport).

Et le RFC 6093 conclut par «*new applications SHOULD NOT employ the TCP urgent mechanism*»...

```
int getsockopt(int sfd, int level, int optname,  
              void *optval, socklen_t * optlen);  
int setsockopt(int sfd, int level, int optname,  
              const void *optval, socklen_t optlen);
```

- ▶ `s` : le descripteur de la socket
- ▶ `level` : indique la portée de l'opération
 - ▶ Valeurs : `SOL_SOCKET`, `SOL_IP`, `SOL_TCP`...
- ▶ `optname` : le nom de l'option
- ▶ `optval` : la valeur de l'option
- ▶ `optlen` : la longueur de l'option

Le niveau `SOL_SOCKET` permet de paramétrer des options propres à la socket mais parfois celles-ci ont malgré tout une action sur le protocole sous-jacent. C'est par exemple le cas de l'option `SO_KEEPALIVE`, `SO_OOBINLINE`, etc.

Les options propres aux protocoles sont décrites dans les pages du manuel telles que `tcp(7)`, `unix(7)`, `socket(7)`.

Remarque : la notation `tcp(7)` indique que la page du manuel ainsi référencée est située dans la section 7 du manuel de référence Linux. On peut l'appeler par la commande `man` de la manière suivante : `man 7 tcp`

- ▶ `SO_BROADCAST` : permet la fonction diffusion générale sur la socket (en UDP)
- ▶ `SO_REUSEADDR` : permet de réutiliser une adresse déjà affectée par `bind()`.
- ▶ `SO_KEEPALIVE` : provoque un envoi de message de test de présence pour les communications en mode connecté qui sont silencieuses pendant un certain temps
- ▶ `SO_RCVBUF`, `SO_SNDBUF` : taille des tampons de réception et d'émission
- ▶ `SO_LINGER` : contrôle l'envoi des données au moment de la fermeture de la connexion
- ▶ Voir man 7 socket sous linux pour compléments

`SO_REUSEADDR` : Deux sockets ne peuvent avoir le même *nom* (sans quoi le système d'exploitation ne sait pas à quel processus confier les données qui arrivent). Ainsi, si un processus serveur a ouvert une socket sur un port TCP, aucun autre processus de la machine ne peut ouvrir une socket sur ce port. Lorsque le processus serveur se termine, le système d'exploitation va placer sa socket TCP dans l'état `TIME_WAIT` en attendant que les connections TCP se clôturent proprement (le RFC préconise 4 minutes). Cependant, "on sait bien que" le port n'est plus vraiment utile pendant ce temps là (en général). Un nouveau processus pourrait le réutiliser immédiatement. C'est possible en le demandant gentiment au système d'exploitation avec l'option de socket `SO_REUSEADDR`.

Autre cas de figure : des récepteurs multicast UDP. Il est normal que plusieurs processus puissent écouter le même canal multicast, et donc d'ouvrir plusieurs sockets UDP sur le même *nom* (adresse IP du flux multicast + numéro de port). Il faut donc le demander gentiment au système d'exploitation avec l'option de socket `SO_REUSEADDR`.

`SO_KEEPALIVE` : considérons une connexion telnet (donc TCP), on peut très bien rester connecté des heures ou des jours sans demander d'échange de données. Dans ce type de connexion silencieuse TCP, il n'y a vraiment pas d'échange, même protocolaire. Si une des applications en communication s'arrête (machine en panne par exemple), l'autre machine ne s'apercevra de rien. Pour cela, un protocole comme TCP prévoit d'envoyer régulièrement des paquets de test (segments vides de données utiles). TCP prévoit de le faire toutes les deux heures si la communication est silencieuse et si l'option en question est positionnée.

Un paquet de test appelle un acquittement, si ce dernier ne vient pas, alors la couche TCP avertit l'application. Dans notre cas il suffit que l'application soit en lecture, le `read()` ou `recv()` revient en rendant `-1`.

`SO_LINGER` : cette option demande l'argument suivant :

```
struct linger {
```

```

int l_onoff; /* linger active */
int l_linger; /* how many seconds to linger for */
};

```

Si `SO_LINGER` est actif, un `close()` ou `shutdown()` sur la socket bloquera tant que les données restant à envoyer (encore présentes dans le tampon d'émission) n'auront pas été correctement envoyées ou que la durée d'attente spécifiée n'aura pas été atteinte. Lors d'une fermeture par `exit()` (terminaison d'application), l'attente se produit en arrière plan, dans la couche protocolaire.

Les sockets sous Windows - win32

156/169

- ▶ Ce que nous venons de voir s'applique aussi sous Windows en environnement win32 avec les exceptions suivantes :
 - ▶ La socket n'est pas un descripteur de fichier mais un type `SOCKET`
 - ▶ Il faut utiliser `send()` et `recv()` à lieu de `write()` et `read()`
 - ▶ La création de processus est réalisée différemment et on préfère utiliser des *threads*
 - ▶ Il existe plus de fonctions que ce que nous avons vu (par exemple il existe `socket()` mais aussi `WSASocket()` plus riche)
 - ▶ Il faut initialiser la dll `winsock2`

Voir en ligne http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/winsock_reference.asp

Exemple de code http://msdn.microsoft.com/library/default.asp?url=/library/en-us/winsock/winsock/complete_client_code.asp

Code d'initialisation de la dll `winsock2` :

```

// Initialize Winsock.
WSADATA wsaData;

int iResult = WSASStartup( MAKEWORD(2,2),&wsaData );
if ( iResult != NO_ERROR ) {
    printf("Error_at_WSASStartup()\n");
    return;
}

```

- ▶ Classes spécifiques intrinsèques au langage
Naturellement adaptées à TCP/UDP-IP
 - ▶ Socket : pour les clients
 - ▶ ServeurSocket : comme son nom l'indique
 - ▶ Une seule ligne de code pour ouvrir et connecter la socket
 - ▶ Des raffinements sont possibles
 - ▶ Nécessité d'associer des flux de lecture et d'écriture aux sockets ainsi ouvertes
 - ▶ DatagramSocket avec UDP
 - ▶ Classe complémentaire : DatagramPacket

Documentation de référence sur l'ensemble des classes du langage : <http://java.sun.com/j2se/1.4.2/docs/api/index.html>

Cette documentation peut être téléchargée pour consultation locale via un browser web.

```
1 import java.io.*;
2 import java.net.*;
3 ...
4 Socket mySocket = null;
5 PrintWriter out = null;
6 BufferedReader in = null;
7 try {
8     mySocket = new Socket("serveur", 7890);
9     out = new PrintWriter(mySocket.getOutputStream(), true);
10    in = new BufferedReader(
11        new InputStreamReader(mySocket.getInputStream()));
12 } catch (UnknownHostException e) {
13     System.err.println("machine_␣serveur_␣inconnue");
14     System.exit(1);
15 } catch (IOException e) {
16     System.err.println("Communication_␣impossible_␣avec_␣serveur");
17     System.exit(1);
18 }
```

Ligne 8 : création de la socket, connexion au serveur indiqué en premier paramètre sur le port précisé en second paramètre. Magique, non? (TCP implicite, cela va de soi).

Ligne 9 et 10 : accroche d'un flux d'écriture (**out**) et d'un flux de lecture (**in**) à la

socket connectée. C'est le seul ennui du mécanisme, les sockets créées ne sont accessibles en lecture et écriture que via des `InputStream` et des `OutputStream`.

Exemple extrait de <http://java.sun.com/docs/books/tutorial/networking/sockets/readingWriting.html>

Sockets et java : exemple serveur

159/169

```
1 ServerSocket serverSocket = null;  
2 try {  
3   serverSocket = new ServerSocket(4444);  
4 } catch(IOException e) {  
5   System.err.println("Could not listen on port: 4444.");  
6   System.exit(1);  
7 }  
8  
9 Socket clientSocket = null;  
10 try {  
11   clientSocket = serverSocket.accept();  
12 } catch(IOException e) {  
13   System.err.println("Accept failed.");  
14   System.exit(1);  
15 }  
16  
17 PrintWriter out = new PrintWriter(clientSocket.getOutputStream()),  
18   true);  
19 BufferedReader in = new BufferedReader(  
20   new InputStreamReader(clientSocket.getInputStream()));
```

Facile, ici aussi.

Ligne 3 : ouverture de la socket et association au port indiqué en paramètre. Le `bind` et le `listen` sont implicites dans cet exemple (il faut savoir que la méthode `bind()` existe cependant).

Ligne 11 : acceptation de connexions. On voit que la méthode `accept()` rend une nouvelle socket comme pour l'API en C.

Lignes 17 et 18 : association de flux d'écriture et de lecture à la socket obtenue par le `accept()`.

Exemple tiré de : <http://java.sun.com/docs/books/tutorial/networking/sockets/example-1dot1/KnockKnockServer.java>

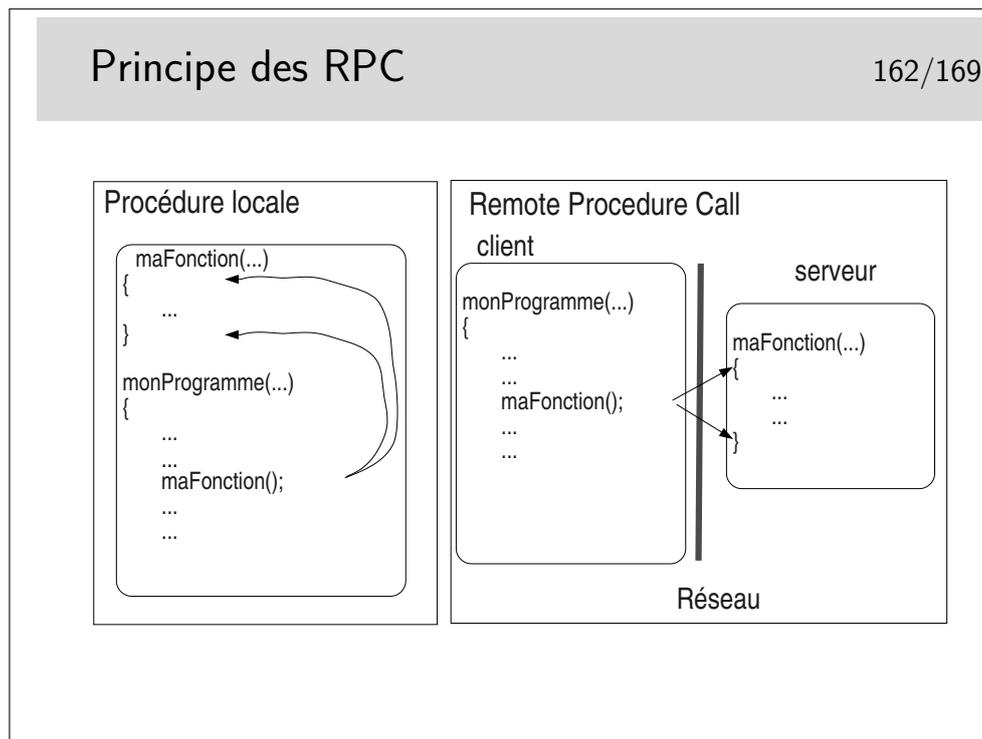
- ▶ `PF_INET6` `AF_INET6` (à la place de `PF_INET` `AF_INET`)
- ▶ `sockaddr_in6` plus grand qu'un `sockaddr` :
 - ▶ pour socket utiliser le type générique `sockaddr_storage` plutôt que `sockaddr`,
 - ▶ ne change rien pour le passage par pointeur (arguments des appels systèmes)
- ▶ `getaddrinfo()` et `getnameinfo()` (à la place de `gethostbyname()` `gethostbyaddr()`)
- ▶ Problématique *double pile IP* (prévoir deux sockets IPv4 + IPv6) vs. *IPv4 mappé* (l'os permet une connexion IPv4 sur socket IPv6)

L'API Socket a été enrichie pour utiliser l'IPv6 (RFC 3493). Tout a été fait que ce soit transparent pour le programmeur... tout du moins au niveau des sockets. Les soucis viennent des applications qui utilisaient les adresses IPv4 à des choses exotiques, comme identifiants internes par exemple... Comme on passe de 32 bits à 128, ça peut changer beaucoup de choses dans les structures de données.

Quand à la question d'un système *double pile* vs. *IPv4 mappé*, la responsabilité se partage entre d'une part l'administrateur qui configure son OS pour adopter l'une ou l'autre des stratégies (`sysctl net.ipv6.bindv6only` sous Linux), et d'autre part le développeur d'application qui doit savoir se débrouiller avec l'une, l'autre ou les deux stratégies... (Voir la célèbre option de java `-Djava.net.preferIPv4Stack=true...`) Notez que la stratégie *double pile* semble prédominer dans les OS d'aujourd'hui.

Une très bonne référence sur le domaine : http://livre.g6.asso.fr/index.php/Programmation_d%27applications

8.3 L'API RPC



Dans le cas classique, une procédure (en C, une fonction) est appelée en local par le processus en cours. La fonction est interne au processus (interne au programme complet, compilé et l'édition de lien réalisée).

Dans le cas des RPC, le corps de la fonction est externe au programme principal (le client). Il existe cependant une instance locale de la fonction dont le rôle est d'appeler la vraie fonction située dans un processus distinct et même distant (le serveur).

Les aspects «Réseaux» doivent être masqués au programmeur (vœux pieux car si les aspects réseaux sont bien masqués, les mécanismes mis en jeu sont complexes et cette complexité ne peut être totalement rendue transparente).

- ▶ Origine Sun Microsystems (1984)
 - ▶ Service NFS et NIS basés sur ces mécanismes
- ▶ Modèle en deux couches
 - ▶ La couche RPC (l'équivalent de la couche Session OSI)
 - ▶ Nombreuses fonctions : voir man rpc
 - ▶ La couche XDR : *eXternal Data Representation* (équivalent à la couche Présentation OSI)
 - ▶ XDR fournit un ensemble de fonctions d'encodage et de décodage en ligne ainsi que d'adaptation à la représentation locale des données en machine (problème *Big/Little Endian*)
 - ▶ Nombreuses fonctions : voir man xdr

Il est possible d'utiliser la couche xdr directement au dessus des sockets si on a besoin de transmettre des données structurées contenant des types tels que des entiers ou des réels, afin de ne pas être gêné par les problèmes «d'endianité»...

- ▶ Les procédures internes à un serveur RPC sont assimilées à des «services»
 - ▶ Un service est identifié par un numéro de service et de version
 - ▶ Un ensemble de procédures est identifié par un numéro de «programme»
 - ▶ Voir : /etc/rpc

RPC sous Unix/Linux : localisation du serveur et des services

II 165/169

- ▶ Un serveur est associé à un port TCP ou UDP
 - ▶ Le serveur s'attribue ce port à son lancement
 - ▶ Il en informe un service central sur la machine : le portmapper
 - ▶ Il communique au portmapper la liste de ses services et son numéro de port
- ▶ Un client désire utiliser une procédure
 - ▶ Il demande à la machine serveur (processus portmapper, port 111) le numéro de port du service correspondant à la procédure
 - ▶ Le portmapper donne l'information
 - ▶ Le client peut contacter directement le serveur

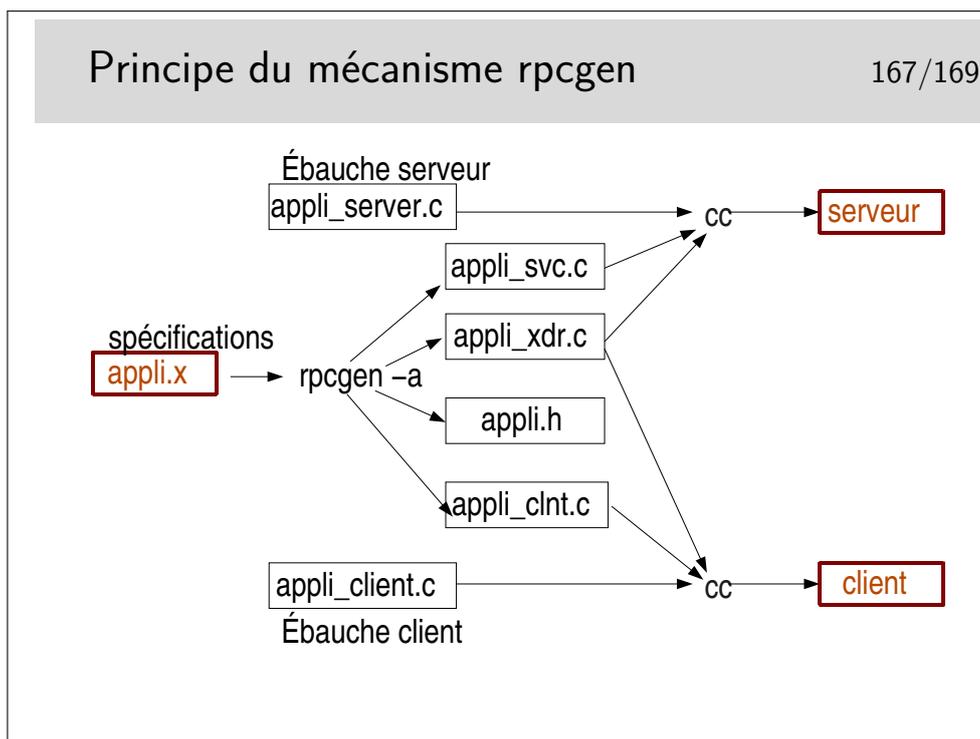
Développement d'applications RPC

166/169

- ▶ «à la main»
 - ▶ En utilisant directement les fonctions `rpc` et `xdr`
 - ▶ Complexe
- ▶ À l'aide de l'outil `rpcgen`
 - ▶ Compilateur de fichiers de spécifications
 - ▶ Fournit des fichiers en C contenant les routines `rpc/xdr` nécessaires (codes talons)
 - ▶ Peut fournir des ébauches des clients et des procédures distantes (code serveur)

Faire `man rpc` et `man xdr` pour voir toutes les fonctions disponibles pour travailler «à la main» et ruez vous sur `rpcgen` pour être plus tranquilles (?).

Retenez toutefois que si vous voulez faire des choses sophistiquées vous devrez en passer par ces fonctions..



Un fichier **Makefile** est aussi produit, on peut donc compiler en utilisant la commande `make`.

Le fichier `..._xdr.c` n'est produit que si nécessaire, que dans les cas où l'on utilise des types complexes dans le fichier de spécifications.

► Spécification

```
program UNAME_PROG {          /* definition du nom du programme RPC */
  version UNAME_VERS {      /* Nom de version */
    string GETUNAME(int uid) = 1; /* procedure, son type, ses
                                  arguments eventuels, son numero. */
  } = 5;                    /* Numero de la version */
} = 0x22222222;             /* Numero du programme */
```

► Fichier d'entête correspondant créé par rpcgen

```
#define UNAME_PROG 0x22222222
#define UNAME_VERS 5

#define GETUNAME 1
extern char ** getuname_5(int *, CLIENT *);
extern char ** getuname_5_svc(int *, struct svc_req *);
extern int uname_prog_5_freeresult (SVCXPRT *, xdrproc_t, caddr_t);
```

Le «programme» présenté ici ne comprend qu'une procédure, de nom `GETUNAME` dans les spécifications. Sous Linux, `rpcgen` va créer deux instances de cette procédure, une pour le client, appelée `getuname_5` et une pour le serveur, de nom `getuname_5_svc`.

La procédure prend un entier en argument et est censée rendre un type string. En fait la procédure coté serveur et coté client rendra un **pointeur de pointeur** sur un caractère, donc un pointeur de chaîne de caractères (une chaîne de caractères est déjà un pointeur).

Les RPC renvoient toujours un pointeur, quelque soit le type de résultat.

Le résultat peut être récupéré dans un argument de la fonction client (et renvoyé par un argument de la fonction serveur) en utilisant l'option **-M** de **rpcgen** sous Linux (**-A** sous Solaris). Les codes produits sont alors «thread safe».

Une procédure de nettoyage du résultat est également fournie.

Voyez que les noms donnés à au «programme» et à sa version seront directement utilisables, ils sont définis dans le fichier d'entête.

Comment retourner des cas d'erreurs		169/169
<ul style="list-style-type: none"> ▶ Quelle doit être la valeur retournée en cas d'erreur dans l'exécution de la procédure distante ? ▶ Utiliser le type «union discriminée» ▶ Exemple : 		
Spécifications		.h produit
<pre> union res switch (int errno) { case 0: string nom<255>; default: void; }; program UNAME_PROG { version UNAME_VERS { res GETUNAME(int uid) = 1; }; }; </pre>		<pre> struct res { int errno; union { char *nom; } res_u; }; typedef struct res res; extern res *getuname_5(int *, CLIENT *); </pre>

C'est un peu barbare en première lecture... L'essayer c'est l'adopter (!)

On imagine, dans le scénario ci-dessus que la procédure nous retourne le nom d'un utilisateur de la machine distante si on lui donne en argument le numéro de l'utilisateur. Passons sur la réalisation pratique de cette traduction pour aller directement au point qui nous occupe, à savoir le cas d'erreur. Que se passe-t'il si l'utilisateur n'existe pas. Comment indiquer ce cas. On pourrait rendre la chaîne de caractère **NULL** par exemple. Mais un utilisateur pourrait s'appeler **NULL** (des noms, des noms!!!), donc ce n'est pas bon.

On va utiliser un paramètre supplémentaire, appelé ici **errno** (mais son nom pourrait être autre. Si la procédure est correcte, on renvoie 0 dans ce paramètre et on peut alors lire la réponse, si on renvoie un nombre différent de 0, alors il n'y a pas de réponse (résultat **void**).

En pratique, on utilise la structure générée par **rpcgen** dans l'entête **xxx.h**. Dans la procédure on écrirait :

```
/* Cas correct */
```

```
res.errno = 0;
res.res_u.nom = lePointeurSurLeNom;

/* cas pas bon */
res.errno = 1; res.res_u.nom = NULL; /* le pointeur NULL */
return &res; /* on renvoie un pointeur */
```