



**IMT Atlantique**  
Bretagne-Pays de la Loire  
École Mines-Télécom

# Programmation Système et Réseaux

Christophe Lohr  
Printemps 2024

Les outils pour la programmation système  
Appels systèmes, fonctions, libc

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

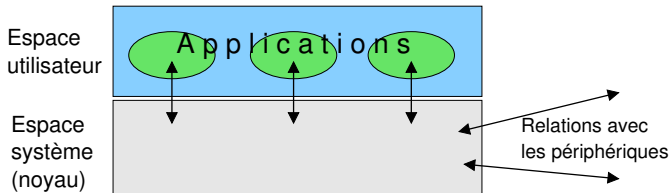
L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

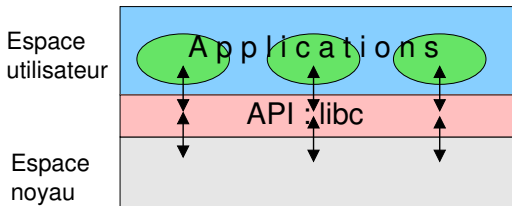
Programmation d'applications Réseau

Processus légers

- ▶ 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



- ▶ 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



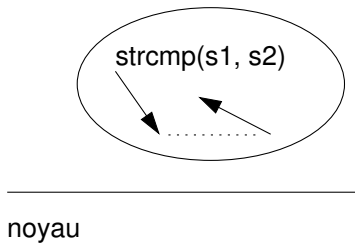
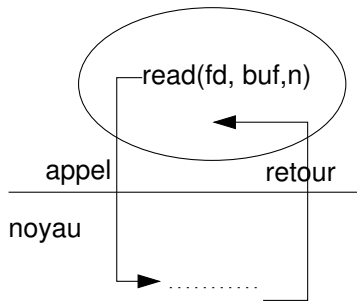
- ▶ Il existe différentes versions de libc
- ▶ Sous linux aujourd'hui nous en sommes à la version 6 : libc-6
  - ▶ La libc-6 est en réalité d'origine GNU, c'est la glibc-2 :-)
  - ▶ La libc-5 et la glibc sont différentes :-)
- ▶ Les pages du manuel de référence via la commande `man` ne sont pas obligatoirement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

- ▶ 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.

- ▶ 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

- ▶ 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





- ▶ 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 ?

- ▶ 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()`

- ▶ 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

- ▶ 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)

Les outils pour la programmation système

Les processus

Création, environnement, signaux, terminaison

Les entrées sorties

Structure d'un logiciel Unix/Linux

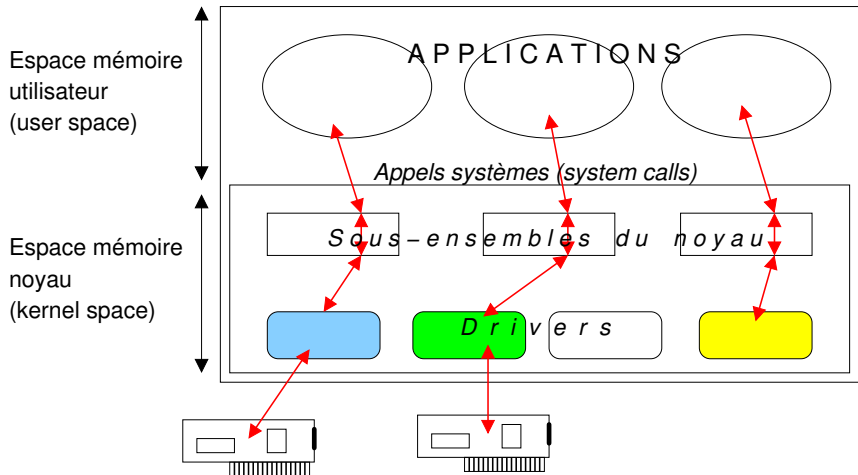
Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

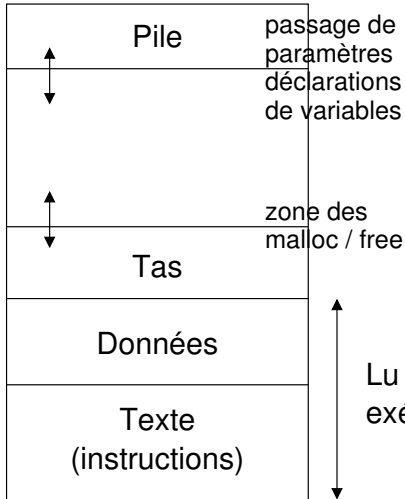
Processus légers



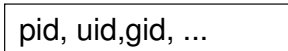
- ▶ Application
  - ▶ Au moins un fichier exécutable
  - ▶ Plus éventuellement des bibliothèques dynamiques
- ▶ Lancement d'une application
  - ▶ Chargement en mémoire du fichier exécutable et lancement de l'exécution de la fonction `main()`
  - ▶ Le fichier chargé en mémoire et en cours d'exécution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



## Espace utilisateur



## Espace noyau



- ▶ *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()`)

- ▶ 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

- ▶ 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

- ▶ 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

- ▶ 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.



- ▶ 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)

```
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 */
}
}
```

- ▶ 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()`

- ▶ 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()`

- ▶ 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()`

- ▶ 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 kill (kill -9 ou kill -KILL)
TERM	15	Envoyé via le clavier par la commande kill 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



- ▶ 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

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

▶ Syntaxe :

```
#include <signal.h>
```

```
typedef void (*sighandler_t)(int);
```

```
sighandler_t signal(int signum, sighandler_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 `sighandler_t`

► 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;  
}
```

▶ 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

- ▶ 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`

- ▶ 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



- ▶ 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);
```

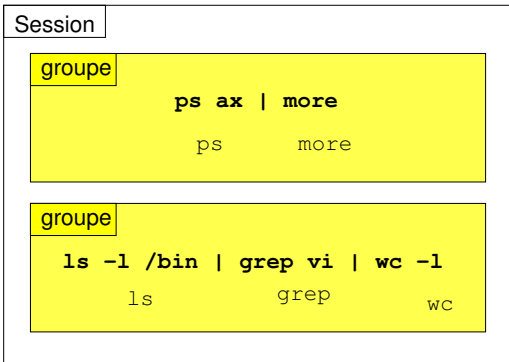
## ▶ Macros spécifiques

- ▶ `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

- ▶ 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()`

- ▶ 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

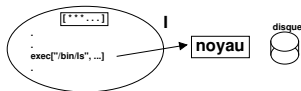


- ▶ Le processus lancé par l'appel du fichier exécutable crée un fils
  - ▶ Il se termine tout de suite par `exit()`
  - ▶ Son fils appelle `setsid()` et devient *Session Leader* et perd son terminal de contrôle, le processus fils est le démon
  - ▶ Le processus `init(1)` «adopte» le fils orphelin
 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

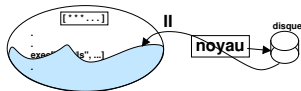
# Exécution d'un fichier par un processus

## L'appel système `execve()`

48/205



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

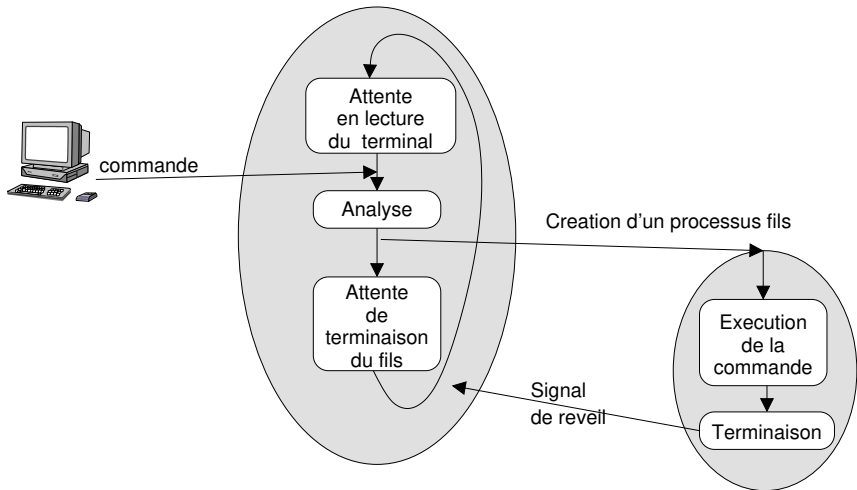


Étape III : le processus commence l'exécution de son nouveau code. Il ne peut y avoir retour à l'ancien code

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.



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



Les outils pour la programmation système

Les processus

**Les entrées sorties**

**Descripteurs et pointeurs de fichiers, primitives et fonctions**

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

Processus légers

- ▶ 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.

- ▶ 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...

- ▶ 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 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

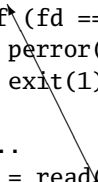
- ▶ Les classiques :
  - ▶ `read()`, `write()` : lire des blocks d'octets
  - ▶ `getchar()`, `putchar()` : lire et écrire un caractère
  - ▶ `gets()`, `puts()` : lire et écrire une ligne
  - ▶ `printf()`, `scanf()` : écrire et lire du texte formaté
- ▶ En utilisant spécifiquement les FILE Pointers `stdin/stdout/stderr`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



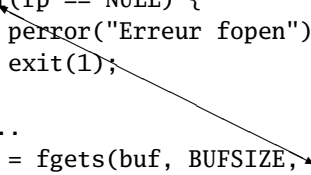
```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

#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 */
    ...
```



```
#include <stdio.h>
#define BUFZISE 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);
...
```



- ▶ 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

## ▶ Grace à `dup()` ou `dup2()`

- ▶ Exemple de redirection de sortie standard :

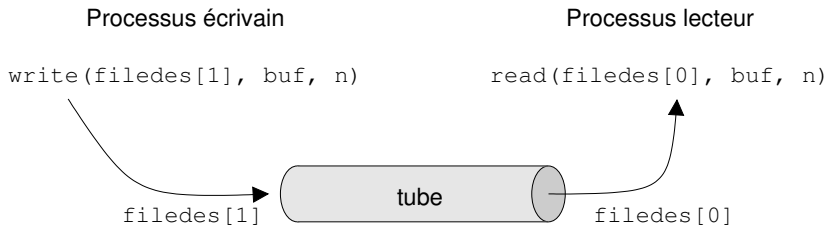
```
int fd;
...
fd = open("le_fichier", O_WRONLY | O_CREAT, 0666);
if (fd < 0) {...}

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



- ▶ Entre processus indépendants
- ▶ Accès au service : création d'un fichier spécial
- ▶ Utilisation : lectures écritures sur ce fichier

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

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

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

- ▶ 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)`



- ▶ 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)`

- ▶ 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

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

Les outils pour la programmation système

Les processus

Les entrées sorties

**Structure d'un logiciel Unix/Linux**

**Compilation, édition de liens, bibliothèques**

Les outils d'aide à la mise au point

L'utilitaire Make

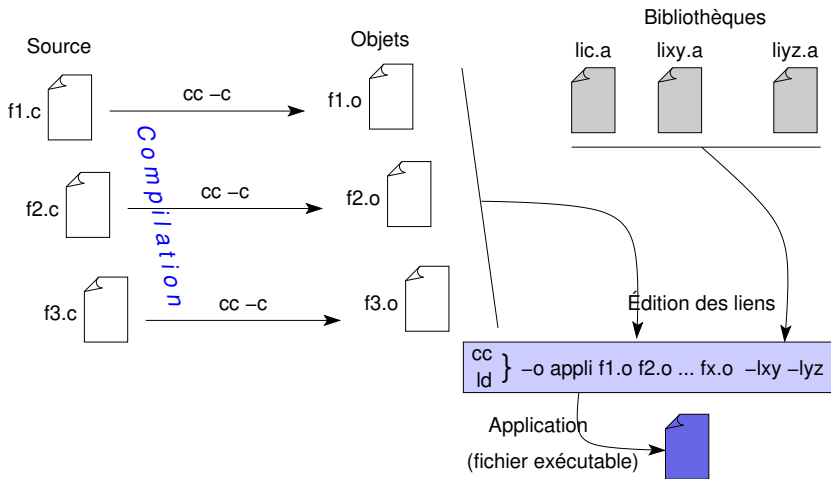
Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

Processus légers

- ▶ 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 70/205



```
$ 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
.LCO:
.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
    fildl -4(%ebp)
    leal -8(%esp), %esp
    fstpl (%esp)
    call log
    addl $16, %esp
    leal -8(%esp), %esp
    fstpl (%esp)
    pushl -4(%ebp)
    pushl $.LCO
    call printf
    addl $16, %esp
    ...
```

## ▶ 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`



- ▶ 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*)

- ▶ 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 1d`)
  - ▶ `-static`
  - ▶ `-dynamic` (option par défaut)

```
$ 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 -Irepinclude1 -Irepinclude2  
-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



Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

**Les outils d'aide à la mise au point**

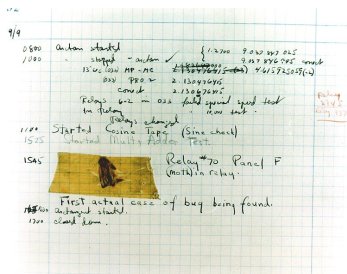
**Débugage, tracage, profilage**

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

Processus légers



Bug du Mark II découvert par Grace Hopper en 1947...

- ▶ Décorer son code de `printf()`
  - ▶ aux endroit judicieux / suspects
  - ▶ ... passage dans `for()`, les `if()` ...
  - ▶ affichage des valeurs de variables
  - ▶ etc.

- ▶ Lance une commande indiquée en argument et affiche tous les appels systèmes effectués par la commande ainsi que leur succès ou échec
- ▶ Peut suivre les processus pères et fils (option `-f`)
- ▶ Peut être lancée sur un processus qui est déjà lancé (option `-p`)
- ▶ La variante `ltrace` permet de tracer les appels aux bibliothèques dynamiques

- ▶ 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

- ▶ 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 appellable depuis emacs : `<M-x>`

- ▶ 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`

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

**L'utilitaire Make**

**automatisation des phases de compilation**

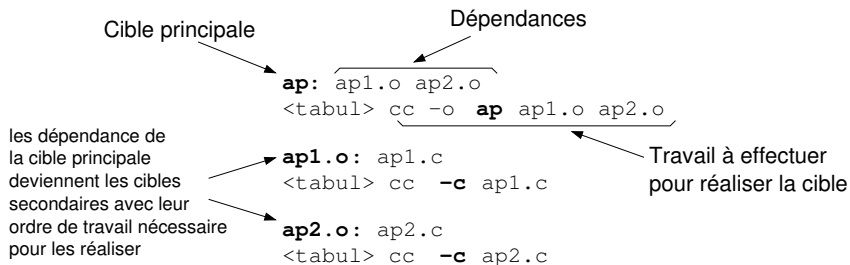
Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

Processus légers

- ▶ 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.





- ▶ 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.

Déclaration de variables ou macros

```
{ VAR1 = valeur1  
  VAR2 = valeur2  
  VARX = $(VAR1) $(VAR2) etc.
```

Utilisation des valeurs de variables  
ou macros:  
On peut référencer aussi des variables  
d'environnement définies hors  
du Makefile

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

Le bloc de travaux associé à une cible  
s'arrête à la première ligne ne  
débutant pas par une tabulation

```
dépendance1: dépendances...  
<tabul> travail...
```

- ▶ 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 95/205

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

- ▶ 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` ?

- ▶ 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 99/205

- ▶ 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.

- `$<` le nom du fichier dépendant déterminé par `make`
- `$@` 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).

- ▶ 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
```

- ▶ `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 ...`



- 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

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

**Paquetages logiciels : rpm, debian, Gnu tar  
gnu tar, debian, red hat, etc.**

Programmation d'applications Réseau

Processus légers

- ▶ 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`

- ▶ 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 -L 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 Linuxconf
linux#
```

# Packages Debian - Installation d'un logiciel 110/205

```
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.rpmshave}`
- ▶ Commande `yum` recherche, télécharge et installe un paquetage



- ▶ 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`
  - ▶ ...

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

**Programmation d'applications Réseau**

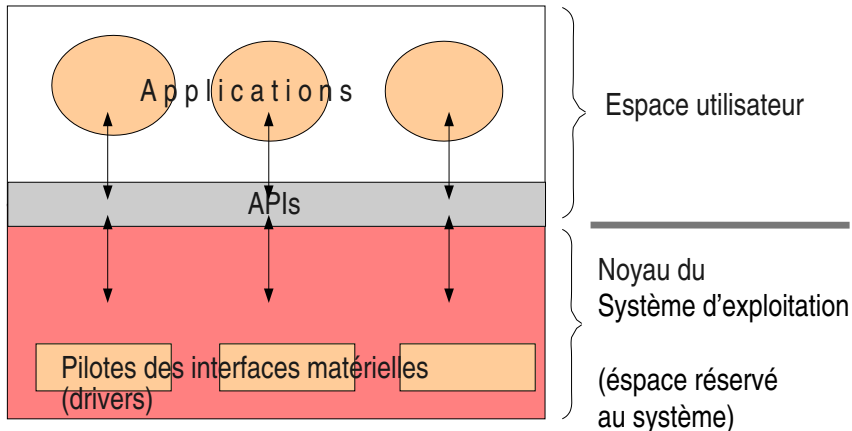
Concepts généraux

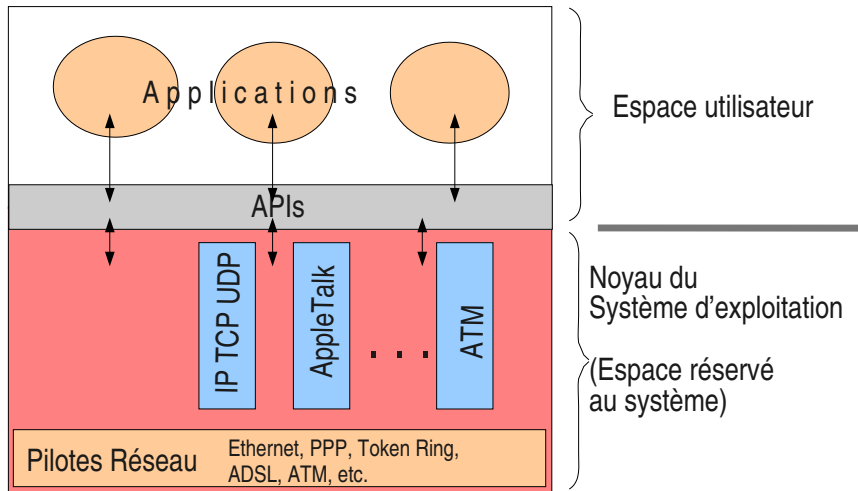
L'API Socket

L'API RPC

Processus légers

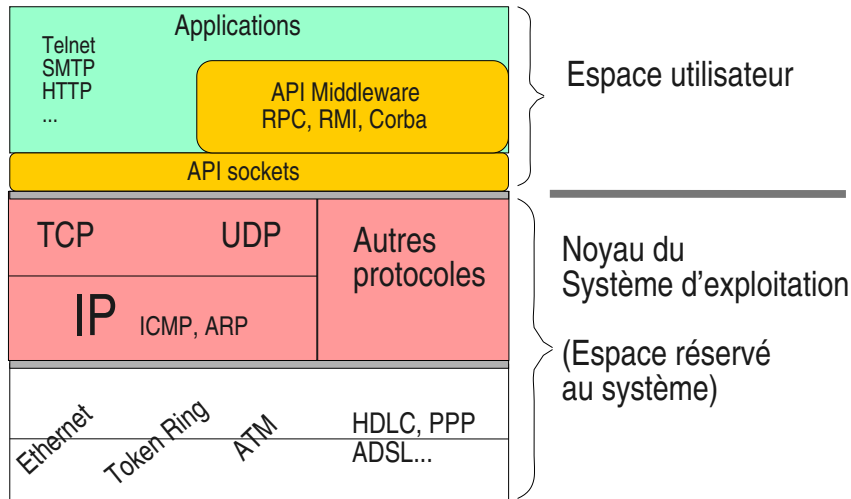
# Les applications et le système d'exploitation 115/205





# Les protocoles Internet et le système d'exploitation

117/205



Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

**Programmation d'applications Réseau**

Concepts généraux

**L'API Socket**

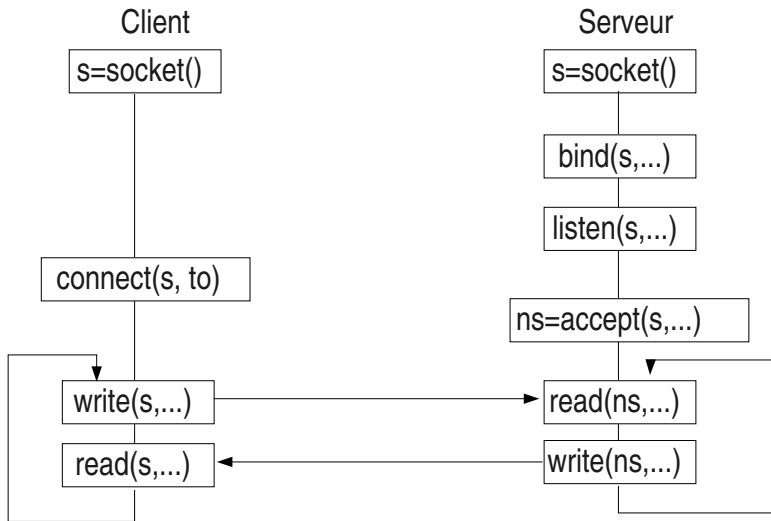
L'API RPC

Processus légers

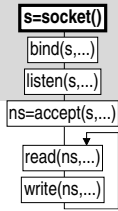
# Architecture Client-Serveur avec les sockets en

mode connecté

119/205



# 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

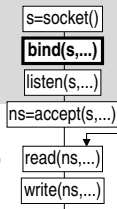
- ▶ La valeur renvoyée (`int`) est l'identificateur de ce point d'accès, c'est «la socket»
- ▶ `domain` : `PF_UNIX`, `PF_INET`, `PF_INET6`
- ▶ `type` : `SOCK_STREAM`, `SOCK_DGRAM`, `SOCK_RAW`
- ▶ `protocol` : permet d'indiquer le protocole à utiliser s'il n'est pas implicite avec le domaine et le type



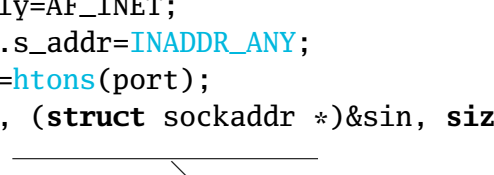
# La fonction `bind()`

```
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

```
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. */
};
```

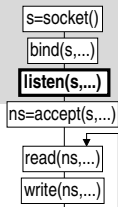
```
struct sockaddr_in6 {
    sa_family_t    sin6_family;    /* AF_INET6 */
    in_port_t      sin6_port;      /* numero de port */
    uint32_t       sin6_flowinfo;  /* flux IPv6 */
    struct in6_addr sin6_addr;      /* adresse IPv6 */
    uint32_t       sin6_scope_id;  /* Scope ID */
};

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

```
#define UNIX_PATH_MAX    108

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

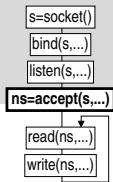
# La fonction `listen()`



```
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

# La fonction accept() |

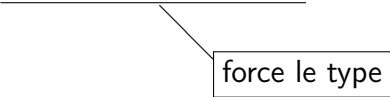


```
int accept (int s, struct sockaddr *addr,  
            socklen_t *addrlen);
```

- ▶ 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

- ▶ 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



# La fonction connect() |

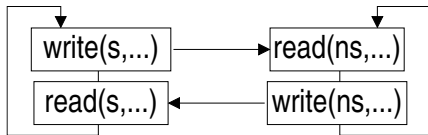
```
int connect(int sfd,  
            const struct sockaddr *serv_addr,  
            socklen_t addrlen);
```

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



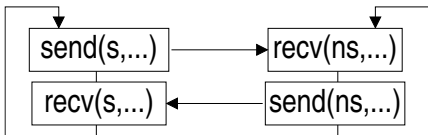
```
1 int sfd, r;
2 struct addrinfo hints, *result;
3
4 memset(&hints, 0, sizeof(struct addrinfo));
5 hints.ai_family = AF_UNSPEC;
6 hints.ai_socktype = SOCK_STREAM;
7 hints.ai_protocol = 0;
8 hints.ai_flags = 0;
9
10 r = getaddrinfo(argv[1], argv[2], &hints, &result);
11 if (r != 0) { ... }
12 ...
13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

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



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

- ▶ 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`, ...

## ▶ 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)

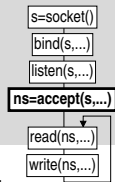
## ▶ Fonction `shutdown()`

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

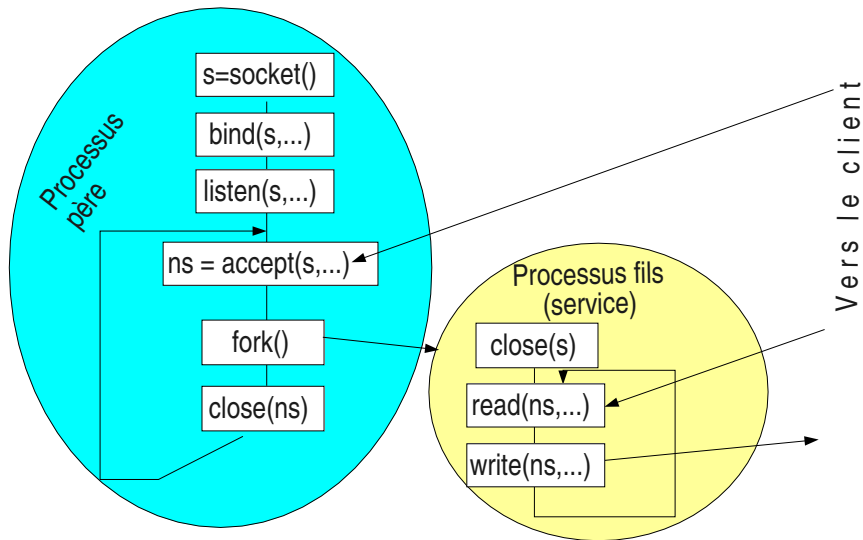
### ▶ Paramètre `how`

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

# Comment un serveur peut gérer plusieurs connexions simultanées



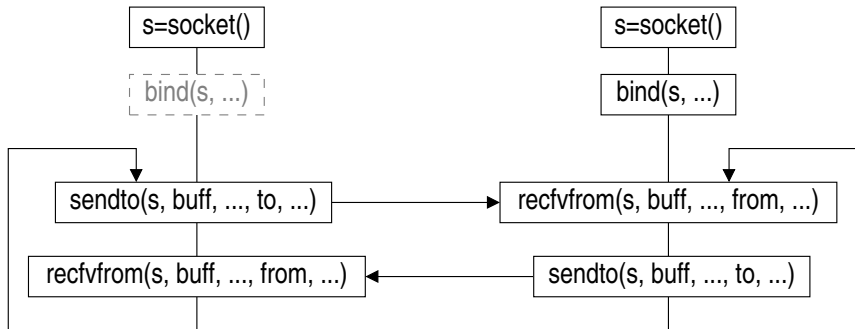
- ▶ `accept()` est bloquant et c'est bien embêtant...
- ▶ `accept()` ne peut «accepter» que sur une seule socket
- ▶ Il faudrait que le serveur puisse se dupliquer après le `accept()` pour d'une part revenir sur le `accept()` et d'autre part traiter la communication
- ▶ Solutions :
  - ▶ Générer un nouveau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



## Exemple de création de processus sous Unix136/205

```
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...
}
```





```
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);
```

- ▶ 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.

```
#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);
```

```
#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];
```

- ▶ 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.

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

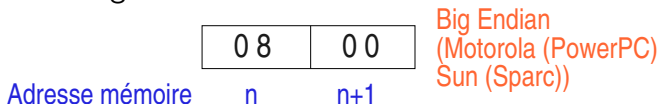
- ▶ Fonction `getsockname()`

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

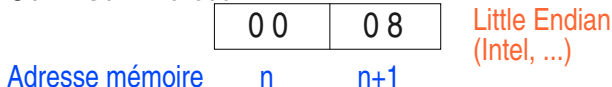
- ▶ Fonction `getpeername()`

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

- ▶ 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 :

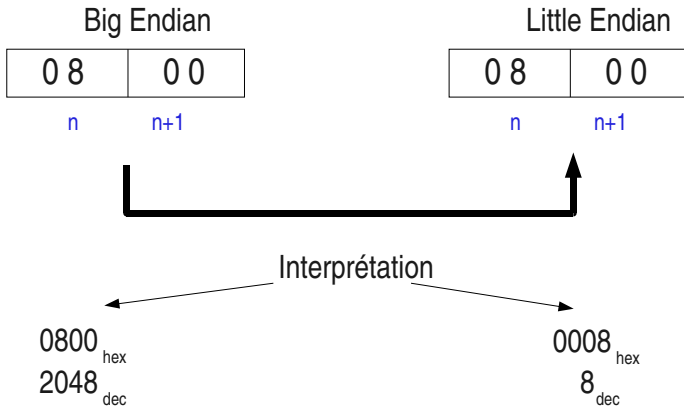


- ▶ Ou... Comme ceci :





- Envoyons  $0x0800$  ( $2048_{\text{déc}}$ ) d'une machine *Big Endian* vers une machine *Little Endian*



- ▶ Les conversions Machine/Réseau
  - ▶ Le réseau est «Big Endian»
  - ▶ Les fonctions de conversion :
    - ▶ Host to network : `htons()`, `htonl()`
    - ▶ Network to host : `ntohs()`, `ntohl()`
  - ▶ Exemple : `sin.sin_port = htons(1234);`
- ▶ Les fonctions `getaddrinfo()` et `getnameinfo()` gèrent cela elles-mêmes (une autre bonne raison de les utiliser)

# Les fonctions bloquantes sont parfois gênantes

147/205

## ▶ Exemple de protocole simple bloquant

```
write(sfd, "blabla...#", ...)
```

```
...
```

```
...
```

```
read(sfd, buf, ...)
```

```
read(s, buff, ...)
```

blocage

- ▶ 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

```
int select(int n,          /* nb descripteurs +1 */  
          fd_set *readfds, /* masque lecture */  
          fd_set *writefds, /* masque écriture */  
          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

- ▶ `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

151/205

```
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 }
```

- ▶ 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

## Exemple type

II

153/205

```
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    .....
```

## ▶ 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    .....
```

```
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

- ▶ `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

- ▶ 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()` a 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

- ▶ Classes spécifiques intrinsèques au langage  
Naturellement adaptées à TCP/UDP-IP
  - ▶ Socket : pour les clients
  - ▶ ServerSocket : 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

```
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 }
```

```
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()));
```



- ▶ `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)

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

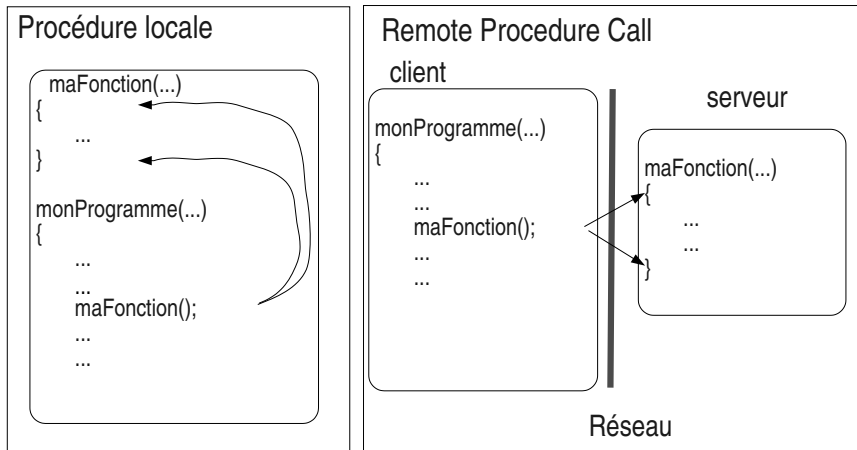
**Programmation d'applications Réseau**

Concepts généraux

L'API Socket

**L'API RPC**

Processus légers

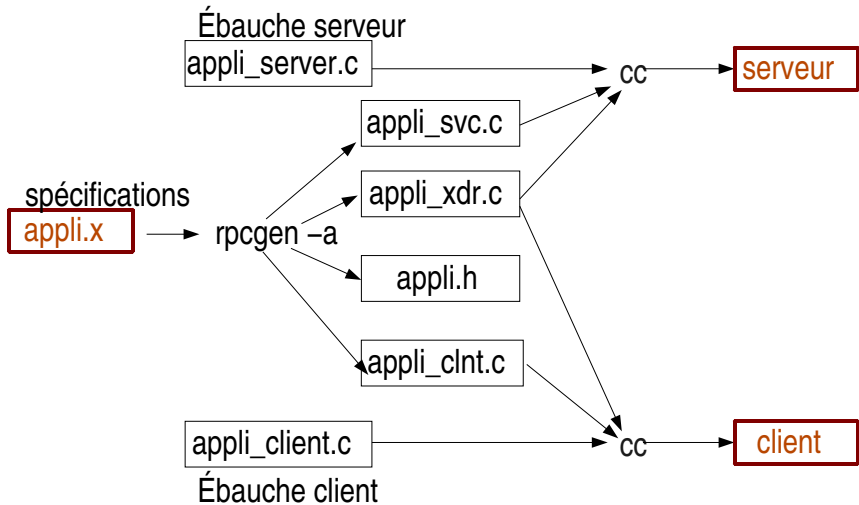


- ▶ Origine Sun Microsystems (1984)
  - ▶ Service NFS et NIS basés sur ces mécanismes
- ▶ Modèle en deux couches
  - ▶ La couche RPC (l'équivalent de la couche Session OSI)
    - ▶ Nombreuses fonctions : voir man rpc
  - ▶ La couche XDR : *eXternal Data Representation* (équivalent à la couche Présentation 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

- ▶ 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`

- ▶ 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

- ▶ «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)





## ► 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);
```

- ▶ Quelle doit être la valeur retournée en cas d'erreur dans l'exécution de la procédure distante ?
- ▶ Utiliser le type «union discriminée»
- ▶ Exemple :

### Spécifications

```
union res switch (int errno) {
  case 0:
    string nom<255>;
  default:
    void;
};

program UNAME_PROG { */
  version UNAME_VERS {
    res GETUNAME(int uid) = 1;
```

### .h produit

```
struct res {
  int errno;
  union {
    char *nom;
  } res_u;
};

typedef struct res res;

extern res *getuname_5(int *,
                       CLIENT *);
```

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

**Processus légers**

Concepts généraux

API Threads POSIX

Problèmes classiques

## Des tâches concurrentes

- ▶ Comme des processus “lourds” :
  - ▶ Organise code et algorithme de manière parallèle
  - ▶ Laisse l'OS gérer et exploiter le parallélisme matériel (multi-processeurs / multi-coeurs)
- ▶ Plus “légers” :
  - ▶ Mécanisme de création et de commutation de tâche moins coûteux
  - ▶ Partage des données (espace mémoire) entre tâches
- ▶ N'existent que à l'intérieur d'un processus lourd  
Et un processus lourd a au moins un thread (le `main()`)

Tous les threads d'un même processus partagent :

- ▶ Espace d'adressage (le code, les variables, etc.)
- ▶ Fichiers ouverts (table des *files-descriptors* ouverts)
- ▶ Variables d'environnement
- ▶ pid, parent pid, user id, groupe id, permissions, etc.
- ▶ handlers de signaux

# Non partagé entre threads d'un même processus

174/205

Chaque thread a son propre :

- ▶ Identifiant
- ▶ Pointeur d'instruction
- ▶ Pile d'exécution
- ▶ Ses masques de signaux
- ▶ Du stockage local de données
- ▶ `errno` (cas particulier du stockage local, transparent au programmeur)

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

**Processus légers**

Concepts généraux

API Threads POSIX

Problèmes classiques

- ▶ POSIX - Portable Operating System Interface (X as in UniX)  
Famille de normes et APIs standards IEEE 1003 (indépendants des constructeurs)
- ▶ API Threads POSIX : IEEE Std 1003.1c-1995
- ▶ Fournit :
  - ▶ Gestion : création attente annulation terminaison
  - ▶ Synchronisation : sémaphore condition barrière
  - ▶ Compléments : ordonnancement stockage-local etc.
- ▶ `#include <pthread.h>`
- ▶ `$ gcc -pthread prog.c -o prog`



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

- ▶ **thread** : pointeur sur un `pthread_t` (type opaque)  
Argument de sortie : identifiant du thread  
Voir `pthread_equal()` `pthread_self()`
- ▶ **attr** : les attributs, les dispositions du thread  
NULL = attributs par défaut
- ▶ **start\_routine** : exécutée par le thread créé  
De la forme  

```
void *start_routine(void *arg){...}
```
- ▶ **arg** : pointeur sur des arguments passés à la routine

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

void *routine(void *arg) {
    printf("thread_1%d\n", *((int *)arg) );
    return NULL;
}

int main(int argc, char *argv[]) {
    pthread_t thread;
    int arg = 1;

    if (pthread_create(&thread, NULL, routine, &arg) != 0) {
        perror("thread_1creation_1error");
        return EXIT_FAILURE;
    }

    for (int i = 0; i < 30; i++)
        printf("main\n");

    return EXIT_SUCCESS;
}
```

```
../..
```

```
typedef struct {  
    int numero;  
    char *nom;  
} arg_t;
```

```
void *routine(void *arg) {  
    arg_t *my_arg = arg;  
    printf("thread_[%d]: '%s'\n", my_arg->numero, my_arg->nom);  
    return NULL;  
}
```

```
int main(int argc, char *argv[]) {  
    pthread_t thread;  
    arg_t my_arg = {1, "Bob"};  
  
    if (pthread_create(&thread, NULL, routine, &my_arg) != 0) {  
        perror("thread_creation_error");  
        return EXIT_FAILURE;  
    }
```

```
../..
```

```
}
```

- ▶ Par terminaison de la routine qui a créée le thread

```
void *start_routine(void *arg) {  
    struct retval_t *retval;  
    ...  
    ...  retval = malloc(sizeof(struct retval_t));  
    ...  retval->... = ... ;  
    ...  
    return retval;  
    ...  
}
```

- ▶ Par appel, dans la routine ou une sous-routine, à :  
 void pthread\_exit(void \*retval);
- ▶ Valeur de retour récupérable par un pthread\_join()

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

- ▶ Joindre un thread déjà terminé ou attend sa terminaison
- ▶ `*retval` : Récupère sa valeur de terminaison ou la valeur `PTHREAD_CANCELED` si le thread a été annulé
- ▶ Retourne 0 si le thread a pu être rejoint
- ▶ ou bien un code d'erreur
  - ▶ `EDEADLK` : deadlock détectable (deux threads tentent de se joindre mutuellement)
  - ▶ `EINVAL` : thread non joignable, détaché
  - ▶ `EINVAL` : un autre thread tente déjà de joindre ce thread
  - ▶ `ESRCH` : mauvais identifiant de thread

```
...
pthread_t thread;
if (pthread_create(&thread, ...
...

...
struct retval_t *retval;
if (pthread_join(thread, &retval) == 0) {
    ... = retval->...;
    free(retval);
} else {
    perror("thread_join_error");
    ...
}
...
```

▶ Ne pas oublier de libérer la mémoire !

- ▶ Un thread est soit *joignable* soit *détaché*
- ▶ Créer un thread en mode détaché

```
pthread_attr_t attr;  
pthread_attr_init(&attr);  
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);  
pthread_create(..., &attr, ..., ...);  
pthread_attr_destroy(&attr);
```

- ▶ Détacher un thread existant

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

- ▶ Indique au système qu'il peut libérer les ressources du thread à sa terminaison sans attendre d'être rejoint par un autre.

Attention aux «*threads zombi*» !

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

- ▶ Un thread peut être annulable (par défaut), ou non
- ▶ de manière différée à un point d'annulation (par défaut), ou de manière asynchrone dès que possible
- ▶ Le thread est nettoyé (pile, verrous en cours, etc.), et n'est plus joignable. Il s'arrête.



On attend *le bon moment*. Différent points de vue :

- ▶ mutex : protéger l'entrée et la sortie à une section critique exclusive, c.à.d. sérialiser (la base !)
- ▶ condition : attendre qu'un prédicat devienne vrai
- ▶ barrière : on s'attend tous avant de passer à la suite ensemble
- ▶ rwlock : verrou pour gérer des lectures-écritures à plusieurs

Attente passive (efficace !)

- ▶ Protéger une section critique, une ressource partagée
  - ▶ typiquement : on est plusieurs à vouloir y accéder, un producteur et un consommateur ; il faut sérialiser cela
- ▶ Principe d'un verrou, jeton :
  - ▶ verrou libre : je le prend et je rentre dans la section critique et je n'oublie pas de le libérer rapidement, quand j'ai fini...
  - ▶ verrou occupé : je suis mis en attente
  - ▶ verrou libéré : l'un des thread en attente est réveillé ; il prend le jeton et rentre en section critique
- ▶ Le mutex :
  - ▶ est un genre de variable booléenne
  - ▶ possède une file d'attente de threads
  - ▶ est manipulé par deux opérations *atomiques* de test-et-modification : *lock* et *unlock*

## ▶ Mutex avec allocation statique

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- ▶ variable globale, création simple, attributs par défaut

## ▶ Mutex avec allocation dynamique

```
int pthread_mutex_init(pthread_mutex_t *restrict mutex,  
                       const pthread_mutexattr_t *restrict attr);
```

```
int pthread_mutex_destroy(pthread_mutex_t *mutex);
```

- ▶ préciser des *attributs* pour typer le mutex  
NULL = attributs par défaut
- ▶ peut être détruit & réinitialisé
- ▶ Attention : déverrouiller avant de détruire !
- ▶ mot clef `restrict` : pointeur unique et exclusif, responsabilité du programmeur

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);

int pthread_mutexattr_gettype(
    const pthread_mutexattr_t *restrict attr,
    int *restrict type);

int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
    int type);

int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- ▶ Et si on tente de le verrouiller plusieurs fois ?
- ▶ *rapide* : PTHREAD\_MUTEX\_NORMAL PTHREAD\_MUTEX\_DEFAULT,  
*à vérification d'erreur* : PTHREAD\_MUTEX\_ERRORCHECK,  
*récurusif* : PTHREAD\_MUTEX\_RECURSIVE
- ▶ Voir le man `pthread_mutexattr_init`

```
int pthread_mutexattr_getrobust(const pthread_mutexattr_t *attr,  
                                int *robustness);  
  
int pthread_mutexattr_setrobust(const pthread_mutexattr_t *attr,  
                                int robustness);
```

Et si un thread se termine sans libérer le mutex ?

- ▶ par défaut : le mutex reste indéfiniment bloqué
- ▶ si robuste : erreur EOWNERDEAD à traiter avec un `pthread_mutex_consistent()` avant de pouvoir refaire des lock unlock

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

- ▶ *lock unlock* : comme le nom l'indique...
- ▶ *trylock* : une opération lock non-blocante

- ▶ Attendre d'être prévenu qu'une ressource ou quelque chose soit prêt
  - ▶ typiquement : on est plusieurs à vouloir y accéder en lecture ; attend que l'écriture soit terminée
- ▶ Principe d'une condition
- ▶ *wait* : se mettre en attente jusqu'à être réveillé par une notification
- ▶ *signal* : envoie une notification
- ▶ Note : la *variable condition* doit être elle-même protégée par un mutex

## ▶ Condition avec allocation statique

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- ▶ variable globale, création simple, attributs par défaut

## ▶ Condition avec allocation dynamique

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

```
int pthread_cond_destroy(pthread_cond_t *cond);
```

- ▶ préciser des *attributs* pour typer la condition  
NULL = attributs par défaut
- ▶ Note : Linux ignore ce paramètre et n'implémente aucun type de condition...



```
int pthread_cond_wait(pthread_cond_t *cond,  
                      pthread_mutex_t *mutex);  
  
int pthread_cond_signal(pthread_cond_t *cond);  
  
int pthread_cond_broadcast(pthread_cond_t *cond);
```

- ▶ *wait* : dans l'ordre
  1. déverrouille le mutex
  2. met le thread en attente d'un signal sur la condition
  3. une fois réveillé donc, verrouille le mutex
- ▶ *signal* : réveille l'un (et un seul) des threads en attente (s'il y en a)
- ▶ *broadcast* : réveille tous les threads

- ▶ Attendre que tout le monde soit prêt avant de passer à la suite ensemble
- ▶ précise à *combien* de threads on s'attend

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

- ▶ attr = NULL      pas d'implémentation Linux

- ▶ Un genre de mutex étendu, spécialisé pour protéger des opérations de lecture-écriture

```
pthread_rwlock_t rwlock = PTHREAD_RWLOCK_INITIALIZER;  
  
int pthread_rwlock_init(pthread_rwlock_t *restrict rwlock,  
                        const pthread_rwlockattr_t *restrict attr);  
  
int pthread_rwlock_destroy(pthread_rwlock_t *rwlock);  
  
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_tryrdlock(pthread_rwlock_t *rwlock);  
  
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);  
int pthread_rwlock_trywrlock(pthread_rwlock_t *rwlock);  
  
int pthread_rwlock_unlock(pthread_rwlock_t *rwlock);
```

```
pthread_once_t once_control = PTHREAD_ONCE_INIT;  
  
int pthread_once(pthread_once_t *once_control, void (*init_routi
```

- ▶ Exécuter une routine qu'une seule fois
- ▶ Par exemple, pour gérer des initialisations globales...

```
int pthread_key_create(pthread_key_t *key,  
                       void (*destr_function) (void *));  
  
int pthread_key_delete(pthread_key_t key);  
  
int pthread_setspecific(pthread_key_t key,  
                        const void *pointer);  
  
void * pthread_getspecific(pthread_key_t key);
```

- ▶ Des variable globales MAIS des instances spécifique
- ▶ Dans une zone de mémoire associée à une *clef*
- ▶ Le nom de la variable clef est le même dans tous les threads mais pas sa valeur

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

- ▶ Pile de routines appelées automatiquement sur `pthread_cancel()` et `pthread_exit()`, mais pas sur `return` !
- ▶ *push* : empile une routine définie par l'utilisateur l'argument sera passé à la routine lorsqu'elle sera appelée
- ▶ *pop* : dépile la routine au sommet de la pile l'exécute si le paramètre `execute != 0`
- ▶ Une pile : exécutés dans l'ordre inverse de l'empilement
- ▶ Par exemple pour libérer des mutex...

```
#include <sched.h>

int sched_yield();
```

- ▶ Force le thread à relâcher le processeur
- ▶ Le thread est placé à la fin de la file d'attente
- ▶ Le thread suivant est alors activé
- ▶ Renvoie zéro en cas de succès (toujours, en fait)

Les outils pour la programmation système

Les processus

Les entrées sorties

Structure d'un logiciel Unix/Linux

Les outils d'aide à la mise au point

L'utilitaire Make

Paquetages logiciels : rpm, debian, Gnu tar

Programmation d'applications Réseau

**Processus légers**

Concepts généraux

API Threads POSIX

**Problèmes classiques**



## Définition

Une fonction est dite **thread-safe** si elle peut être appelée “sans risque” par plusieurs threads concurrents, c.à.d. sans introduire d’interblocage, sans corruption de donnée, ou autre problème de synchronisation.

- ▶ Par conséquent : n’utiliser que des fonctions thread-safe dans vos threads !
- ▶ ... ou alors en étant certain que les fonctions non thread-safe ne sont appelées que par un seul thread à la fois (mutex)

- ▶ Typiquement : utilisation de variable globale ou variable statique
- ▶ Attention : beaucoup de fonctions historiques de la libc sont non thread-safe !
- ▶ Lire le man !
- ▶ et considérer a-priori que tout est non thread-safe sauf mention contraire explicite...
- ▶ Les fonctions non thread-safe de la libc ont généralement une variante *réentrant* "...\_r()" et donc thread-safe

- ▶ Rendre la fonction **réentrant** :
  - ▶ pas de variable globales, ni de variable statique
  - ▶ n'utiliser que des variables locales, ou les variables passées en paramètre
  
- ▶ Ou bien **protéger** ! ... par un mutex ou autre

- ▶ Rappel :
  - ▶ les “disposition” (handlers) sont globaux
  - ▶ les masques sont spécifiques à chaque thread
- ▶ Un signal va être traité, a-priori, par n'importe quel thread (s'interrompt pour exécuter le handler)
- ▶ Stratégies :
  - ▶ Ne rien tenter, laisser faire la nature (conseillé!)
  - ▶ Gérer les signaux dans un seul thread, et tout masquer dans les autres
  - ▶ Surtout éviter les mélanges! (c.à.d. tenter de gérer certains signaux dans certains threads et pas dans d'autres...)
- ▶ `pthread_kill()` : Envoyer un signal à un thread précis du processus.  
Faisable, à vos risques et périls...

- ▶ passage en paramètre à un nouveau thread d'un pointeur dans la pile de l'appelant
- ▶ accès à la mémoire globale sans protection (mutex...)
- ▶ création de deadlock : deux threads tentent d'acquérir deux ressources dans un ordre différent
- ▶ tentative de ré-acquérir un verrou déjà détenu
- ▶ utilisation des signaux Unix et de threads
- ▶ oublie de gérer la terminaison (`pthread_join()`)
- ▶ prudence insuffisante lors de l'utilisation de cast (`void *`) pour les paramètres et retours des threads (`pthread_create()`)
- ▶ ...