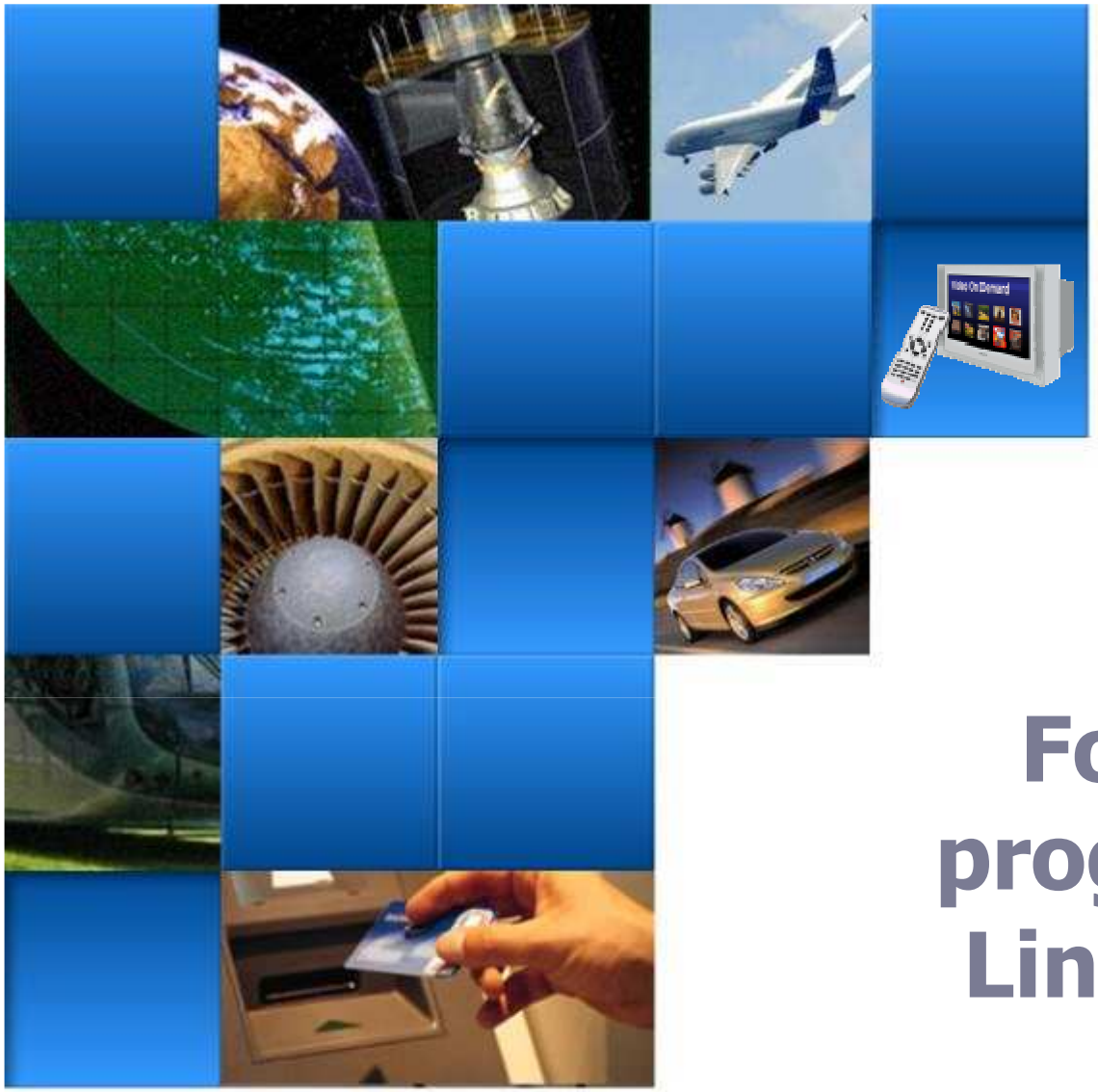


EUROGICIEL GROUPE

Formation programmation Linux système

GIFAS aerospace Syntec informatique

Version du 11 octobre 2010

Sommaire

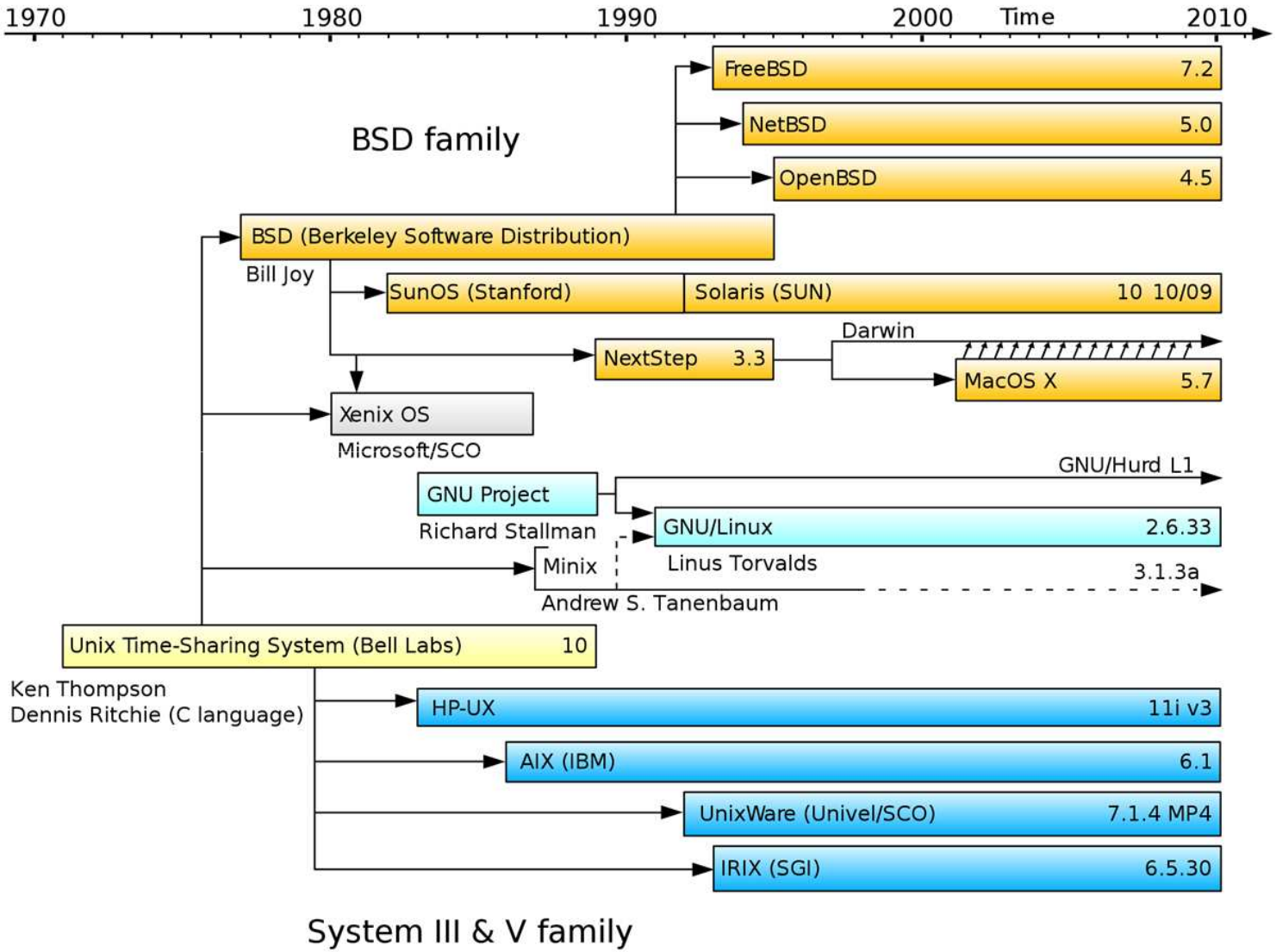
Introduction

Qu'est-ce que Linux ?

- Linux est le noyau du système d'exploitation GNU/Linux
- GNU/Linux est le mariage du noyau Linux et des programmes et utilitaires systèmes du projet GNU (boot loader, toolchain, librairies C, shells, etc)
  - Logiciel libre (open source)

- Plusieurs organisations (à but lucratif ou non) distribuent le noyau Linux, les utilitaires GNU, plus d'autres logiciels comme le gestionnaire de bureau Gnome ou KDE, sous la forme d'un CD ou DVD facile à installer
  - On parle alors de distributions Linux
  - Exemples :
    - Red-Hat
    - Fedora
    - Ubuntu
    - Debian
    - Gentoo
    - …

Unix – Historique



FORMATION

EUROGICIEL GROUPE

Linux – Historique

- **1991** : Le noyau Linux est écrit à partir de zéro en 6 mois par Linus Torvalds dans sa chambre de l'université d'Helsinki, afin de contourner les limitations de son PC 80386
- **1991** : Linus distribue son noyau sur Internet. Des programmeurs du monde entier le rejoignent et contribuent au code et aux tests
- **1992** : Plus de 100 développeurs travaillent sur le noyau
- **1992** : Linux est distribué sous la licence GNU GPL
- **1994** : Sortie de Linux 1.0
- **1994** : La société Red Hat est fondée par Bob Young et Marc Ewing, créant ainsi un nouveau modèle économique basé sur une technologie OSS
- **1995** : GNU/Linux et les logiciels libres se répandent dans les serveurs Internet
- **2001** : IBM investit 1 milliard de dollars dans Linux
- **2002**: L'adoption massive de GNU/Linux démarre dans de nombreux secteurs de l'industrie

Linux – Historique (suite) – Premier contact

Premier post effectué par Linus Torvald sur Usenet :

Path: gmdzi!unido!fauern!ira.uka.de!sol.ctr.columbia.edu!zaphod.mps.ohio-state.edu!wupost!uunet!mcsun!news.funet.fi!hydra!klaava!torvalds From: torva...@klaava.Helsinki.FI (Linus Benedict Torvalds) Newsgroups: comp.os.minix Subject: What would you like to see most in minix? Summary: small poll for my new operating system Keywords: 386, preferences Message-ID: <1991Aug25.205708.9541@klaava.Helsinki.FI> Date: 25 Aug 91 20:57:08 GMT Organization: University of Helsinki Lines: 20

Hello everybody out there using minix –

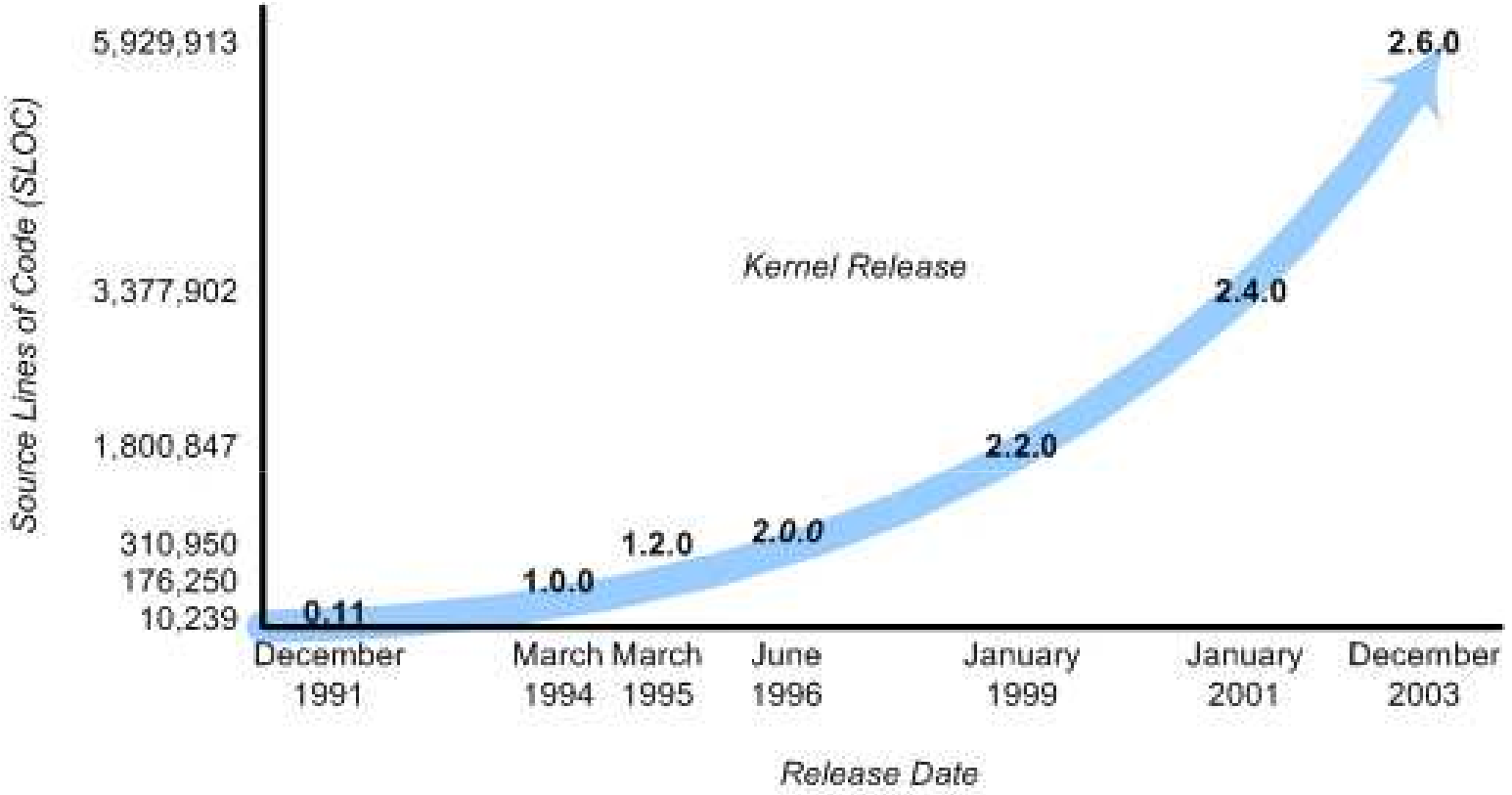
I'm doing a (**free**) **operating system** (just a hobby, won't be big and professional like gnu) for 386(486) AT clones. This has been brewing since April, and is starting to get ready. I'd like any feedback on things people like/dislike in minix, as my OS resembles it somewhat (same physical layout of the file-system (due to practical reasons) among other things).

I've currently ported bash(1.08) and gcc(1.40), and things seem to work. This implies that I'll get something practical within a few months, and I'd like to know what features most people would want. Any suggestions are welcome, but I won't promise I'll implement them :-)

Linus (torva...@kruuna.helsinki.fi)

PS. Yes - it's free of any minix code, and it has a multi-threaded fs. It is NOT portable (uses 386 task switching etc), and it probably never will support anything other than AT-harddisks, as that's all I have :-(.

Linux – Historique (suite)



Linux 2.6.32 :

SLOC = 12 Millions

Debian 4.0 :

SLOC = 283 Millions

FORMATION

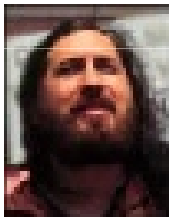
EUROGICIEL GROUPE

Linux – Organisation du développement

Linus TORVALD
le décideur et responsable du projet ; propriétaire du nom "Linux"

Andrew MORTON
adjoint au projet, le numéro 2

David Miller



Alan COX
responsable de la partie réseau.

Russell KING
responsable de l'architecture ARM.

Andi KLEEN
responsable de l'architecture x86-64.

etc
...

FORMATION

EUROGICIEL GROUPE

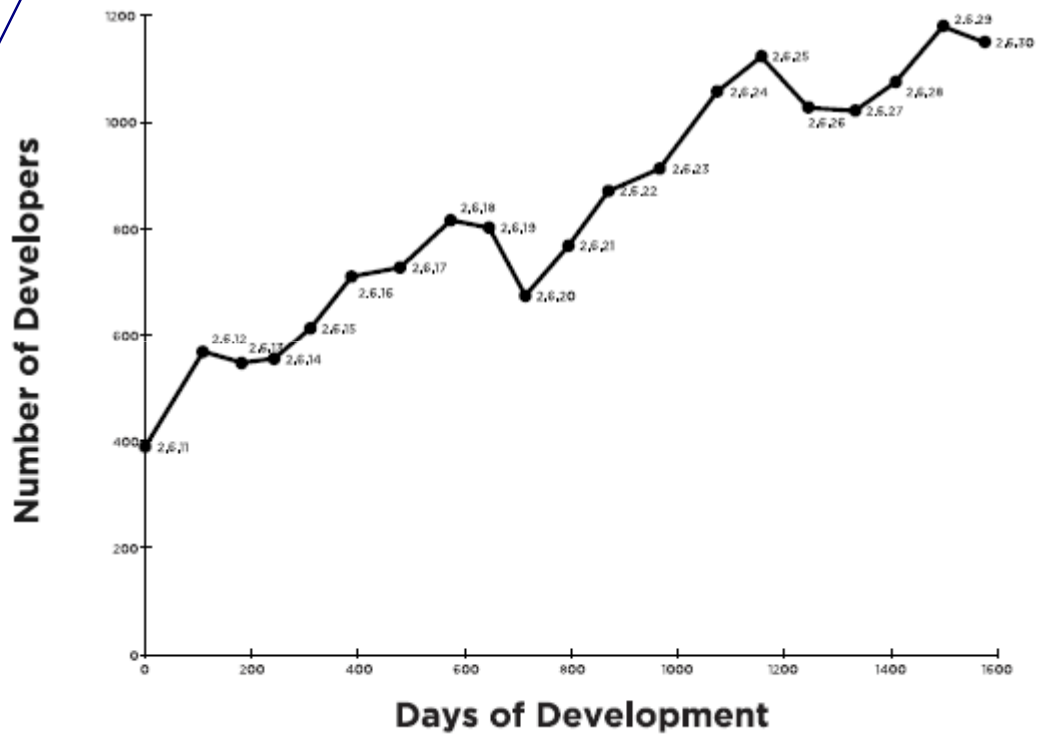
Linux – L'équipe des développeurs

" Un travail de longue durée par une équipe de Geeks / Nerds / professionnels passionnés. "

Linux – Acteurs professionnels du noyau

Kernel Version	Number of Developers	Number of Known Companies
2.6.11	389	68
2.6.12	566	90
2.6.13	545	94
2.6.14	553	90
2.6.15	612	108
2.6.16	709	111
2.6.17	726	120
2.6.18	815	133
2.6.19	801	128
2.6.20	673	138
2.6.21	767	143
2.6.22	870	180
2.6.23	912	181
2.6.24	1,057	194
2.6.25	1,123	228
2.6.26	1,027	203
2.6.27	1,021	188
2.6.28	1,075	213
2.6.29	1,180	230
2.6.30	1,150	240
All	4,910	532

Forte croissance…



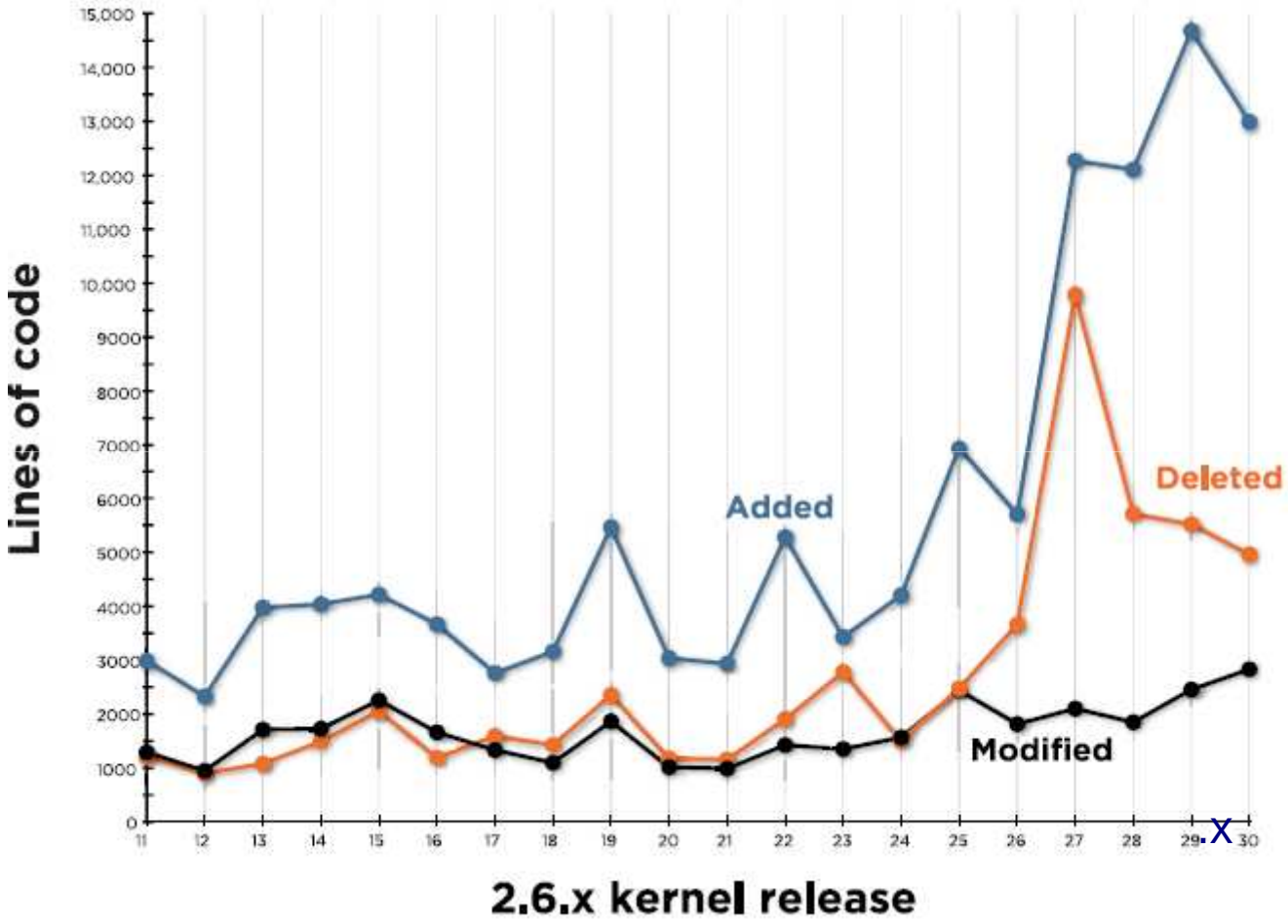
2.6.11 = mars 2011

Sources :
http://www.kernel.org/pub/linux/kernel/people/gregkh/kernel_history/
http://www.linuxfoundation.org/publications/whowriteslinux.pdf

FORMATION

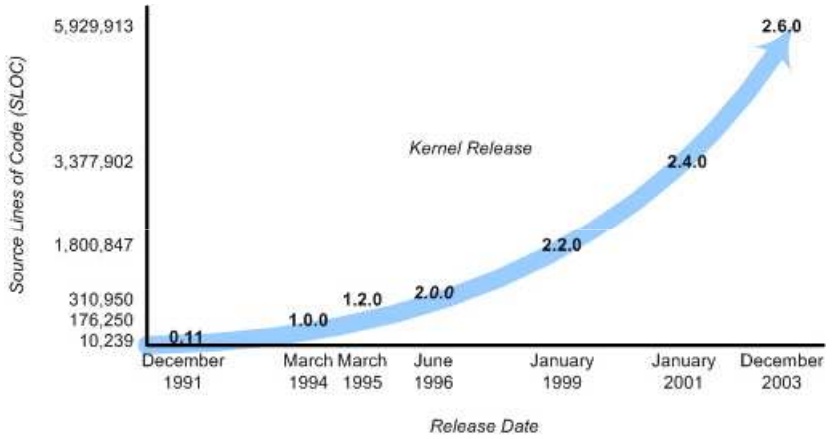
EUROGICIEL GROUPE

Linux – Statistiques noyau



La version 2.6.33 du noyau a été publiée le 24 février 2010

Linux – Statistiques noyau (suite)



Kernel Version	Files	Lines
2.6.11	17,090	6,624,076
2.6.12	17,360	6,777,860
2.6.13	18,090	6,988,800
2.6.14	18,434	7,143,233
2.6.15	18,811	7,290,070
2.6.16	19,251	7,480,062
2.6.17	19,553	7,588,014
2.6.18	20,208	7,752,846
2.6.19	20,936	7,976,221
2.6.20	21,280	8,102,533
2.6.21	21,614	8,246,517
2.6.22	22,411	8,499,410
2.6.23	22,530	8,566,606
2.6.24	23,062	8,859,683
2.6.25	23,813	9,232,592
2.6.26	24,273	9,411,841
2.6.27	24,356	9,630,074
2.6.28	25,276	10,118,757
2.6.29	26,702	10,934,554
2.6.30	27,911	11,560,971

Linux – Statistiques noyau (suite)

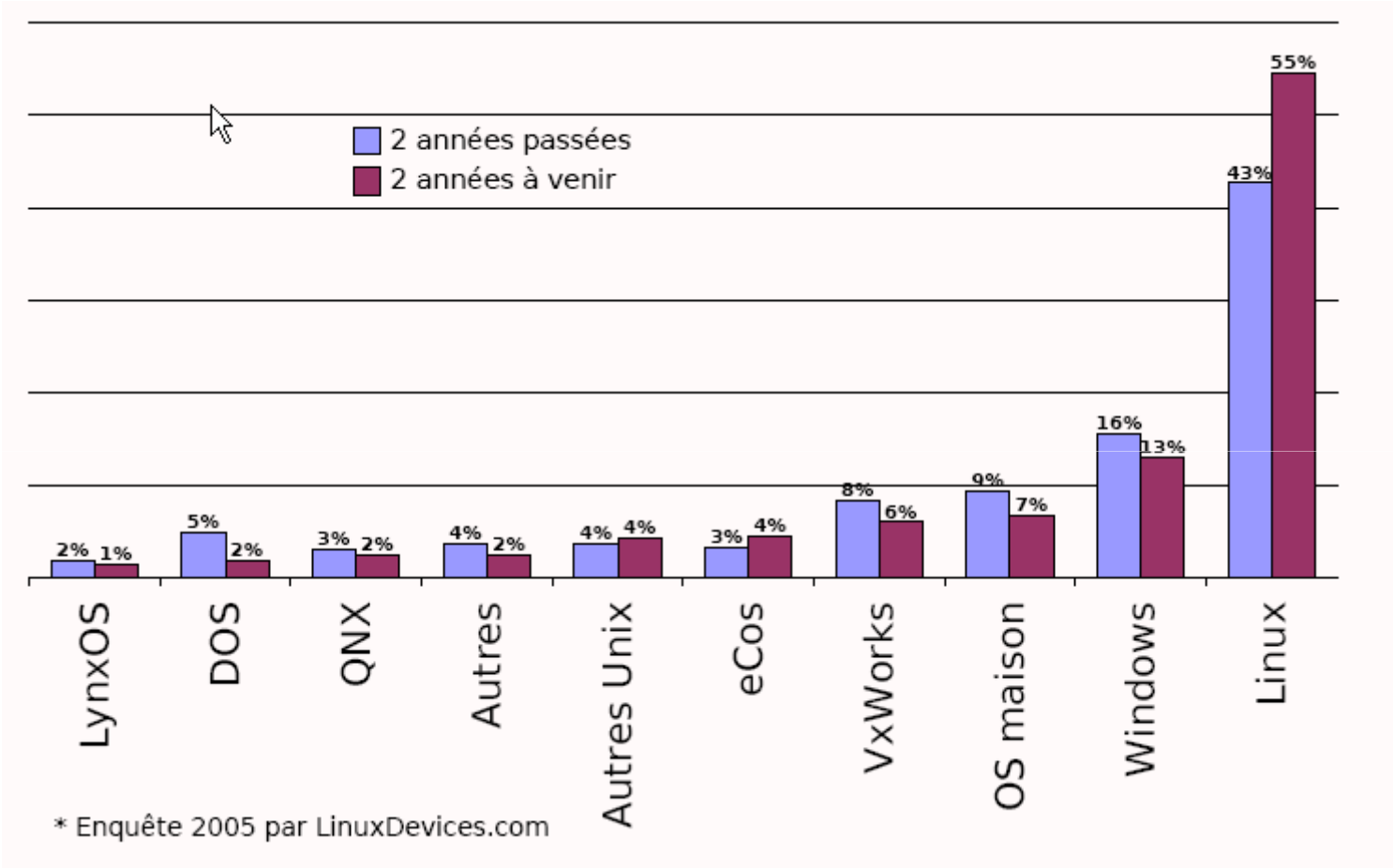
Kernel Version	Lines Added per Day	Lines Deleted per Day	Lines Modified per Day
2.6.11	3,224	1,360	1,290
2.6.12	2,375	951	949
2.6.13	4,443	1,553	1,711
2.6.14	4,181	1,637	1,726
2.6.15	5,614	3,454	2,219
2.6.16	3,853	1,388	1,649
2.6.17	3,635	2,469	1,329
2.6.18	3,230	1,497	1,096
2.6.19	6,013	2,900	1,862
2.6.20	3,120	1,342	1,013
2.6.21	3,256	1,479	982
2.6.22	6,067	2,694	1,523
2.6.23	3,747	3,034	1,343
2.6.24	6,893	4,181	1,563
2.6.25	7,980	3,488	2,430
2.6.26	5,698	3,662	1,815
2.6.27	12,270	9,791	2,102
2.6.28	12,105	5,707	1,850
2.6.29	14,678	5,516	2,454
2.6.30	12,993	4,958	2,830

Linux – Contributeurs

Company Name	Number of Changes	Percent of Total
None	13,850	21.1%
Red Hat	7,897	12.0%
IBM	4,150	6.3%
Novell	4,021	6.1%
Intel	3,923	6.0%
Unknown	2,765	4.2%
Oracle	2,003	3.1%
Consultant	1,480	2.3%
Parallels	1,142	1.7%
Fujitsu	1,007	1.5%
Academia	992	1.5%
Analog Devices	889	1.4%
Renesas Technology	884	1.3%
SGI	755	1.2%
Movial	738	1.1%
Sun	639	1.0%
HP	628	1.0%
Freescale	613	0.9%
Marvell	601	0.9%
MontaVista	574	0.9%
AMD	552	0.8%
Nokia	549	0.8%
Vyatta	513	0.8%
Google	512	0.8%
Atheros Communications	494	0.8%
NTT	445	0.7%
linutronix	445	0.7%
XenSource	432	0.7%
Simtec	414	0.6%
Astaro	411	0.6%

- Contributeurs depuis la 2.6.24
- En 2009, 55000 changements (2.8 millions de lignes de code) ont été fait par 2700 développeurs
- 70% des développeurs sont payés par leur entreprise pour leur contribution
- « Unknown » : pas de connaissance de l'entreprise (petit contributeurs mais nombreux)
- « None » : contributeur indépendant

Linux – Linux vs. autres OS



Source :
http://www.linuxfordevices.com/c/a/Linux-For-Devices-Articles/Embedded-Linux-market-snapshot-2005/

Linux – Linux vs. autres OS

- Marché très fermé des OS propriétaires :
  - Pas de compatibilités entre eux
  - Kits de développement coûteux et figés
  - Royalties élevées
  - Dépendance d'un éditeur
  - Problème de pérennité
  - Portabilité réduite

- Aujourd'hui, fort développement de Linux et des logiciels libres
  - Alternative très sérieuse allant du prototypage au produit fini

Projet GNU

- GNU est un projet de système d'exploitation composé exclusivement de logiciels libres
- GNU a été initié en 1983 par Richard Stallman à la suite de son désaccord avec les licences de Berkeley
- En 1985, Stallman fonde la Free Software Foundation
- Le système d'exploitation GNU reprend les concepts d'Unix, mais son implémentation est indépendante et originale
- Le projet GNU avait prévu le développement d'un noyau (Hurd)
- L'arrivée du noyau Linux rendit disponibles les logiciels du projet GNU sur x86 (Torvalds a mis Linux sous licence GPL en 1992)
- L'ensemble des distributions Linux portent l'empreinte du projet GNU (ne serait-ce que dans leurs licences), d'où l'appellation distribution GNU/Linux défendue par R. Stallman

Licences – General Public License

- La licence GPL fixe les conditions légales de distribution des logiciels libres du projet GNU.
- Richard Stallman et Eben Moglen en sont les auteurs.

- L'objectif de la licence GNU GPL est de garantir à l'utilisateur les droits suivants sur un programme :
  - La liberté d'exécuter le logiciel, pour n'importe quel usage.
  - La liberté d'étudier le fonctionnement d'un programme et de l'adapter à ses besoins, ce qui passe par l'accès aux codes sources.
  - La liberté de redistribuer des copies.
  - La liberté d'améliorer le programme et l'obligation de rendre publiques les modifications afin que l'ensemble de la communauté en bénéficie.

Licences – General Public License (suite)

- La licence GPL requiert que les modifications et les travaux dérivés soient aussi placés sous licence GPL.
  - Ne s'applique qu'aux logiciels distribués (donc pas utilisés en interne, en cours de développement, en phase de test, etc.)
  - Intégrer du code d'un programme GPL dans un autre implique de placer ce programme sous GPL.
  - Tout programme utilisant du code GPL (lié statiquement ou dynamiquement) est considéré comme une extension de ce code et donc placé sous GPL.
- GCC a une clause spéciale : il est permis de compiler un programme sans qu'il soit placé sous GPL.

- http://www.gnu.org/licenses/gpl-faq.html
- http://ffii.org/index.en.html

Licences – Lesser General Public License (LGPL)

- La licence LGPL permet d'intégrer une partie non modifiée d'un programme LGPL sans contrainte.
  - Si le code est modifié, alors le programme l'intégrant doit être redistribué sous LGPL.

- Il est autorisé de lier, statiquement ou dynamiquement, un programme avec une librairie LGPL, sans contrainte de licence ou de distribution de code source

- La LGPL autorise à lier un programme sous cette licence à du code non LGPL, sans pour autant révoquer la licence.

Licences – Autres licences OSS

- La licence MIT ou X11 donne à toute personne recevant le logiciel le droit illimité de l'utiliser, le copier, le modifier, le fusionner, le publier, le distribuer, le vendre et de changer sa licence. La seule obligation est de mettre le nom des auteurs avec la notice copyright.

- La licence BSD a essentiellement les mêmes conditions que la licence MIT/X11. Avant 1999, elle contenait une clause publicitaire maintenant disparue.

- Il existe une multitude d'autres licences OSS.

- Toutes les licences OSS ne sont pas compatibles avec la GPL.

Licences – Le noyau Linux

- Le noyau Linux est exclusivement sous licence GPL version 2.
- Les modules propriétaires sont tolérés (mais non recommandés) tant qu'ils ne sont pas considérés comme dérivés de code GPL.
- Les pilotes propriétaires ne peuvent pas être liés statiquement au noyau.

Linux – Caractéristiques principales

- Le noyau Linux est monolithique (un binaire statique)
- Les pilotes de périphériques tournent en espace noyau
- Les pilotes peuvent être compilés comme modules, et chargés ou déchargés tandis que le système tourne
- Le noyau est multithreadé et est préemptable
- Le noyau supporte SMP
- La plupart des interfaces graphiques sont en espace utilisateur

- Portabilité et support matériel : voir liste des processeurs supportés
- Scalabilité : tourne sur des super ordinateurs aussi bien que sur des petits appareils
- Conformité aux standards et interopérabilité (POSIX)
- Support réseau avec une pile très complète
- Sécurité : mandatory access control, extended FS attributes, etc.
- Stabilité et fiabilité

Linux – Portabilité du noyau

- Il est développé principalement en langage C (pas de langage objet tel que le C++) avec une légère couche en assembleur.
- Pour les portages sur une nouvelle architecture, il convient donc d'adapter cette dernière.
- En résumé, ce qui est propre à un linux donné est localisé dans le répertoire `arch/` des sources du noyau.

- Minimum : processeurs 32 bits, avec ou sans MMU (Memory Management Unit).
- Architectures :
  - 32 bits : alpha, arm, cris, h8300, i386, m68k, m68knommu, mips, parisc, ppc, s390, sh, sparc, um, v850
  - 64 bits : ia64, mips64, ppc64, sh64, sparc64, x86_64
  - Voir `arch/README` ou `Documentation/arch/README` pour plus de détails

Linux – Etat des versions des noyaux

Linux 2.2

- Branche plus maintenue au même titre que les versions 1.x.

Linux 2.4

- Mûr et exhaustif.
- Les développements sont arrêtés; peu de support de la communauté de développeurs Linux.
- En passe de devenir obsolète.
- Toujours bien si les sources, outils et support viennent de vendeurs Linux commerciaux.

Linux 2.6

- Toutes nouvelles fonctionnalités et performances accrues (POSIX IPC, threads, scheduler, préemption, asynchronous I/O, etc.).
- Supporté par la communauté de développeurs Linux.
- Désormais mature et exhaustif. La plupart des pilotes ont été mis à niveau.

Outils – Editeurs

- Emacs
- Xemac
- Vim
- Nedit
- Eclipse
- Source Navigator
- KScope

Outils – man

- `man [section] <nom_de_commande>`
- Exemple, pour voir le manuel de la commande ftp, il faut taper :

```
$ man ftp
```

- Les pages du manuel Unix sont divisées en plusieurs sections :
  - 1 : Commandes utilisateur
  - 2 : Appels système
  - 3 : Fonctions de bibliothèque
  - 4 : Périphériques et fichiers spéciaux
  - 5 : Formats de fichier
  - 6 : Jeux
  - 7 : Divers
  - 8 : Administration du système
  - 9 : Interface du noyau
- Chaque section possède une page d'introduction qui présente la section, disponible en tapant `man <section> intro`
- L'option -k liste les sections contenant la commande :

```
$ man –k open
$ man 2 open
$ man 1 open
```

Outils – GCC

- GCC est le compilateur standard sous Linux
- L'auteur original est R. Stallman, pour le projet GNU, en 1984
- GCC fut le premier compilateur portable ANSI C avec optimisation
- GCC a été étendu à d'autres langages que le C ou le C++ (Fortran, ADA, Java, Objective-C, Lisp, Smalltalk, ...)
- L'acronyme GCC réfère maintenant à "GNU Compiler Collection"
- GCC est non seulement un compilateur natif mais aussi un compilateur croisé
- GCC est écrit en C et peut se compiler lui-même

- Pour les programmes C et C++, GCC est une suite d'outils composée (essentiellement) de :
  - Préprocesseur (`cpp`)
  - Assembleur (`gas`)
  - Editeur de lien ou linker (`ld`)
- La commande `gcc` se charge d'invoquer le préprocesseur, l'assembler et le linker

Outils – GCC (suite)

- La commande du compilateur C est gcc, celle du compilateur C++ est g++

```
$ gcc hello.c
$ g++ hello.cpp
```

- Le nom du ficher binaire final est par défaut `a.out`. Il est souvent spécifié avec l'option -o

```
$ gcc hello.c -o hello
```

- Par défaut, GCC ne produit aucun warning, il est donc recommandé d'utiliser l'option -Wall

```
$ gcc -Wall hello.c -o hello
```

- Plusieurs fichiers sources peuvent être spécifiés (le "lien" est fait par un fichier header .h inclus dans les fichiers .c) :

```
$ gcc –Wall main.c hello.c -o hello
```

- Lorsqu'un beaucoup de fichiers sources sont utilisés, si un seul est modifié, il devient une perte de temps de tous les recompiler.
  - On préfère alors utiliser une compilation en deux étapes ; la compilation en fichiers objets suivie de l'édition de lien :

```
$ gcc -Wall -c main.c
$ gcc -Wall -c hello.c
$ gcc main.o hello.o -o hello
```

Outils - make

- make est un utilitaire permettant d'invoquer des commandes générant des fichiers, mais seulement si elles sont nécessaires
- Pour chaque commande, on spécifie quels sont les fichiers dont elle dépend
- La commande n'est exécutée que si un ou plusieurs de ces fichiers est plus récent que le fichier généré
- make sert principalement à faciliter la compilation et l'édition de liens puisque le résultat final dépend d'opérations précédentes

- Les règles sont enregistrées dans un fichier `Makefile`
- La syntaxe d'une règle est :

```
cible: liste de dépendances
[tab] commande
```

- Une règle peut dépendre d'une autre
- Il est possible de définir et d'utiliser des variables
- Des variables prédéfinies existent pour accéder à la cible, aux dépendances, etc

`$@` `$<` `$?` `$^` `$*`

Outils – make (exemple)

```
CC=g++
CFLAGS=-c -Wall
LDFLAGS=
SOURCES=main.cpp hello.cpp
OBJECTS=$(SOURCES:.cpp=.o)
EXECUTABLE=hello

all: $(SOURCES) $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(LDFLAGS) $(OBJECTS) -o $@

.cpp.o:
    $(CC) $(CFLAGS) $< -o $@
```

- http://www.gnu.org/software/make/manual/

Outils – Autres utilitaires

- **lclint / splint** : vérifient et détectent les problèmes courants dans du code
  - http://www.splint.org/
- **indent** : indentation de source code

- **objdump** : permet de récupérer beaucoup d'informations en provenance d'un fichier objet, comme son désassemblage, le contenu des variables initialisées, etc.

- **nm** : permet de lister le contenu d'un fichier objet, avec ses différents symboles privés ou externes, les routines, les variables, etc.

- **gprof** : profileur : enregistre dans un fichier les temps de présence dans chaque fonction du programme.

- **gcov** : couverture de code : permet de savoir quelles lignes de code sont exécutées.

- **strace** : permet de détecter tous les appels-système invoqués par un processus.

Généralités

Généralités – Appels systèmes

- Typiquement sur architectures avec MMU :
  - Le noyau a accès à tout l'espace d'adressage et à tout le hardware (mode superviseur, mode protégé ou ring 0)
  - Les applications ont une espace d'adressage virtuel
  - Les applications n'ont accès qu'à un nombre limité de ressources

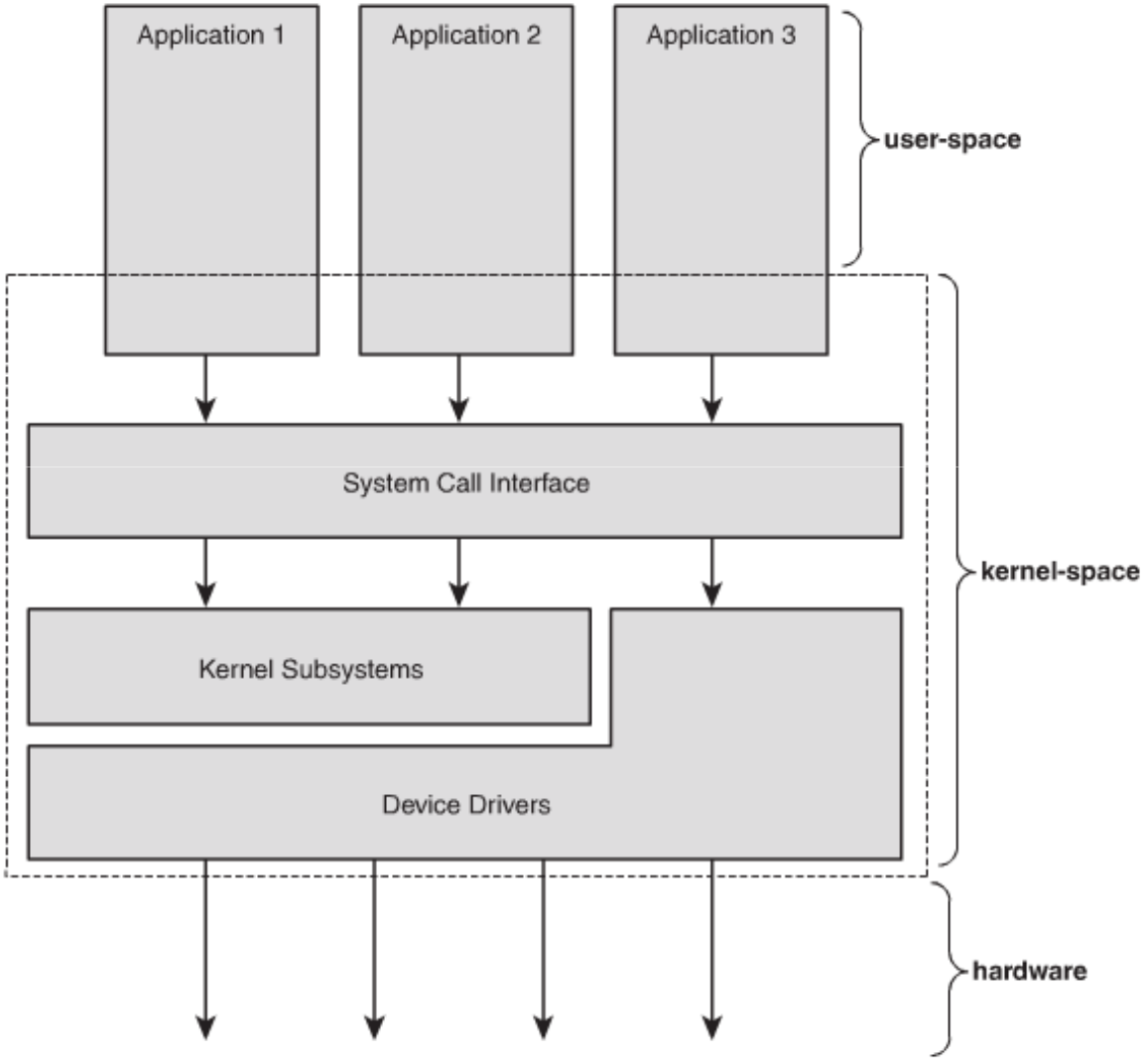
- Les applications communiquent avec le noyau via des appels systèmes (syscalls)
- Linux à une interface système (SCI) d'environ 340 fonctions

- Normalement une librairie (telle que glibc) interface les applications et la SCI

- Les applications ne peuvent être compilées et liées directement au noyau
  - Le noyau propose une interface qui permet aux applications de signaler un appel (trap)
  - Cette interface est dépendante de l'architecture (ex : `int 0x80` sur x86)
  - Cet appel cause un changement de contexte espace utilisateur / espace noyau

- A chaque appel, un double changement de contexte doit être fait
  - Ceci est couteux en performance

Généralités – Appels systèmes



Généralités – errno, perror, strerror

- La variable `errno` contient une constante de description de l'erreur
- Voir la page `man` de la fonction pour connaitre les possibles valeurs retournées et les possibles valeur d'`errno`

- Il n'est plus nécessaire de déclarer une variable globale `errno`
- Il suffit d'inclure le header `<errno.h>` pour avoir accès à la variable `errno`
- `<errno.h>` défini des constantes représentant chaque erreur

- Les fonctions `perror()` (dans `<stdio.h>` ou `<cstdio>`) et `strerror()` (dans `<string.h>`) permettent d'afficher un message à propos d'une valeur de `errno`

Généralités – errno, perror, strerror – Exemple

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

int main(void)
{
   int s = socket(PF_INET, SOCK_STREAM, 0);
   if (s == -1) {
        // affiche "socket error: " + le message d'erreur
        perror("socket error");
        exit(EXIT_FAILURE);
   }
   if (listen(s, 10) == -1) {
        // affiche "an error: " + le message d'erreur
        printf("an error: %s\n", strerror(errno));
        exit(EXIT_FAILURE);
   }
   /* do some work... */
   exit(EXIT_SUCCESS);
}
```

Fichiers

Fichiers – Philosophie Unix

- Sous Unix, presque toute ressource est un fichier :
  - Fichiers normaux
  - Répertoires
    - Un répertoire est un fichier contenant une liste de fichiers
  - Liens symboliques
    - Fichiers référant à d'autres fichiers
  - Périphériques
    - Lecture et écriture sur périphériques comme sur des fichiers normaux
  - Tubes
    - Utilisés pour enchainer des programmes

      ```
      cat *.log | grep error
      ```
  - Sockets
    - Communications interprocessus

Fichiers – Généralités

- Noms des fichiers
  - Sensibles à la casse
  - La longueur maximale du nom dépend du système de fichier (typiquement 255 caractères)
  - Le chemin (path) est typiquement limité à 4096 caractères
  - Peuvent contenir n'importe quel caractère sauf `/` et `NUL`
  - Nom commençant par un point = fichier caché
  - Les extensions ne sont pas utilisées
  - Types déterminés par le contenu des fichiers
- Les fichiers `.` et `..` sont automatiquement créés dans chaque répertoire
- Chaque fichier appartient à un utilisateur et à un groupe
  - Utilisateurs définis dans `/etc/passwd`
  - Groupes définis dans `/etc/group`

Fichiers – Système de fichiers

- Structure du système de fichiers :

`/`	racine (rootfs)
`/bin`	commandes systèmes de base
`/boot`	image(s) kernel, initrd, fichiers de configuration
`/dev`	fichiers représentant des périphériques
`/etc`	fichiers de configuration du système
`/home`	répertoires utilisateurs
`/lib`	librairies systèmes de base
`/mnt`	points de montage
`/opt`	programmes ajoutés par l'administrateur
`/proc`	informations systèmes exportées par le noyau
`/root`	répertoire de l'utilisateur root
`/sbin`	commandes réservées à l'administrateur
`/sys`	contrôle du système et de périphériques
`/tmp`	fichiers temporaires
`/usr`	commandes (non essentielles)
	`/usr/bin, /usr/lib, /usr/sbin`

Fichiers – Système de fichiers (suite)

`/usr/local`	programmes ajoutés par l'administrateur
`/var`	données utilisées par le système ou programmes `/var/log`, `/var/spool`, ...

- La structure du système de fichier est définie par le "Filesystem Hierarchy Standard" (FHS)
  - http://www.pathname.com/fhs/

Fichiers – /dev

- `/dev` contient des fichiers spéciaux représentant une interface fichier sur des périphériques de type caractère ou bloc

- Exemples :

`/dev/null`	data sink
`/dev/zero`	retourne toujours `NUL`
`/dev/random`	retourne des nombres aléatoires, peut être bloquant
`/dev/urandom`	retourne des nombres aléatoires, non bloquant
`/dev/console`	console courante (souvent `/dev/tty0`)
`/dev/full`	se comporte comme un périphérique plein
`/dev/hda`	premier disque dur
`/dev/hda2`	deuxième partition

- Tous les périphériques ne sont pas représentés dans `/dev` (ex : interfaces réseaux)

Fichiers – /dev (suite)

- Un périphérique du type caractère est symbolisé par un c
- Un périphérique du type bloc est symbolisé par un b

```
crw-rw-rw- ... /dev/null
brw-rw---- ... /dev/sda
```

- Les fichiers de `/dev` indiquent au noyau quel driver utiliser et comment accéder à un périphérique particulier par deux numéros, appelés majeur et mineur :
  - `crw-rw-rw- ... 1, 3 Mar 14 09:34 /dev/null`
  - `brw-rw---- ... 8, 0 Mar 14 09:34 /dev/sda`

- Le noyau maintient une liste des numéros majeurs
- La liste est disponible dans `linux/Documentation/devices.txt`

- La commande `mknod` permet de créer un fichier périphérique dans `/dev` en précisant les numéros majeur et mineur

- Linux 2.6.15 et suivants peuvent utiliser un système de fichier virtuel en lieu et place de `/dev`

Fichiers – /proc et /sys

- `/proc` et `/sys` sont des systèmes virtuels, exposés par le noyau
- `/proc` exporte des informations sur les processus et des paramètres noyau

- Exemples :
  - `/proc/cpuinfo`
  - `/proc/meminfo`
  - `/proc/vmstat`
  - `/proc/sys/net/ipv4/ip_forward`

- `/sys` exporte des informations sur les périphériques et leurs drivers (PCI, USB, power, etc)
  - N'existe que depuis 2.6
  - Les paramètres noyaux exposés dans `/proc` sont petit à petit déplacés dans `/sys`

Fichiers – Droits d'accès

- Trois types de droits d'accès :
  - Lecture
  - Ecriture
  - Exécution
- Ces types de droits existent pour trois types d'utilisateurs :
  - Utilisateur propriétaire du fichier (user)
  - Utilisateurs dans le même groupe que le propriétaire du fichier (group)
  - Les autres utilisateurs non compris dans les deux premiers types (others)

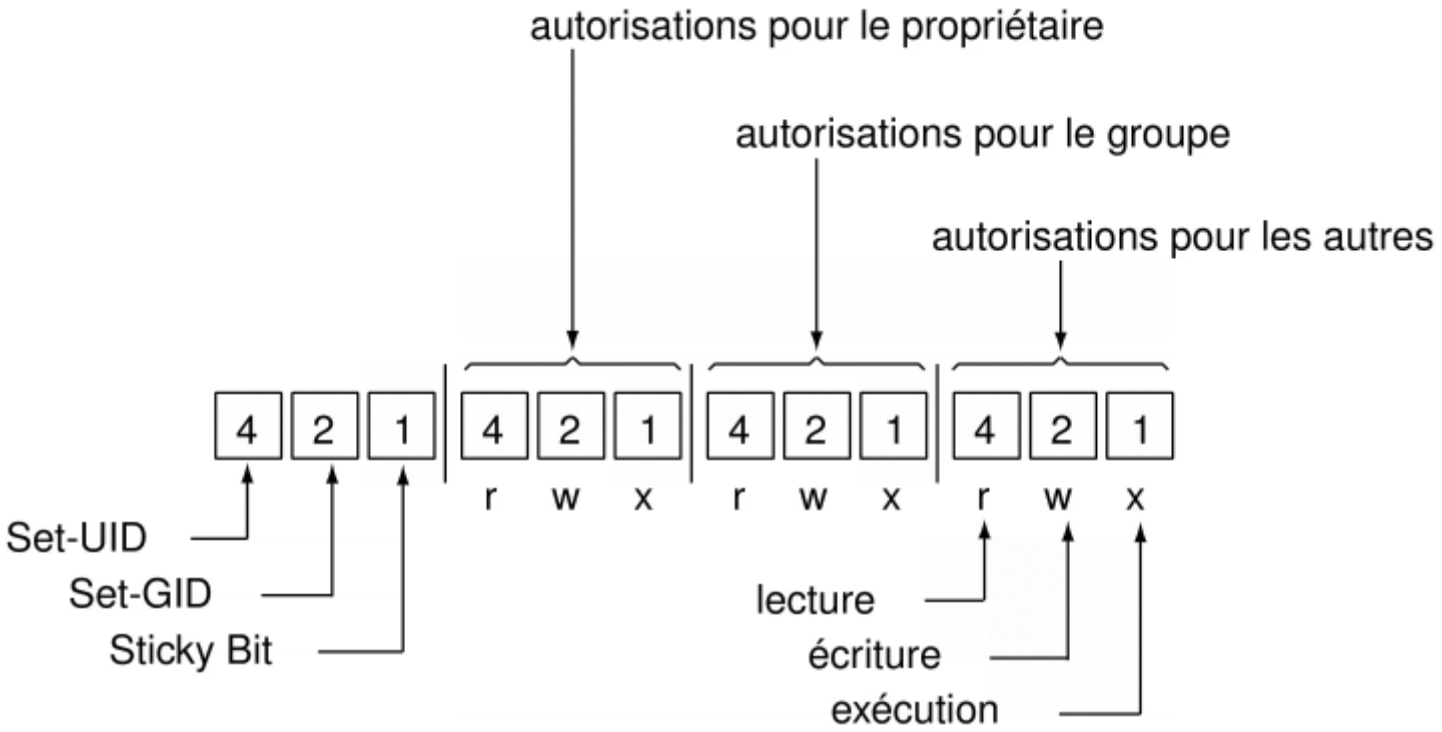
- Exemple :

```
$ ls –l example
-rwxr-xr-- 1 joe team 104K Dec 28 13:28 example
```

  - L'utilisateur `joe` peut lire, modifier et exécuter le fichier
  - Les utilisateurs du groupe `team` peuvent lire, exécuter, mais pas modifier
  - Les autres utilisateurs ne peuvent que lire

- La commande pour changer les droits d'accès est `chmod`
  - Les droits sont spécifiés en octal :
    - `$ chmod 644 file`
  - Ou avec des symboles :
    - `$ chmod u=rwx,g=rx,o=r file`

Fichiers – Droits d'accès (suite)



Fichiers – umask

- umask : masque de création de fichier par l'utilisateur (user file creation mode mask)
- Le umask définit les permissions par défaut d'un répertoire ou d'un fichier créé
- C'est un attribut des processus Unix
- La commande permettant de consulter sa valeur ou de la modifier est… `umask`

- umask spécifie les permissions à enlever à la création d'un fichier ou répertoire, assumant que les permissions sans umask seraient `rw-rw-rw-` et `rwxrwxrwx` respectivement.

- Exemples :
  Un umask de `222` donne `r--r--r--`
  Un umask de `022` donne `rw-r--r--`
  Un umask de `066` donne `rw-------`

Fichiers – Modifier bits

- setuid et setgid : change l'utilisateur (UID) ou le groupe (GID) effectif d'un programme avec celui du fichier :

```
$ ls –l example
-rwsr-xr-x 1 joe users 104K Dec 28 13:28 example
```

- setuid et setgid sont ignorés pour les scripts (fichiers commençant par #!)

- sticky est ignoré pour les fichiers
- Pour les répertoires, les fichiers avec le bit sticky ne peuvent être effacés que par l'utilisateur propriétaire (ou root)

- Posix définies des constantes pour les droits d'accès et le modifier bit, définies dans <sys/stat.h>

Fichiers – Types des fichiers

- Le mode spécifie si le fichier est de type :
  - Répertoire
  - Périphérique mode caractère
  - Périphérique mode bloc
  - Tube
  - Socket
  - Lien symbolique

- Le type de fichier est déterminé à sa création et ne peut être changé ensuite

Fichiers – Inodes

- Un fichier est identifié et décrit sur un disque dur par son numéro d'inode (index node) et non pas par son nom
- Un inode est une structure du système de fichier, qui contient toutes les informations sur un fichier :
  - Utilisateur et groupe propriétaires
  - Type de fichier (ordinaire, répertoire, lien symbolique, etc.)
  - Droits d'accès
  - Dates de dernier accès, de dernière modification, etc.
  - Taille du fichier
  - Nombre de liens
  - Localisation des blocs sur le disque

- Lorsqu'un fichier est ouvert, le noyau copie la structure inode en mémoire
- Il y a en plus l'état de l'inode (verrouillé, lecture seule, etc.)

Fichiers – Descripteurs de fichiers

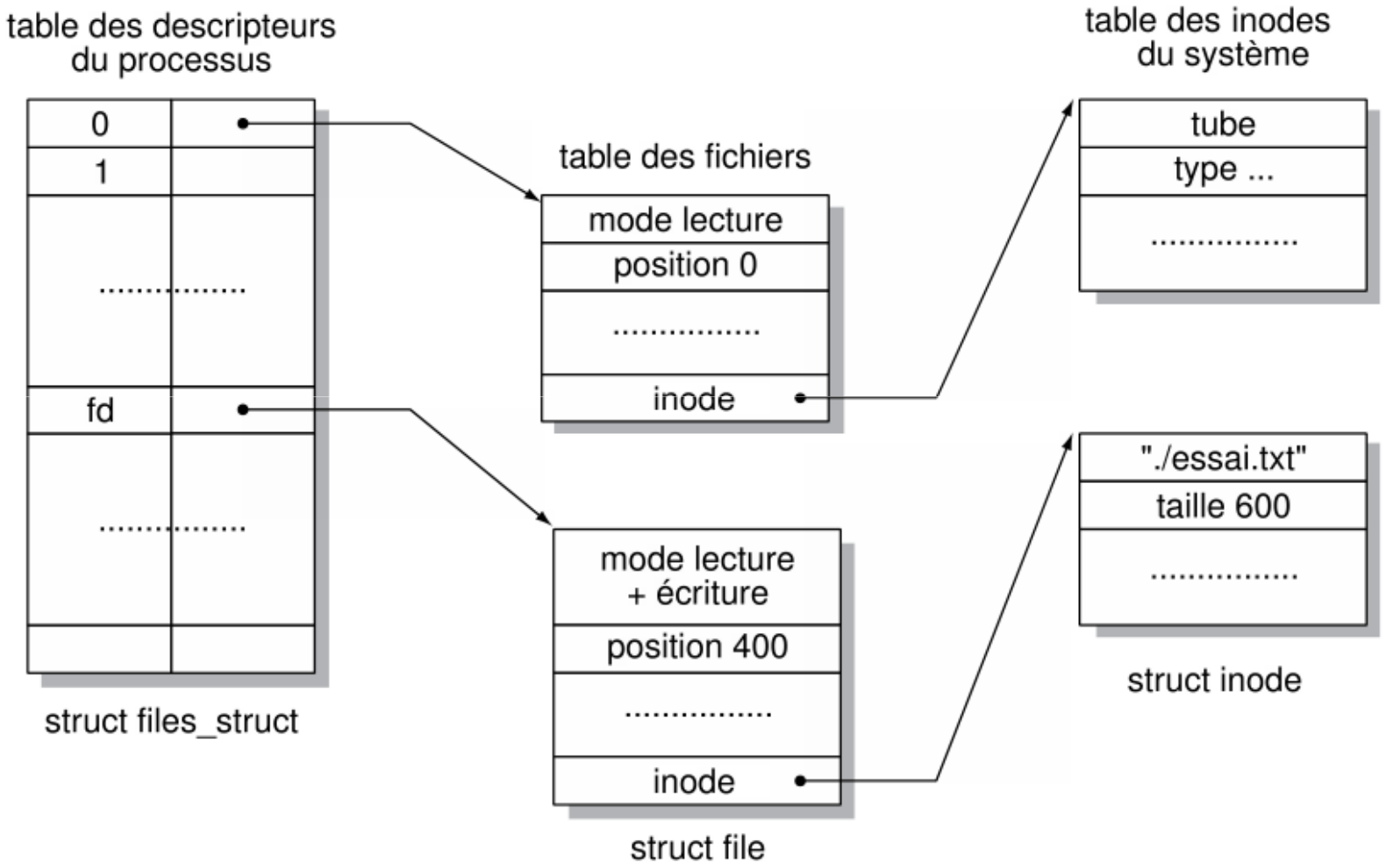
- Pour chaque processus, le noyau maintient un tableau de fichiers ouverts
- L'indice du tableau pour un fichier donné est appelé descripteur du fichier

- Les 3 premiers descripteurs de tout programme ont un usage standardisé :
  - 0 : entrée standard (clavier)
  - 1 : sortie standard (écran)
  - 2 : sortie d'erreur (écran)

- Pour tout programme, le système ouvre automatiquement ces descripteurs de fichier (donc pas nécessaire de faire appel aux fonctions type `fopen()` pour les utiliser)

- Les fonctions systèmes utilisent directement ces 3 descripteurs de fichiers (`0`, `1` et `2`, ou les constantes `STDIN_FILENO`, `STDOUT_FILENO` et `STDERR_FILENO`)
- Les librairies C et C++ suivent ce standard avec :
  - `stdin` et `std::cin`
  - `stdout` et `std::cout`
  - `stderr` et `std::cerr`

Fichiers – Structures noyau



Linux – Fonctions sur les fichiers

- La plupart des fonctions relatives aux fichiers sont disponibles en deux familles :
  - Fonctions des librairies C et C++ standards
    - En C, utilisent un pointeur sur FILE (flux ou stream)
    - En C++, utilisent des objets de la classe iostream (et dérivées)
    - La plupart de ces fonctions sont bufférisées
    - Normalisées ISO C et ISO C++
  - Fonctions systèmes
    - Appels directs aux fonctions du noyau
      - Changement de contexte user / kernel space
    - Utilisent les descripteurs de fichier
    - Non bufférisées
    - Utilisées avec d'autres ressources que des fichiers (tubes, sockets)
    - Non ISO C mais POSIX.1

- Les fonctions systèmes sur les fichiers sont prototypées dans `<unistd.h>`
- La plupart des fonctions systèmes retournent 0 si pas d'erreur et -1 en cas contraire

Fichiers – open()

- La fonction `open()` ouvre ou crée un fichier

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

- `pathname` : nom (et optionnellement chemin) du fichier
- `flags` : spécifie le mode d'accès au fichier

  `O_RDONLY` : ouvre le fichier en lecture seule

  `O_WRONLY` : ouvre le fichier en écriture seule

  `O_RDWR` : ouvre le fichier en écriture et lecture

  Les constantes suivantes sont optionnelles (avec un OU binaire) :

  `O_CREAT` : crée le fichier si inexistant (il faut alors préciser le `mode`)

  `O_TRUNC` : si le fichier existe, sa taille est ramenée à 0

  `O_EXCL` : employé avec `O_CREAT`, la fonction échoue si le fichier existe déjà

  `O_APPEND` : positionne le pointeur d'écriture à la fin du fichier

  `O_NONBLOCK` : les accès au descripteur seront non bloquants

  `O_SYNC` : les écritures seront synchronisées avec le périphérique

Fichiers – open() (suite)

`O_NOFOLLOW` : erreur si lien symbolique

- `mode` : obligatoire avec `O_CREAT`, spécifie les droits d'accès du fichier créé :
  - Peut être spécifié directement (en octal, e.g. `0640`)
  - Ou utilise un OU binaire de macros déclarées dans `<sys/stat.h>`
    - `S_IRUSR | S_IXOTH`
- Retourne le descripteur de fichier ou -1 si erreur (avec `errno`)

Fichiers – umask()

- La fonction `umask()` permet de connaitre ou changer la valeur `umask`

```
mode_t umask(mode_t mask);
```

- `mask` : nouvelle valeur de `umask`
- Retourne la valeur précédente de `umask`

- Lorsque qu'un nouveau fichier est créé avec `open()`, certains des bits de permissions peuvent être désactivés si le `umask` est différent de zéro
- Les permissions effectives sont obtenues en appliquant un ET binaire entre les permissions passées à `open()` et le complément à un de la valeur `umask` :

```
actual_perms = requested_perms & ~umask();
```

- Exemple :
  - `umask(S_IRWXO | S_IWGRP);`

Fichiers – close()

- La fonction `close()` ferme un descripteur de fichier

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

- `fd` : descripteur de fichier
- Retourne 0 ou -1 si erreur (avec `errno`)

- Il est courant de ne pas vérifier le code de retour de `close()`, mais il devrait l'être car une erreur sur une opération d'écriture précédente peut n'être signalée que lors de la fermeture du descripteur de fichier

- Un code de retour de 0 ne garantie pas que les données sont physiquement écrites sur le périphérique

Fichiers – read()

- La fonction `read()` permet de lire des données depuis un descripteur de fichier

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

- `fd` : descripteur de fichier
- `buf` : buffer recevant les octets lus
- `count` : nombre d'octets à lire
  - Si 0, la fonction retourne 0
  - Ne peut pas être plus grand que `SSIZE_MAX`
- Retourne le nombre d'octets lus ou -1 si erreur (avec `errno`)

- Le nombre d'octets lus peut être plus petit que le nombre demandé
- Quand le nombre d'octets lus est 0, la fin du fichier est atteinte

Fichiers – write()

- La fonction `write()` permet d'écrire des données dans un descripteur de fichier

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

- `fd` : descripteur de fichier
- `buf` : contenu à écrire
- `count` : nombre d'octets à écrire
  - Si 0, la fonction retourne 0
  - Ne peut pas être plus grand que `SSIZE_MAX`
- Retourne le nombre d'octets écrits ou -1 si erreur (avec `errno`)

- Le nombre d'octets écrits peut être plus petit que le nombre demandé

Fichiers – read() et write() – Coût en performances

- Les fonctions read() et write() ne sont pas bufférisées
- A chaque appel, un double changement de contexte et d'espace doit être fait
- Il est donc conseillé d'utiliser une taille de buffer adéquate, ou d'utiliser les fonctions des librairies standards qui sont bufférisées

- Exemple : copie d'un fichier de 10MB (Intel Core2 Quad @ 2.99GHz) :

	sdtio.h	read/write 1 byte	read/write 100 B	read/write 1 KB
real	0.233 s	16.281 s	0.212 s	0.045s
user	0.201 s	0.782 s	0.007 s	0.001s
sys	0.029 s	15.446 s	0.193 s	0.035s

Fichiers – open, read, write – Exemple

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <stdio.h>
#include <stdlib.h>

#define BUFFERSIZE 10     /* Size of the buffer used to read and write */

int main(void)
{
   char fsrcname[] = "fread.c";
   char fdstname[] = "/tmp/foo.bar";
   int fdsrc, fddst;
   size_t nread;
   char buf[BUFFERSIZE];

   if ((fdsrc = open(fsrcname, O_RDONLY)) < 0) {
        perror("open fdsrc");
        exit(EXIT_FAILURE);
   }
```

Fichiers – open, read, write – Exemple (suite)

```
    if ((fddst = open(fdstname, O_CREAT|O_TRUNC|O_WRONLY, S_IRUSR|S_IWUSR)) < 0)
    {
         perror("open fddst");
         exit(EXIT_FAILURE);
    }

    while ((nread = read(fdsrc, buf, BUFFERSIZE)) > 0) {
         if (write(fddst, buf, nread) < 0) {
                  perror("write fddst");
                  exit(EXIT_FAILURE);
         }
    }

    close(fdsrc);
    if (close(fddst) < 0) {
         perror("close fddst");
         exit(EXIT_FAILURE);
    }

    exit(EXIT_SUCCESS);
}
```

Fichiers – readv() et writev()

- Ces fonctions permettent la lecture et l'écriture indirecte de zone mémoire non contigües depuis ou vers des descripteurs de fichiers (scatter/gather)
- Evite les copies mémoire et factorise les appels système
- Utilise un tableau de vecteurs pointant vers les zones mémoires

```
ssize_t readv(int fd, struct iovec* vec, size_t count);
ssize_t writev(int fd, struct iovec* vec, size_t count);
```

- `fd` : descripteur de fichier
- `vec` : pointeur sur un tableau de structure du type `iovec` (`<sys/uio.h>`)

```
struct iovec {
   void * iov_base;        // adresse d'une zone mémoire
   size_t iov_len;         // taille de la zone en octets
};
```

- `count` : nombre de structures `iovec` pointées par `vec`
- Retourne le nombre total d'octets lus ou écrit, ou -1 si erreur (avec `errno`)

Fichiers – readv() et writev() – Exemple

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

int main (void)
{
   iovec vecs[3];
   vecs[0].iov_base="first";
   vecs[0].iov_len=5;
   vecs[1].iov_base=" second ";
   vecs[1].iov_len=8;
   vecs[2].iov_base="last\n";
   vecs[2].iov_len=5;

   writev(STDOUT_FILENO, vecs, 3);
}
```

Fichiers – ioctl()

- La fonction `ioctl()` sert à manipuler un driver ou périphérique à travers un fichier spécial (souvent dans `/dev`)

```
int ioctl(int fd, int cmd, ...);
```

- `fd` : descripteur de fichier
- `cmd` : code à écrire
- `...` : argument optionnel (`void *`)
- Retourne soit 0 ou un code spécifique, ou -1 si erreur (avec `errno`)

- Le code de commande est spécifique à chaque périphérique
- Macros et constantes définies dans `<sys/ioctl.h>`
- `man ioctl_list` contient une liste de la plupart des commandes

Fichiers – ioctl() – Exemple

```
#include <unistd.h>
#include <fcntl.h>
#include <sys/stat.h>
#include <sys/ioctl.h>
#include <linux/cdrom.h>
#include <sys/types.h>
#include <stdlib.h>

int main(void)
{
    int fd;

    if ((fd = open("/dev/cdrom", O_RDONLY | O_NONBLOCK)) < 0) {
          perror("open");
          exit(EXIT_FAILURE);
    }
    if ((ioctl(fd, CDROMEJECT, 0)) < 0) {
          perror("ioctl");
          exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Fichiers – lseek()

- La fonction `lseek()` permet de consulter ou déplacer la position courante dans un descripteur

```
off_t lseek(int fd, off_t offset, int whence);
```

- `fd` : descripteur de fichier
- `offset` : offset de déplacement de la position
- `whence` : spécifie comment `offset` est utilisé :
  - `SEEK_SET` : offset est une position absolue
  - `SEEK_CUR` : offset relatif à la position courante
  - `SEEK_END` : offset depuis la fin du fichier
- Retourne la position depuis le début du fichier, en nombre d'octets, ou -1 si erreur (avec `errno`)
- `off_t` est défini dans `<sys/types.h>` et correspond à un `long int`

Fichiers – lseek() – Exemples

- Consulter la position courante :

```
pos = lseek(fd, 0, SEEK_CUR);
```

- Reculer la position courante de 5 octets :

```
pos = lseek(fd, -5, SEEK_CUR);
```

- Connaitre la taille d'un fichier :

```
size = lseek(fd, 0, SEEK_END);
```

- Il est possible de changer la position courante au-delà de la fin d'un fichier :

```
lseek(fd, 100, SEEK_END);
```

  - L'espace disque n'est pas réservé !
  - Un fichier peut donc avoir des "trous" (sparse file)

Fichiers – fstat() et lstat()

- Les fonctions `fstat()` et `lstat()`, permettent d'obtenir des informations sur un descripteur de fichier (permissions, type, etc)

```
int fstat(int fd, struct stat *buf);
int lstat(char *file, struct stat *buf);
```

- `fd` : descripteur de fichier
- `buf` : structure contenant les informations
- Retourne 0 ou -1 si erreur (avec `errno`)

- `fstat()` suit les liens symboliques
- `lstat()` ne suit pas les liens symboliques

- Des macros de test (`S_ISREG`, `S_ISDIR`, `S_ISLNK`, …) sont définies dans `<sys/stat.h>`

Fichiers – struct stat

```
struct stat {
    dev_t     st_dev;     /* ID of device containing file */
    ino_t     st_ino;     /* inode number */
    mode_t    st_mode;    /* access rights */
    nlink_t   st_nlink;   /* number of hard links */
    uid_t     st_uid;     /* user ID of owner */
    gid_t     st_gid;     /* group ID of owner */
    dev_t     st_rdev;    /* device ID (if special file) */
    off_t     st_size;    /* total size, in bytes */
    blksize_t st_blksize; /* blocksize for file system I/O */
    blkcnt_t  st_blocks;  /* number of 512B blocks allocated */
    time_t    st_atime;   /* time of last access */
    time_t    st_mtime;   /* time of last modification */
    time_t    st_ctime;   /* time of last status change */
};
```

Fichiers – lstat() – Exemple

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

int is_dir(char *pathname)
{
   struct stat buf;

   if (NULL == pathname) {
        fprintf(stderr, "is_dir: NULL pointer\n");
        return -1;
   }

   if ((lstat(pathname, &buf)) < 0) {
        perror("is_dir: lstat");
        return -1;
   }

   return (S_ISDIR(buf.mode));
}
```

Fichiers – fstat() et lstat() – Exemple

```
char *file_name = "/home/joe/secret_file";
struct stat orig_st, new_st;

if (lstat(file_name, &orig_st) != 0) {
    /* erreur */
}
if (!S_ISREG(orig_st.st_mode)) {
    /* erreur : fichier non régulier ou symlink */
}
int fd = open(file_name, O_RDWR);
if (fd == -1) {
    /* erreur */
}
if (fstat(fd, &new_st) != 0) {
    /* erreur */
}
if (orig_st.st_dev != new_st.st_dev || orig_st.st_mode != new_st.st_mode ||
                                        orig_st.st_ino != new_st.st_ino) {
    /* erreur : fichier modifié */
}
if (orig_st.st_nlink > 1) {
    /* erreur : plusieurs hard links */
}
```

Fichiers – mmap()

- La fonction `mmap()` permet de charger l'ensemble ou une partie d'un fichier dans l'espace mémoire d'un programme

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

- `fd` : descripteur de fichier
- `addr` : adresse de projection ; la positionner à `NULL`
- `offset` : début de la zone du fichier à projeter
- `len` : longueur de la zone du fichier à projeter
- `prot` : droit d'accès à la mémoire projetée :
  - `PROT_NONE` : aucun accès
  - OU binaire de `PROT_READ`, `PROT_WRITE` et `PROT_EXEC`
- `flags` : `MAP_SHARED` ou `MAP_PRIVATE` : répercutions des modifications sur les projections du même fichier par d'autres programmes (voir page man)
- Retourne un pointeur sur la zone mémoire, ou `MAP_FAILED` en cas d'erreur (avec `errno`)

- La zone mémoire est libérée avec `munmap(void *addr, size_t length);`

Fichiers – mmap() – Exemple

```
/* Toutes les conditions d'erreur ne sont pas vérifiées */
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/mman.h>

int main (void)
{
   char src[] = "inputfile";
   char dst[] = "outputfile";
   struct stat stsrc, stdst;
   int fdsrc, fddst;
   char *psrc;
   int size;
```

Fichiers – mmap() – Exemple (suite)

```
    lstat(src, &stsrc);
    lstat(dst, &stdst);
    if (stsrc.st_ino == stdst.st_ino && stsrc.st_dev == stdst.st_dev) {
         fprintf(stderr, "%s and %s are the same files\n", src, dst);
         exit(EXIT_FAILURE);
    }

    fdsrc = open(src, O_RDONLY);
    fddst = open(dst, O_WRONLY | O_CREAT | O_TRUNC, 0644);
    size = stsrc.st_size;
    psrc = mmap(NULL, size, PROT_READ, MAP_PRIVATE, fdsrc, 0);
    write(fddst, psrc, size);
    munmap(psrc, size);
    close(fdsrc);
    close(fddst);

    exit(EXIT_SUCCESS);
}
```

Fichiers – dup() et dup2()

- Les fonctions `dup()` et `dup2()` servent à dupliquer un descripteur de fichier
- Les descripteurs de fichiers dupliqués partagent les mêmes attributs (mode d'ouverture, droits d'accès, position)
- Permet par exemple de rediriger les E/S standards vers d'autres descripteurs

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

- `oldfd` : descripteur à dupliquer
- Retourne le nouveau descripteur de fichier, ou -1 si erreur (avec `errno`)
- Le descripteur retourné est le plus petit disponible

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

- `oldfd` : descripteur à dupliquer
- `newfd` : nouveau descripteur
- Retourne le nouveau descripteur de fichier, ou -1 si erreur (avec `errno`)
- Ferme `newfd` si ce descripteur était ouvert, puis copie `oldfd` dans `newfd`
- L'opération est atomique

Fichiers – dup() – Exemple

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

int main(void)
{
    int fd;
    if ((fd = open("foo.dup", O_RDWR | O_CREAT | O_TRUNC, 0644)) < 0) {
          perror("open");
          exit(EXIT_FAILURE);
    }
    close(STDOUT_FILENO);
    if (dup(fd) < 0) {
          perror("dup");
          exit(EXIT_FAILURE);
    }
    close(fd);
    execlp("ls", "ls", NULL);
    perror("execlp");
    exit(EXIT_FAILURE);
}
```

Fichiers – fcntl()

- La fonction `fcntl()` permet toute une série d'opérations sur un descripteur

```
int fcntl(int fd, int cmd, ...);
```

- `fd` : descripteur de fichier
- `cmd` : opération à effectuer (exemple ci-après)
- `...` : arguments optionnels à `cmd`
- Retourne 0 ou -1 si erreur (avec `errno`)

Fichiers – fcntl() – Verrous

- La fonction `fcntl()` permet de verrouiller une partie d'une fichier en lecture ou écriture
- L'opération `cmd` est alors l'un
  - `F_SETLK` : verrouille ou déverrouille une partie du fichier
  - `F_GETLK` : demande l'état de verrouillage d'une partie du fichier
  - `F_SETLKW` : comme F_SETLK, mais bloquant
- L'argument de `cmd` est une structure `flock` comprenant :
  - `l_type` : `F_RDLCK`, `F_WRLCK`, `F_UNCLK`
  - `l_whence` : `SEEK_SET`, `SEEK_CUR`, `SEEK_END` (comme `lseek()`)
  - `l_start` : offset de départ
  - `l_end` : nombre d'octets
  - `l_pid` : PID du processus détenant le verrou

- Ce verrou est coopératif ; pas strictement imposé par le noyau

Fichiers –fdopen() et fileno()

- Les fonctions fdopen() et fileno() permettent de convertir un descripteur de fichier en un pointeur sur FILE et vice-versa

```
FILE *fdopen(int fd, const char *mode);
```

- `fd` : descripteur de fichier
- `mode` : mode d'ouverture, comme pour fopen()
- Retourne un pointeur sur FILE ou `NULL` si erreur (avec `errno`)

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

- `stream` : pointeur sur FILE
- Retourne un descripteur de fichier ou -1 si erreur (avec `errno`)

- Il est fortement déconseillé de mélanger des fonctions stream avec des fonctions systèmes !

Fichiers – fopen() en mode exclusif

- Un inconvénient de la fonction `fopen()` est qu'il n'est pas possible de savoir si un fichier ouvert en écriture existait déjà

- La version `fopen()` de glibc dispose d'un mode exclusif mais ceci n'est pas standard et donc non portable :

```
FILE *fp = fopen(filename, "wx");
```

- Une solution est d'utiliser la fonction système `open()` puis `fdopen()` pour associer le descripteur de fichier à un stream :

```
FILE *fp;
int fd;
if ((fd = open(filename, O_CREAT|O_EXCL|O_WRONLY, 0644)) == -1) {
  /* handle error */
}
if ((fp = fdopen(fd, "w")) == NULL) {
  /* handle error */
}
```

Fichiers – Autres fonctions

- `getcwd()` : donne le répertoire courant
- `chdir()` : change le répertoire courant
- `mkdir()` : crée un répertoire
- `chroot()` : change le répertoire racine du processus courant
- `rename()` : déplace ou renomme un fichier
- `access()` : vérifie l'existence d'un fichier
- `unlink()` : efface un fichier
- `truncate()` et `ftruncate()` : réduisent la taille d'un fichier
- `chmod()` et `fchmod()` : changent les droits d'accès
- `chown()` et `fchown()` : changent le propriétaire et le groupe
- `opendir()`, `readdir()`, `closedir()` : permettent de connaitre le contenu d'un répertoire
- `ftw()`, `nftw()` : permettent de parcourir une arborescence

Processus

Processus – Introduction

- Les processus sont la base du multitâche sous Linux
- Un processus est une instance d'exécution d'un programme
  - Plusieurs exécutions de programmes
  - Plusieurs exécutions d'un même programme
  - Plusieurs exécutions "simultanées" de programmes
  - Plusieurs exécutions "simultanées" d'un même programme

- Un processus est composé de :
  - Environnement processeur (registres)
  - Code programme exécuté
  - Variables et pile mémoire
  - Descripteurs de fichiers ouverts
  - Variables d'environnement
  - Un ou plusieurs threads

- Typiquement, code et librairies systèmes sont partagés entre processus

Processus – Introduction (suite)

- Chaque processus est identifié par un identificateur de programme (PID)
- Chaque processus connait aussi le PID de son processus parent (PPID)
  - PID 1 est lancé par le noyau au boot (typiquement `init`)

- Chaque processus s'exécute sous l'identité d'un utilisateur (UID) et d'un groupe (GID)
  - UID et GID réels : utilisateur ayant lancé le programme
  - UID et GID effectifs : privilèges (droits) accordés au processus

- Chaque processus a un répertoire de travail (origine de l'interprétation des chemins relatifs)

- Chaque processus a sa date et heure de création (en secondes depuis l'*epoch*)

- Le noyau garde les temps CPU consommés par un processus (en mode utilisateur et en mode noyau)
- Le noyau utilise d'autres structures internes pour chaque processus (table des descripteurs de fichiers ouverts par exemple)

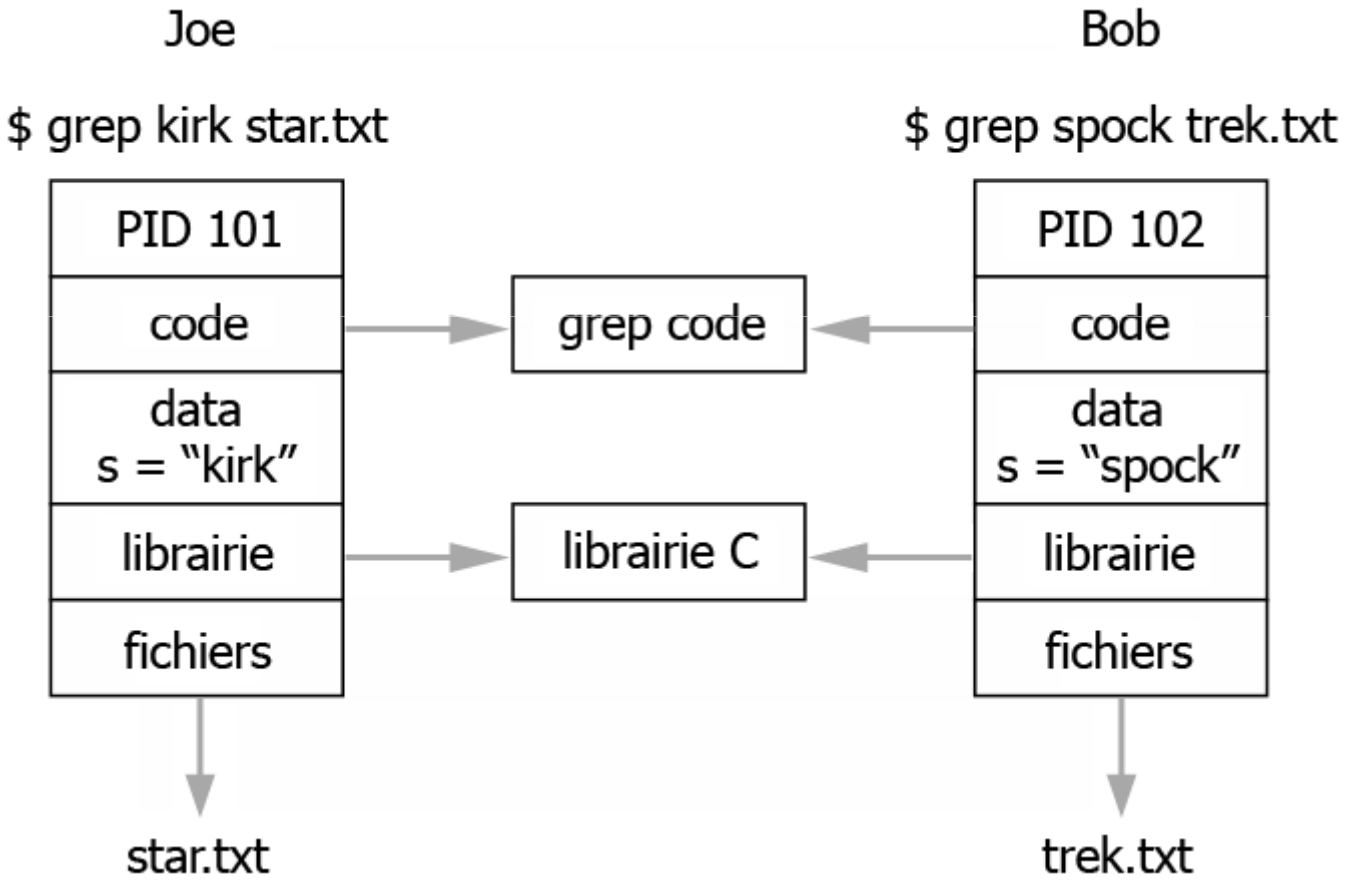
Processus – Groupes et sessions

- Le système partitionne les processus en groupes de processus :
  - Groupes de processus != groupes d'utilisateurs
  - Un processus appartient à un (seul) groupe
  - Ces groupes permettent l'envoi de signaux à un ensemble de processus
  - Un groupe est identifié par le PID du processus leader
  - Les groupes servent surtout aux shells, il est rare de les utiliser dans des applications

- Le système partitionne les processus en sessions
  - Un processus appartient à une (seule) session
  - Par défaut, la session d'un processus est celle de son père
  - Un processus qui crée une session est le leader de cette session
  - Une session est identifiée par le PID du processus leader
  - Une session est typiquement liée à un (pseudo)-terminal
  - Dans une session, un groupe de processus est en avant-plan (reçoit les données du clavier et affiche sur le terminal)
  - Les autres groupes sont en arrière plan (ne peuvent accéder au terminal)

Processus – Structure

- Le code programme est partagé en mémoire (en lecture seule)
- Les librairies peuvent aussi être partagées en mémoire



Processus – Structure (suite)

- La commande `ps` permet de voir les processus et leur statuts

```
$ ps –ef
UID   PID   PPID   C   STIME   TTY    TIME       CMD
Joe   101   96     0   18:24   tty2   00:00:00   grep kirk star.txt
Bob   102   92     0   18:24   tty4   00:00:00   grep spock trek.txt
```

- La commande `pstree` permet de voir les processus sous forme d'arborescence

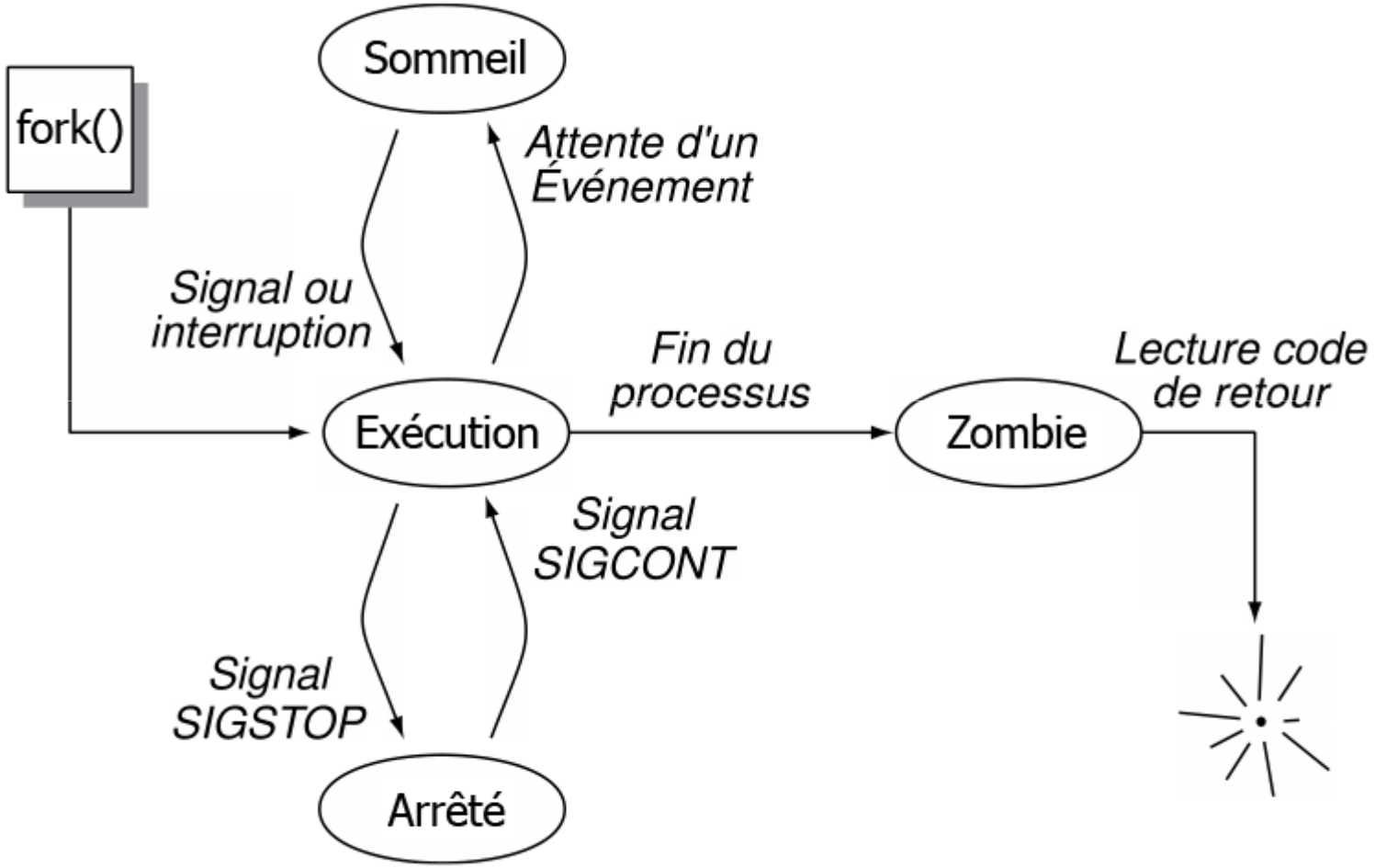
- La commande `top` permet de voir une liste dynamique et ordonnée des processus

Processus – Etats

- Un processus est à un moment donné dans un état particulier : actif, suspendu, etc.
- La commande `ps` permet de voir les processus et leurs états

Etat	Description
S	En sommeil (attente d'un événement extérieur)
R	En cours d'exécution (actif)
D	En sommeil in-interruptible
T	Stoppé (par SIGSTOP)
Z	Zombie (le processus parent n'a pas encore lu le code retour)
N	Processus basse priorité
W	Paging (pas sous Linux 2.6)
s	Session leader
+	Processus foreground (pas en attente d'E/S ou de proc. fils)
l	Processus multithreadé
<	Processus haute priorité

Processus – Etats – Exemple



Processus – Attributs

- Principaux attributs d'un processus :
  - Identificateur de processus
    ```
    pid_t = getpid(void);
    ```
  - Identificateur du processus parent
    ```
    pid_t = getppid(void);
    ```
  - Identificateur de l'utilisateur propriétaire réel
    ```
    uid_t = getuid(void);
    ```
  - Identificateur de l'utilisateur propriétaire effectif
    ```
    uid_t = geteuid(void);
    ```
  - Identificateur du groupe propriétaire réel
    ```
    gid_t = getgid(void);
    ```
  - Identificateur du groupe propriétaire effectif
    ```
    gid_t = getegid(void);
    ```
  - Répertoire de travail
    ```
    char *getcwd(char *buf, size_t size);
    ```
  - Groupe du processus
    ```
    pid_t getpgid (pid_t pid);
    int setpgid (pid_t pid, pid_t pgid);
    ```

Processus – Attributs (suite)

- Temps d'exécution

```
clock_t times(struct tms *buf);
```

- Ressources utilisées

```
int getrusage(int who; struct rusage *usage);
```

- Groupe du processus et identificateur de session
- Variables d'environnement
- Limites de ressources
- Descripteurs de fichiers ouverts
- Segments de mémoire partagée

Processus – Attributs (suite)

- A chaque processus correspond un répertoire nommé par son PID dans `/proc`, contenant les paramètres du processus :
  - `/proc/<pid>/status`
  - `/proc/<pid>/limits`
  - `/proc/<pid>/...`
  - Voir `man 5 proc`
- La commande `lsof` montre toutes les ressources utilisées :
  - `lsof | grep <pid>`
- La commande `pmap` montre les ressources mémoire d'un processus :
  - `pmap -x <pid>`

Processus – system()

- La fonction `system()` permet de lancer un processus depuis un autre, à travers un shell

```
int system(const char *string);
```

- `string` : commande à exécuter
- Retourne le code de sortie de la commande, 127 si un shell ne peut être lancé, ou -1 si erreur

- Fait appel à `/bin/sh –c string`
- Ne retourne qu'une fois le nouveau processus terminé
- Exemples :

```
#include <stdlib.h>
system("ps ax");
system("ps ax &");
```

- Ne pas utiliser `system()` si pas besoin d'interpréter la commande, préférer `exec..()`

Processus – exec*()

- Les fonctions de la famille `exec*()` permettent de remplacer un processus par un autre

- Deux familles :
  - `execl` : nombre variable d'arguments, finissant pas `NULL`
  - `execv` : second argument est un tableau de chaines

- Le nouveau processus reçoit ces arguments dans `argv`
- Les fonctions finissant par `p` cherche l'exécutable dans `PATH`
- La variable globale `environ` permet de passer une nouvelle variable dans l'environnement du nouveau processus
- Les fonctions `execle` et `execve` permettent de passer un tableau de chaines comme nouvel environnement du nouveau processus
- Ces fonctions ne retournent normalement pas, sauf -1 en cas d'erreur (avec `errno`)
- Le processus fils hérite du PID du père et des descripteurs de fichiers ouverts

Processus – exec…() – Exemples

```
#include <unistd.h>

char *const ps_argv[] = {"ps", "ax", 0};
char *const ps_envp[] = {"PATH=/bin:/usr/bin", "TERM=console", 0};

execl("/bin/ps", "ps", "ax", 0);
execlp("ps", "ps", "ax", 0);
execle("/bin/ps", "ps", "ax", 0, ps_envp);

execv("/bin/ps", ps_argv);
execvp("ps", ps_argv);
execve("/bin/ps", ps_argv, ps_envp);
```

Processus – fork()

- La fonction `fork()` permet de lancer un processus depuis un autre, en "dupliquant" le processus courant

```
pid_t fork();
```

- Retourne le PID du processus fils créé au parent, et 0 au processus fils, ou -1 si erreur (avec `errno`)
- Le processus fils exécute le même code que le processus parent
- Le processus fils hérite :
  - De l'espace d'adressage du processus parent
  - Des descripteurs de fichiers ouverts
- Tous les attributs du processus parent sont hérités, sauf :
  - Le PID et le PPID (le PPID du fils est le PID du parent)
  - Signaux, sémaphores, verrous sur fichiers ou mémoire, timers
  - Les compteurs de ressources (remis à zéro)
- Il est impossible de prédire l'ordre d'exécution entre le parent et le fils

Processus – fork() – Exemple

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

int main(void)
{
   pid_t child;

   if ((child = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
   } else if (child == 0) {
        printf("in child\n");
        printf("child pid = %d, ppid = %d\n", getpid(), getppid());
        exit(EXIT_SUCCESS);
   } else {
        printf("in parent\n");
        printf("parent pid = %d, ppid = %d\n", getpid(), getppid());
   }
   exit(EXIT_SUCCESS);
}
```

Processus – Terminaison d'un processus

- Un processus se termine lorsque :
  - Sa fonction `main()` appelle `return()`
  - Il appelle `exit()`
  - Il appelle `_exit()`
  - Il appelle `abort()`
  - Il est terminé par un signal

Processus – _exit()

- La fonction `_exit()` termine le processus immédiatement

```
void _exit(int status);
```

- `status` : code retourné au processus parent

- Les descripteurs ouverts sont fermés
- Les buffers d'entrées / sorties ne sont pas forcément flushés
- Les processus fils héritent du parent PID 1
- Le processus parent reçoit un signal SIGCHLD
- Les fonctions enregistrées avec `atexit()` ou `on_exit()` ne sont pas exécutées

- La fonction de la librairie standard `exit()` appelle `_exit()` après avoir fait certaines "taches ménagères" comme le flush des buffers, l'appel des fonctions enregistrées avec `atexit()` ou `on_exit()`, etc.

Processus – abort()

- La fonction `abort()` termine le processus immédiatement

```
void abort(void);
```

- Débloque le signal `SIGABRT`
- Les descripteurs ouverts sont fermés
- Les buffers d'entrées / sorties sont flushés
- Crée un fichier core

Processus – Zombis

- Lorsqu'un processus se termine, son code de retour est renvoyé au processus parent

- Si un processus fils termine avant que le parent ne récupère son code de retour, il devient un zombi
  - Un processus zombi reste dans la table des processus du noyau mais ses ressources ont étés libérées
  - Le noyau garde alors le code retour du fils jusqu'à ce qu'il soit récupéré par le parent

- Si un processus parent termine avant un fils sans avoir lu code de retour, le fils devient orphelin
  - Le processus de PID 1 (`init`) devient le parent du processus orphelin
  - Lorsque le processus orphelin se termine, le processus de PID 1 lit son code de retour, prévenant ainsi un zombie

Processus – wait()

- La fonction `wait()` récupère le code retour d'un processus fils terminé

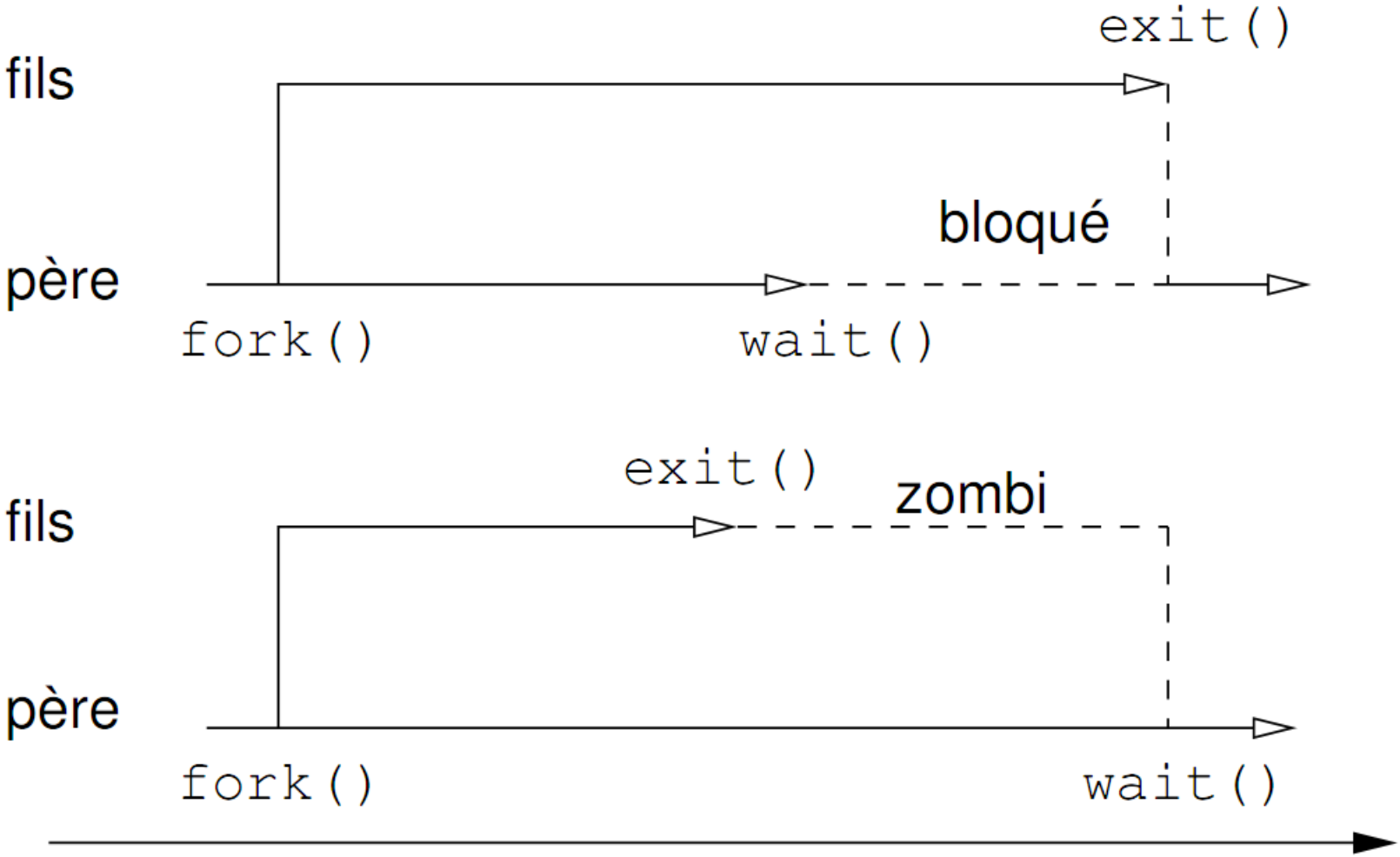
```
pid_t wait(int *status);
```

- `status` : code de retour du processus fils
- Retourne le PID du processus fils terminé, ou -1 si erreur (avec `errno`)

- L'exécution du processus utilisant `wait()` est suspendue jusqu'à ce que l'un des processus fils termine

- L'entier `status` peut être examiné avec des macros :
  - `WIFEXITED(status)` : le processus fils a terminé normalement
  - `WEXITSTATUS(status)` : retourne les 8 bits de poids faible de la valeur passée à `exit()`, `_exit()` ou `return()`
  - `WIFSIGNALED(status)` : le processus fils a été terminé par un signal
  - `WTERMSIG(status)` : retourne le numéro du signal
  - Voir aussi `man 2 wait`

Processus – Zombis (suite)



FORMATION

EUROGICIEL GROUPE

Processus – waitpid()

- La fonction `waitpid()` récupère le code retour d'un ou plusieurs processus fils

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

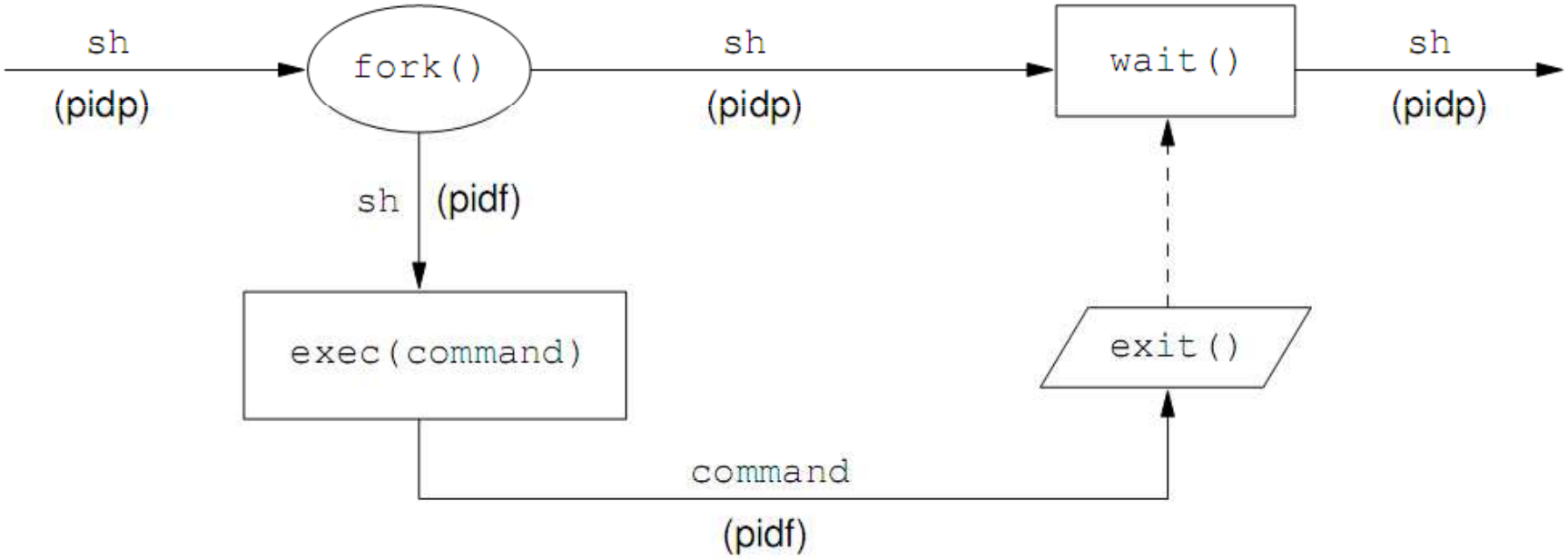
- `pid` :
  - `< -1` : attendre n'importe quel processus fils dont le PGID est la valeur absolue de `pid`
  - `-1` : attendre n'importe quel processus fils
  - `0` : attendre n'importe quel processus fils dont le PGID est celui du parent
  - `> 0` : attendre le processus fils dont le PID est `pid`
- `status` : code de retour du processus fils
- `option` :
  - `0` : bloque en attendant la fin d'un processus fils
  - `WNOHANG` : ne bloque pas si aucun processus fils n'est terminé
- Retourne le PID du processus fils terminé, ou -1 si erreur (avec `errno`)
- L'entier `status` peut être examiné avec les mêmes macros que `wait()`
- `wait(&status);` est équivalent à `waitpid(-1, &status, 0);`

Processus – waitpid() - Exemple

```
#include <unistd.h>
#include <sys/wait.h>
#include <stdio.h>
#include <stdlib.h>

int main(void)
{
   pid_t child;
   int status;

   if ((child = fork()) == -1) {
        perror("fork");
        exit(EXIT_FAILURE);
   } else if (child == 0) {
        printf("in child\n");
        exit(EXIT_SUCCESS);
   } else {
        waitpid(child, &status, 0);
        printf("child exited with status %d\n", WEXITSTATUS(status));
   }
   exit(EXIT_SUCCESS);
}
```

Processus – Complémentarité fork() et exec*()

- Les fonctions `exec*()` ne retournent pas
- Pour faire exécuter un programme par un processus, on fait d'abord un `fork()` puis un `exec*()` dans le processus fils
- Une erreur dans le programme n'affecte pas le processus parent
- C'est typiquement ce que fait un shell au lancement d'un programme



Processus – Daemons

- Un démon (daemon) est un programme qui fonctionne en permanence, et est non interactif (pas de saisie clavier, pas d'affichage, donc pas de terminal associé au programme)
- Il effectue généralement une tache système (ex. syslogd) ou propose un service (ex. http)

- C'est un processus détaché de tout terminal, lancé en arrière plan (avec `setsid()`) qui permet d'obtenir un nouveau GID et une nouvelle session)
  - Evite de recevoir un signal destiné à un groupe de processus tel que `SIGHUP` à la fermeture d'un terminal
- En général, il est lancé par root, et souvent, pour des raisons de sécurité, le processus change d'UID et de GID (utilise un utilisateur sans aucun droit, dont le shell est `/bin/false` par exemple), change de répertoire courant, ferme les descripteurs non utilisés, et/ou tourne dans un "chroot jail"

Processus – Daemons – Exemple

```
if ((pid = fork()) < 0) {  /* Error */  }
if (pid > 0) {
   /* Parent process */
   exit(EXIT_SUCCESS);
}

/* Child process */
if (setsid() == -1) {  /* Error */  }
if (chdir("/") == -1) {  /* Error */  }
umask(0);
if (freopen("/dev/null", "r", stdin) != stdin || \
        freopen("/dev/null", "w", stdout) != stdout || \
        freopen("/dev/null", "w", stderr) != stderr) {
        /* Error */
}
close(STDIN_FILENO);
close(STDOUT_FILENO);
close(STDERR_FILENO);
```

Processus – Priorité d'un processus

- La priorité d'un processus peut être changée par la valeur de `nice` ou `niceness`
- La fonction `nice()` permet de changer la priorité d'un processus

```
int nice(int nice);
```

- `nice` : valeur à ajouter à la valeur de `nice` courante du processus
- Retourne la nouvelle valeur de `nice` (glibc ≥ 2.2.4) (avec `errno`)

- Normalement, un processus a une valeur de `nice` de 0
- Si la valeur de `nice` augmente, la priorité diminue, et vice-versa
- Seul l'utilisateur avec UID 0 (root) peut changer `nice` avec une valeur négative

Processus – Politiques d'ordonnancement

- Le noyau Linux supporte plusieurs politiques d'ordonnancement :
  - FIFO :
    - Changement de contexte quand le processus rend la main ou termine
    - Préemption par processus plus prioritaire
      - Le processus préempté est de nouveau exécuté dès que le processus plus prioritaire est bloqué
  - Round-Robin :
    - Processeur alloué successivement à chacun des processus
    - Expiration du quantum : réquisition du processeur
    - Processus placé en queue de la file d'attente
    - Election des processus de plus forte priorité
  - Other : l'ordonnancement Unix (Posix) traditionnel
    - Quantum, à priorité dynamique
    - Priorité de base (statique) + priorité dynamique (valeur de nice)
    - Principe d'extinction des priorités
  - Batch : nouveau depuis 2.6.16 : comme Other, mais permet de défavoriser un processus en tache de fond et non interactif

- Ordonnancement des processus gérés par les trois politiques :
  - 1. Processus associé à une FIFO
  - 2. Processus associé à un tourniquet
  - 3. Processus associé à la politique Other ou Batch

Processus – sched_get_priority_*()

- Les fonctions relatives aux politiques d'ordonnancement et à leurs paramètres sont définies dans `<sched.h>`

- Les fonctions sched_get_priority_max() et sched_get_priority_min() permettent de connaitre l'échelle des priorités pour une politique donnée

```
int sched_get_priority_max(int policy);
int sched_get_priority_min(int policy);
```

- `policy` : `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER` ou `SCHED_BATCH`
- Retourne la priorité minimale ou maximale pour la politique spécifiée, ou -1 si erreur (avec `errno`)

- Couramment, les politiques FIFO et RR ont une échelle de priorité de 0 à 99.
- Les politiques OTHER et BATCH n'ont qu'un seul niveau de priorité : 0

Processus – sched_*scheduler()

- La fonction `sched_setscheduler()` permet de sélectionner une politique et un niveau de priorité

```
int sched_setscheduler(pid_t pid, int policy, const struct
                                        sched_param *param);
```

- `pid` : processus (si 0, processus courant)
- `policy` : `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER` ou `SCHED_BATCH`
- `param` : structure :

```
struct sched_param {
    int sched_priority;
}
```

Priorité de 0 à 99 pour FIFO et RR

- Retourne 0 ou -1 si erreur (avec `errno`)

```
int sched_getscheduler(pid_t pid);
```

- `pid` : processus (si 0, processus courant)
- Retourne `SCHED_FIFO`, `SCHED_RR`, `SCHED_OTHER` ou `SCHED_BATCH`

Processus – sched_yield() et sched_rr_get_interval()

- La fonction `sched_yield()` permet de volontairement rendre la main

```
int sched_yield(void);
```

- Retourne 0 ou -1 si erreur (avec `errno`)

- La fonction `sched_rr_get_interval()` permet de connaitre le quantum de temps Round-Robin alloué à un processus

```
int sched_rr_get_interval(pid_t pid, struct timespec * tp);
```

- `pid` : PID du processus (si 0, processus appelant)
- `tp` : pointeur sur structure timespec :

```
struct timespec {
   time_t tv_sec;          Secondes
   long   tv_nsec;         Nanosecondes
};
```

- Retourne 0 ou -1 si erreur (avec `errno`)

Signaux

Signaux - Introduction

- Les signaux sont des "interruptions logicielles"
- Informent un processus d'un événement de façon asynchrone
- Emis par un processus ou par le système
- Les programmes non triviaux se doivent de les gérer
- Chaque signal a un nom : `SIGxxx`
- Forme de communication bas-niveau
  - Préférable d'avoir un accord entre l'émetteur et le récepteur sur la sémantique de l'événement

- Exemples :
  - Ctrl-C génère `SIGINT`
  - Le noyau répercute une exception hardware par un signal, par exemple `SIGSEV` pour une référence mémoire invalide
  - Certaines conditions logicielles envoient un signal, par exemple `SIGPIPE` ou `SIGURG`
  - La commande `kill` et la fonction `kill()` permettent d'envoyer un signal à un autre processus

Signaux – Introduction (suite)

- Le noyau gère l'envoi des signaux vers le processus
- Chaque signal engendre une action par défaut
  - Cette action est paramétrable pour certains signaux

- Un processus peut notifier à l'avance le noyau d'une action à suivre :
  - Ignorer le signal (rien ne se passe)
  - Appel d'une fonction du processus (callback) de façon asynchrone (intercepter du signal)
    - L'exécution normale reprend à la terminaison de la fonction

- `SIGKILL` et `SIGSTOP` ne peuvent ni ignorés ni interceptés

- Si un signal intercepté est envoyé plusieurs fois, un seul signal est délivré

- Les fonctions sur les signaux sont prototypées depuis `<signal.h>`

Signaux – Principaux signaux

Numéro	Nom	Description	Action par défaut
1	SIGHUP	Terminal déconnecté - Fin de session	Terminaison
2	SIGINT	Interruption Ctrl-C	Terminaison
3	SIGQUIT	Interruption Ctrl-\	Terminaison + Core
4	SIGILL	Instruction illégale	Terminaison + Core
9	SIGKILL	Tue le processus (arrêt brutal) – immuable	Terminaison
10	SIGUSR1	Pas de signification particulière, peut être utilisé de manière différente par chaque programme.	Terminaison
11	SIGSEGV	Violation de mémoire	Terminaison + Core
12	SIGUSR2	Comme SIGUSR1	Terminaison
14	SIGALARM	Timer de la fonction alarm()	Terminaison
15	SIGTERM	Termine le processus	Terminaison
17	SIGCHLD	Envoyé à un processus dont un fils est arrêté ou terminé.	Signal ignoré
18	SIGCONT	Reprendre l'exécution suite à un SIGSTOP	Fin de suspension
19	SIGSTOP	Suspend l'exécution – immuable	Suspension

Signaux – signal()

- La fonction `signal()` permet d'ignorer ou intercepter un signal

```
sighandler_t (*signal(int sig, void (*func)(int)))(int);
```

- `sig` : signal à traiter
- `func` : pointeur sur fonction ou :
  - `SIG_IGN` : ignore le signal
  - `SIG_DFL` : reset l'action par défaut
- Retourne la valeur précédente de `func` ou `SIG_ERR` si erreur (avec `errno`)
- L'action du signal n'est pas remise à sa valeur par défaut une fois le signal intercepté (c'est le cas sous d'autres Unix)

- Les signaux ne sont pas bloqués durant l'exécution du callback
  - Fonction désuète, préférer `sigaction()`

Signaux – signal() - Exemple

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

void handler(int sig)
{
   printf("Signal %d reçu\n", sig);
   (void) signal(SIGINT, SIG_DFL);
}

int main(void)
{
   (void) signal(SIGINT, handler);
   while (1) {
         printf("Hello world\n");
         sleep(1);
   }
}
```

Signaux – sig*set()

- Les fonctions `sig*set()` sont utilisées pour créer des masques de signaux à traiter

```
int sigaddset(sigset_t *set, ing sig);
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigdelset(sigset_t *set, int sig);
```

- Retournent 0 ou -1 si erreur (avec `errno`)

```
int sigismember(const sigset_t *set, int sig);
```

- Retourne 1 (true) ou 0 (false), ou -1 si erreur

Signaux – sigaction()

- La fonction `sigaction()` est similaire à `signal()`, mais bloque les signaux durant l'exécution du callback

```
int sigaction(int sig, struct sigaction *act, struct
                                          sigaction *oact);
```

- `sig` : signal à traiter
- `act` : pointeur sur structure `sigaction`

```
struct sigaction {
   void(*)(int)sa_handler;  pointeur sur fonction, SIG_DFL ou SIG_IGN
   sigset_t sa_mask;        signaux à bloquer dans sa_handler
   int sa_flags;            comportement : SA_ONESHOT, SA_RESTART
}
```

- `oact` : pointeur sur structure `sigaction` précedente

- Retourne 0 ou -1 si erreur (avec `errno`)

- Voir `man 2 sigaction`

Signaux – sigaction() - Exemple

```
void handler(int sig)
{
   printf("Signal %d reçu\n", sig);
}

int main(void)
{
   struct sigaction act;

   act.sa_handler = handler;
   sigemptyset(&act.sa_mask);
   act.sa_flags = SA_ONESHOT;
   sigaction(SIGINT, &act, 0);

   while (1) {
        printf("Hello world\n");
        sleep(1);
   }
}
```

Signaux – sigaction() - sig_atomic_t

- La fonction callback est appelée de façon asynchrone
- Si cette fonction modifie une ou des variables globale, il est préférable d'utiliser le type `sig_atomic_t`

- Exemple :

```
volatile sig_atomic_t keep_going = 1;

void handler(int sig)
{
   keep_going = 0;
   signal(sig, handler);
}

int main (void)
{
   signal(SIGINT, handler);
   while (keep_going) {
     /* do stuff */
   }
}
```

Signaux – sigprocmask()

- La fonction `sigprocmask()` permet de récupérer ou de changer le masque de signaux du processus appelant

```
int sigprocmask(int how, sigset_t *set, sigset_t *oldset);
```

- `how` : `SIG_BLOCK`, `SIG_UNBLOCK`, `SIG_SETMASK`
- `set` : selon la valeur de `how` :
  - masque de signaux à bloquer (en plus des signaux déjà bloqués)
  - masque de signaux à débloquer
  - masque de signaux à bloquer
- `oldset` : contient le masque précédent si non `NULL`
- Retourne 0 ou -1 si erreur (avec `errno`)

- Exemple :

```
sigset_t set;
sigemptyset(&set);
sigaddset(&set, SIGTERM);
sigprocmask(SIG_BLOCK, set, NULL)
```

Signaux – kill()

- La fonction `kill()` permet d'envoyer un signal à un autre processus

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

- `pid` :
  - > 0 : PID du processus cible
  - 0 : signal envoyé à tout les processus de même groupe que l'émetteur
  - -1 : signal envoyé à tout les processus pour lesquels l'émetteur en a le droit
  - < -1 : signal envoyé à tout les processus dont le groupe est la valeur absolue de `pid`
- `sig` : signal à envoyer
- Retourne 0 ou -1 si erreur (avec `errno`)

- Pour qu'un processus est le droit d'envoyer un signal, son UID doit être 0 (root) ou son UID doit être égal à celui du processus receveur

Signaux – alarm()

- Envoi le signal `SIGALRM` au processus appelant après un délai

```
unsigned int alarm(unsigned int seconds);
```

- `seconds` : nombre de secondes avant l'avant du signal
- Retourne le nombre de secondes restantes d'un armement précédant ou 0 si il n'y en avait pas

- Une seule alarme par processus
- Tout nouvel armement annule le précédent
- Un délai nul supprime l'armement en cours

- D'autres fonctions comme `sleep()` ou `setitimer()` peuvent être implémentée avec `SIGALRM`, mélanger des appels à ces fonctions est donc prohibé.

Signaux – setitimer()

- Permet d'utiliser un des trois timers que le système met à disposition de tout processus

```
int setitimer(int which, struct itimerval *new, struct
                                                itimerval *old);
```

- `which` : spécifie un des trois timers :
  - `ITIMER_REAL` : décrémente en temps réél, délivre `SIGALRM`
  - `ITIMER_VIRTUAL` : décrémente seulement lorsque le processus s'exécute (en mode user), délivre `SIGVTALRM`
  - `ITIMER_PROF` : décrémente les deux autres timers, délivre `SIGPROF`
- `new` et `old` : pointeurs sur structure itimerval :

```
struct itermerval {
   struct timeval it_interval;      Intervalle entre échéances
   struct timeval it_value;         Spécifie la première échéance
};
struct timeval {
   long tv_sec;
   long tv_user;
};
```

- Retourne 0 ou -1 si erreur (avec `errno`)

Signaux – SIGCHLD

- `SIGCHLD` est automatiquement envoyé à un processus parent dont un fils termine, ou est stoppé par réception d'un `SIGSTOP` ou `SIGTSTP`
- Par défaut, `SIGCHLD` est ignoré
- Peut donc être utilisé par un processus parent afin de ne pas bloquer ou continuellement appeler `wait()` ou `waitpid()`

Signaux – Signaux temps-réel

- Les signaux temps-réel n'ont pas de signification prédéfinie
- Ils sont réservés aux utilisateurs (comme `SIGUSR1` et `SIGUSR2`)
- L'action par défaut est de terminer le processus récepteur
- Linux supporte 32 signaux temps-réel numéroté de 33 (`SIGRTMIN`) à 64 (`SIGRTMAX`) – (utiliser la notation `SIGRTMIN + n`)

- Les signaux temps-réel bloqués sont empilés
- Les signaux temps-réel empilés sont délivrés dans un ordre précis :
  - Si de même numéro, ils sont délivrés dans l'ordre d'émission
  - Si de numéros différents, ils sont délivrés en commençant par le signal de numéro le moins élevé

- Si des signaux classiques et des signaux temps-réel sont en même temps en attente pour un processus, Linux donne priorité aux signaux classiques

Tubes

Tubes – Introduction

- Un tube (ou pipe) est un moyen de communication entre processus (locaux)
- Un processus écrit des données dans le tube, un autre processus les lit

- Très courant dans un shell pour envoyer la sortie d'une commande vers l'entrée d'une autre

```
cat file | more
```

- Il existe deux types de tubes :
  - Anonymes : gérés par le noyau, en mémoire
  - Nommés : existe dans le système de fichier

Tubes – Introduction (suite)

- Les tubes sont unidirectionnels (half-duplex : une entrée et une sortie)
- Se comportent comme une FIFO

- Un tube anonyme ne peut être utilisé que par des processus relatifs (ancêtre commun)

- Les tubes sont bufférisés, mais :
  - Si le buffer d'écriture est plein, un processus écrivant dans le tube sera bloqué
  - Si le buffer de lecture est vide, un processus lisant le tube sera bloqué

- La taille maximale des buffers des tubes est `PIPE_BUF`, déclaré dans `<limits.h>`
  - Ecriture atomique si < `PIPE_BUF`
  - Sinon découpage par le système en plusieurs écritures (non-portable)
  - Bloquant tant qu'il n'y a pas la place pour écrire la quantité demandée

- Les fonctions relatives aux tubes sont déclarées dans `<unistd.h>`

Tubes – pipe()

- La fonction `pipe()` permet de créer un tube anonyme

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

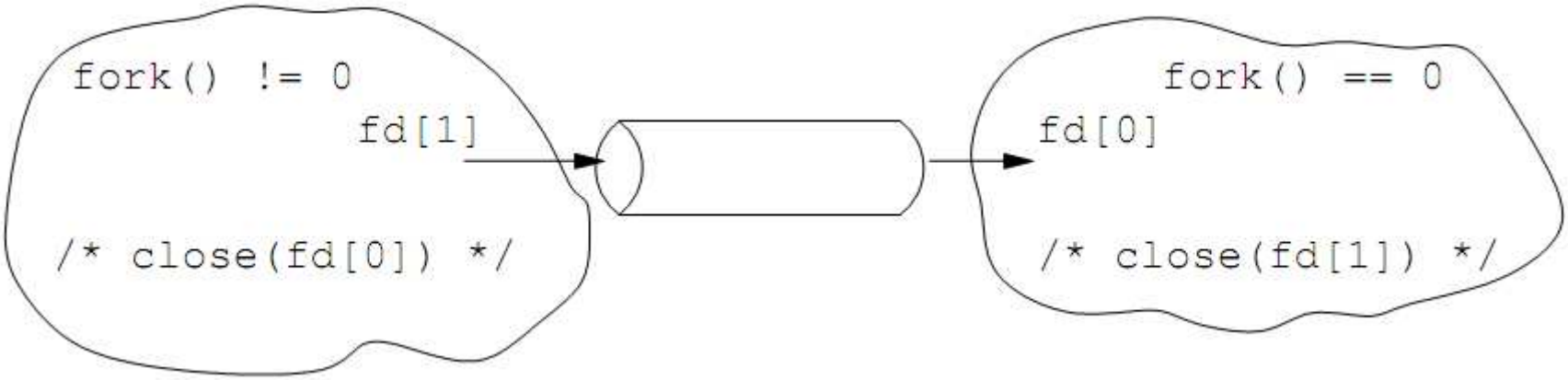
- `pipefd` : tableau de descripteurs de fichiers retournés
  - `pipefd[0]` : utilisé en lecture
  - `pipefd[1]` : utilisé en écriture
- Retourne 0 ou -1 si erreur (avec `errno`)

- Les descripteurs sont utilisables avec les fonctions systèmes (`read()`, `write()`)
- Il n'est pas possible de positionner le pointeur de lecture / écriture (`lseek()`)

- Lorsque les 2 descripteurs sont fermés - avec `close()` - le tube est automatiquement détruit
- S'il n'a pas de lecteur, les données écrites sont perdues
  - Processus écrivant reçoit `SIGPIPE` (terminaison par défaut)
  - Si ce signal est ignoré par le processus, `write()` retourne -1 avec `EPIPE`
- S'il n'y plus d'écrivain, une lecture avec `read()` retourne 0

Tubes – pipe() (suite)

- Typiquement, un tube anonyme est utilisé pour passer des données entre un processus et un des ses descendants :
  - Le tube est créé par le processus père
  - Le processus père crée un processus fils avec `fork()`
  - Le père et le fils partagent les descripteurs du tube
  - Chacun utilise un bout du tube
  - Chacun ferme son coté du tube



Tubes – pipe() – Exemple

```
/* Toutes les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

int fd[2];
char buf[PIPE_BUF];
int pid, len;

pipe(fd);
pid = fork();
if (pid == 0) {
   close(fd[1]); /* Child is reader, close the write descriptor */
   while ((len = read(fd[0], buf, PIPE_BUF)) > 0) {
        write(STDOUT_FILENO, buf, len);
   }
   close(fd[0]);
} else {
   close(fd[0]); /* Parent is writer, close the read descriptor */
   write(fd[1], "123\n", 4);
   close(fd[1]);
   waitpid(pid, NULL, 0);
}
```

Tubes – Tubes nommés

- Inconvénients des tubes anonymes :
  - Processus utilisant le tube doivent avoir un ancêtre commun
  - Pas de rémanence du tube (au plus tard détruit lorsque les processus terminent)
- Les tubes nommés pallient à ces inconvénients :
  - Pas de lien de parenté ou autre entre processus requis
  - Présents dans le système de fichiers
  - Rémanence

- Les tubes nommés ont le même comportement en lecture et écriture que les tubes anonymes

- Un tube nommé peut être créé et utilisé depuis un shell

```
$ mkfifo /tmp/fifo
$ ls -l /tmp/fifo
prw-r--r-- 1 joe users 0 Jan 12 14:25 /tmp/fifo
$ cat > /tmp/fifo
```

Tubes – mkfifo()

- La fonction `mkfifo()` permet de créer un tube nommé dans le système de fichiers

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

- `pathname` : nom du tube nommé
- `mode` : permissions (comme pour `open()`)
- Retourne 0 ou -1 si erreur (avec `errno`)

- Un tube nommé est accessible comme un fichier régulier
- Impératif de choisir un mode d'écriture ou de lecture (exclusif, pas de `O_RDWR`)
- L'ouverture du tube avec `open()` est bloquante par défaut :
  - En lecture, bloque jusqu'à ce que le tube soit ouvert en écriture
  - En écriture, bloque jusqu'à ce que le tube soit ouvert en lecture

- Une lecture dans un tube non ouvert renvoie `EOF`
- Une écriture dans un tube non ouvert génère le signal `SIGPIPE`

Inter-Process Communications – System V

IPC – Introduction

- System V définit trois mécanismes de communication entre processus :
  - Files de messages
  - Sémaphores
  - Segments de mémoire partagée

- Les trois mécanismes ont un certain nombre de propriétés en commun :
  - Toute ressource IPC active est associée à une clé choisie par l'utilisateur, servant d'identification globale
  - Les fonctions "...get" permettent de créer une nouvelle ressource ou d'accéder à une ressource existante
  - Les fonctions "...ctl" permettent d'interroger ou modifier les paramètres d'une ressource, ainsi que de la supprimer
  - Chaque ressource a des droits d'accès similaires à ceux des fichiers (UID, GID, permissions)
  - Une ressource doit être explicitement détruite (rémanence)

- La commande `ipcs` liste les ressources IPC actives du système
- La commande `ipcrm` permet d'effacer une ressource IPC

IPC – Introduction (suite)

- Une clé est utilisée pour identifier une ressource IPC
- Cette clé est partagée entre les processus partageant une même ressource IPC

- Un processus ne peut donc accéder à une ressource IPC que s'il en connait la clé
- L'accès à une ressource IPC se fait par une des fonctions "...get" en précisant la clé
  - Ces fonctions "...get" retournent un identificateur de la ressource, utilisé dans les fonctions accédant à cette ressource

- Une clé peut être un simple nombre entier décidé à l'avance
- Une ressource IPC peut être complètement privée en utilisant `IPC_PRIVATE`
  - Le système génère alors la clé

- `IPC_CREAT` : permet de créer une nouvelle entrée s'il n'en existe pas déjà une pour cette clef
- `IPC_EXCL` : lorsque que combiné avec le précédent, une erreur est générée si l'entrée existe déjà

- Fonctions et macros relatives aux clés IPC sont définies dans `<sys/ipc.h>`

IPC –ftok()

- La fonction `ftok()` permet de générer une clé IPC depuis un nom de fichier

```
key_t ftok(const char *pathname, int proj_id);
```

- `pathname` : nom d'un fichier
- `proj_id` : nombre différent de 0
- Retourne une clé ou -1 si erreur (avec `errno`)

- La clé généré est toujours la même tant que `pathname` et que les 16 bits les moins significatifs de `proj_id` sont identiques
- Le fichier doit exister et être accessible (le système utilise le numéro d'inode)

IPC System V - Files de Messages

Files de messages – Introduction

- Une file de messages est une liste chainée de données gérée par le noyau
- Les messages consistent en une structure composée d'un type et du message en lui-même
- Les messages sont ajoutés à la fin de la file
- Les messages peuvent être lus dans n'importe quel ordre

- Les files de messages sont identifiées par une clé IPC
- Une file de messages peut donc être utilisée par n'importe quel processus connaissant la clé IPC

- Les fonctions sur les files de messages sont déclarées dans `<sys/msg.h>`

Files de messages – msgget()

- La fonction `msgget()` crée ou ouvre une file de messages

```
int msgget(key_t key, int msgflg);
```

- `key` : clé IPC ou `IPC_PRIVATE`
- `msgflg` : permissions d'accès, `IPC_CREAT` ou `IPC_EXCL`
- Retourne un identificateur de file ou -1 si erreur (avec `errno`)

- Crée une nouvelle file si :
  - Aucune file n'existe avec cette clé et `IPC_CREAT` est spécifié dans `msgflg`
  - `IPC_EXCL` est spécifié dans `msgflg`
  - La clé spécifiée est `IPC_PRIVATE`

Files de messages – msgget() - Exemples

- Création d'une file privée, utilisable que par le processus courant et ses descendants :

```
int qid = msgget(IPC_PRIVATE, 0600);
```

- Création d'une file permettant aux processus d'un même groupe de communiquer :

```
#define MAGIC_NUMBER 0x1234
key_t my_key = (key_t) MAGIC_NUMBER;
int qid = msgget(my_key, IPC_CREAT | 0660);
```

- Création d'une file permettant à tout processus de lire la file mais réservant l'écriture au processus de même UID que la file :

```
#define VERSION 0x1234
key_t my_key = ftok(argv[0], VERSION);
int qid = msgget(my_key, IPC_CREAT | IPC_EXCL | 0622);
```

Files de messages – Structure des messages

- Un message est une zone de mémoire contigüe contenant le type du message suivi des données
- Le type du message est un nombre interne à l'application et est employé pour filtrer les messages en lecture
- Le type du message doit être supérieur à 0

- Un message est donc déclaré comme une structure

```
typedef struct {
    long mtype;
    char mtext[25];
    time_t mtime;
};
```

- La structure ne peut pas contenir de pointeur

Files de messages – msgsnd()

- La fonction `msgsnd()` envoie un message dans une file

```
ssize_t msgsnd(int qid, const void *msgp, size_t msgsz, int
                                                    msgflg);
```

- `gid` : identificateur de file
- `msgp` : pointeur sur la structure du message à envoyer
- `msgsz` : longueur de `msgp` sans inclure le type
- `msgflg` : 0 ou `IPC_NOWAIT` (ne bloque pas si la file est pleine)
- Retourne 0 ou -1 si erreur (avec `errno`)

Files de messages – msgrcv()

- La fonction `msgrcv()` permet de récupérer un message dans une file

```
ssize_t msgrcv(int qid, void *msgp, size_t msgsz, long
                                        msgtyp, int msgflg);
```

- `qid` : identificateur de file
- `msgp` : pointeur sur la structure du message à lire
- `msgsz` : longueur de `msgp` sans inclure le type ou `MSGMAX`
- `msgtyp` : sélectionne les messages à lire :
  - `0` : premier message de la liste (FIFO)
  - `> 0` : premier message de même type
  - `< 0` : premier message ayant le plus petit type inférieur ou égal à la valeur absolue de `msgtyp`
- `msgflg` : 0 ou OU binaire de :
  - `IPC_NOWAIT` : ne bloque pas si la file est vide
  - `MSG_EXCEPT` : récupère un message de n'importe quel type sauf celui spécifié dans `msgtyp` (qui doit être alors >0)
  - `MSG_NOERROR` : si le message est trop long, il sera tronqué sans que `E2BIG` se produise
- Retourne la taille du message lu, sans son type, ou -1 si erreur (avec `errno`)

Files de messages – msgctl()

- La fonction `msgctl()` permet de contrôler ou effacer une file

```
int msgctl(int qid, int cmd, struct msqid_ds *buf);
```

- `qid` : identificateur de file
- `cmd`: commande :
  - `IPC_RMID` : supprime la file
  - `IPC_STAT` : renseigne la structure `buf`
  - `IPC_SET` : modifie les droits d'accès de la file
- `*buf` : structure comprenant les droits d'accès (`NULL` pour `IPC_RMID`)
- Retourne 0 ou une valeur non négative (dépend de `cmd`) ou -1 si erreur (avec `errno`)

Files de messages – Exemple

```
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

#define MSG_MAXLEN 100
#define MSG_TYPE 0x555

struct msg_st {
   long msg_type;
   char msg_text[MSG_MAXLEN];
};

int sendmsg(void)
{
   int qid, len;
   struct msg_st message;
   message.msg_type = MSG_TYPE;
```

Files de messages – Exemple (suite)

```
    strcpy(message.msg_text, "Linux rulez");
    len = strlen(message.msg_text) + 1;
    if ((qid = msgget((key_t)1234, 0666 | IPC_CREAT)) == -1) {
         perror("msgget");
         return -1;
    }
    if (msgsnd(qid, (void *)&message, len, 0) == -1) {
         perror("msgsnd");
         return -1;
    }

    /* do something */

    if (msgctl(qid, IPC_RMID, 0) == -1) {
         perror("msgctl");
         return -1;
    }
    return 0;
}
```

Files de messages – Exemple (suite)

```
int readmsg(void)
{
   int qid, len;
   struct msg_st message;

   if ((qid = msgget((key_t)1234, 0666 | IPC_CREAT)) == -1) {
        perror("msgget");
        return -1;
   }

   if (msgrcv(qid, (void *)&message, MSG_MAXLEN, MSG_TYPE, 0) == -1) {
        perror("msgsnd");
        return -1;
   }

   /* do something */

   return 0;
}
```

IPC System V - Sémaphores

Sémaphores – Introduction

- Les sémaphores System V sont une généralisation des sémaphores de Dijkstra
- Ils ne sont pas binaires mais sont des compteurs :
  - Le sémaphore est incrémenté ou décrémenté
  - Permet d'autoriser l'accès simultané d'un certain nombre de processus à une quantité donnée de ressources (ex. système multi-CPU)
  - Les compteurs des sémaphores sont initialement supposés vides (POSIX) mais Linux les initialise à 0
- Un sémaphore System V est en fait un ensemble de sémaphores :
  - Chaque sémaphore est constitué :
    - La valeur du sémaphore
    - Le PID du dernier processus l'ayant manipulé
    - Le nombre de processus en attente d'une augmentation
    - Le nombre de processus en attente d'une mise à zéro
- Un ensemble de sémaphores est identifié par une clé IPC

- Les fonctions sur les sémaphores System V sont déclarées dans `<sys/sem.h>` et sont atomiques

- Les sémaphores Posix sont plus simples à utiliser…

Sémaphores – semget()

- La fonction `semget()` crée ou accède à un ensemble de sémaphores

```
int semget(key_t key, int nsems, int semflg);
```

- `key` : clé IPC ou `IPC_PRIVATE`
- `nsems` : nombre de sémaphores requis dans l'ensemble
- `semflg` : permissions d'accès, `IPC_CREAT` ou `IPC_EXCL`
- Retourne un identificateur de l'ensemble de sémaphores ou -1 si erreur (avec `errno`)

- Crée une nouvel ensemble de sémaphores si :
  - Aucune ensemble n'existe avec cette clé et `IPC_CREAT` est spécifié dans `semflg`
  - `IPC_EXCL` est spécifié dans `semflg`
  - La clé spécifiée est `IPC_PRIVATE`

Sémaphores – semop()

- La fonction `semop()` permet de changer la valeur du compteur d'un sémaphore

```
int semop(int semid, struct sembuf *sops, unsigned nsops);
```

- `semid` : identificateur de l'ensemble de sémaphores
- `sops` : pointeur sur tableau de structures `sembuf` :

```
struct sembuf {
    short sem_num;   numéro du sémaphore dans l'ensemble
    short sem_op;    opération à effectuer (voir ci-après)
    short sem_flg;   0 ou SEM_UNDO (voir ci-après)
};
```

- `nsops` : nombre de structures `sops`
- Retourne 0 ou -1 si erreur (avec `errno`)

Sémaphores – semop() (suite)

- SEM_UNDO indique au noyau de mémoriser l'opération faite par le processus courant ; si celui-ci termine sans libérer le sémaphore, le noyau fera alors l'opération inverse

- L'opération effectuée dépend de la valeur de sem_op :
- Si sem_op > 0 :
  - Le compteur du sémaphore est augmenté de la valeur de sem_op
  - Le processus appelant n'est jamais bloqué
  - Tous les processus en attente d'une augmentation du sémaphores sont réveillés
- Si sem_op = 0 :
  - Le processus est mis en attente jusqu'à ce que le compteur soit égal à 0
- Si sem_op < 0 :
  - Si le compteur est supérieur ou égal à la valeur absolue de sem_op :
    - La valeur absolue de sem_op est retranchée du compteur
    - Le processus n'est pas bloqué
  - Sinon le processus est bloqué jusqu'à ce que le compteur soit supérieur ou égal à la valeur absolue de sem_op

Sémaphores – semctl()

- La fonction `semctl()` permet de contrôler ou effacer un ensemble de sémaphores

```
int semctl(int semid, int semnum, int cmd, ...);
```

- `semid` : identificateur de l'ensemble de sémaphores
- `semnum` : numéro du sémaphore dans l'ensemble
- `cmd`: commande :
  - `IPC_RMID` : supprime l'ensemble de sémaphores
  - `SETVAL` : initialise le compteur du sémaphore semnum avec une valeur spécifique
  - `SETALL` : initialise les compteurs des sémaphores de l'ensemble avec une valeur spécifique
- `...` : paramètres éventuels de `cmd`
- Retourne une valeur non négative (dépend de `cmd`) ou -1 si erreur (avec `errno`)

Sémaphores – semctl() (suite)

- SETVAL et SETALL utilise un membre d'un union :

```
union semun {
   int val;
   struct semid_ds *buf;
   ushort *array;
} sem_union;
```

- Pour passer une valeur à la commande SETVAL, on utilise le membre `val`
- Pour passer une valeur à la commande SETALL, on utilise le membre `array`
- Exemple :

```
sem_union.val = 1;
semtctl(semid, 0, SETVAL, sem_union);
```

- IPC_RMID supprime l'ensemble de sémaphores associé à la clé, et débloque tous les processus en attente (Ils peuvent recevoir EIDRM) :

```
semtctl(semid, 0, IPC_RMID);
```

Sémaphores – Exemple

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
union semun {
    int val;
    struct semid_ds *buf;
    ushort *array;
} sem_union;

int main(void)
{
    int sid;
    struct sembuf sem_b;
    sem_b.sem_num = 0;
    sem_union.val = 1;
    if ((sid = semget((key_t)1234, 1, 0666 | IPC_CREAT)) == -1) {
          perror("semget");
          exit(EXIT_FAILURE);
    }
    if (semctl(sid, 0, SETVAL, sem_union) == -1) {
          perror("semctl");
          exit(EXIT_FAILURE);
    }
```

Sémaphores – Exemple (suite)

```
        /* begin critical section */
        sem_b.sem_op = -1;
        if (semop(sid, &sem_b, 1) == -1) {
              perror("semop: -1");
              exit(EXIT_FAILURE);
        }

        /* do something */

        /* end critical section */
        sem_b.sem_op = 1;
        if (semop(sid, &sem_b, 1) == -1) {
              perror("semop: +1");
              exit(EXIT_FAILURE);
        }
        if (semctl(sid, 0, IPC_RMID) == -1) {
              perror("semctl: IPC_RMID");
              exit(EXIT_FAILURE);
        }
        exit(EXIT_SUCCESS);
}
```

IPC System V - Mémoire partagée

Mémoire partagée – Introduction

- La mémoire partagée permet à deux processus non relatifs d'accéder au même segment de mémoire logique
- C'est le moins de communication entre processus le plus efficace (pas de copie des données transmises, pas de changement de contexte user / kernel space)
- La taille d'un segment est arrondie au multiple supérieur de la dimension d'une page (4KB sur x86)
  - Il faut éviter de créer trop de petits segments
  - Pour partager de multiples variables, le mieux est donc de les réunir dans un tableau ou une structure
- Il est impératif de synchroniser les processus accédant à un segment (sémaphores par exemple)
- Les fonctions relatives utilisent une clé IPC et sont déclarées dans `<sys/shm.h>`

Mémoire partagée – shmget()

- La fonction `shmget()` crée ou accède à un segment de mémoire partagée

```
int shmget(key_t key, size_t size, int shmflg);
```

- `key` : clé IPC ou `IPC_PRIVATE`
- `size` : taille du segment en octets
- `shmflg` : permissions d'accès, `IPC_CREAT` ou `IPC_EXCL`
- Retourne un identificateur de segment de mémoire partagée ou -1 si erreur (avec `errno`)

- Crée un nouvel segment de mémoire partagée si :
  - Aucune ensemble n'existe avec cette clé et `IPC_CREAT` est spécifié dans `shmflg`
  - `IPC_EXCL` est spécifié dans `shmflg`
  - La clé spécifiée est `IPC_PRIVATE`

Mémoire partagée – shmget() (suite)

- La taille spécifiée lors de la création d'un segment doit être comprise entre SHMIN et SHMAX (sinon une erreur est retournée)
- Lors de la création, la taille du segment est arrondie au multiple supérieur de la dimension d'une page (souvent 4 kB sur un système 32 bits)
- Lors de l'accès à un segment déjà existant, la taille spécifiée ne doit pas être supérieure à la taille effective du segment
  - La taille du segment ne sera jamais diminuée, on peut donc spécifier 0

Mémoire partagée – shmat()

- Une fois l'identifiant d'un segment obtenu avec `shmget()`, il faut l'attacher dans l'espace mémoire du processus courant, avec la fonction `shmat()`.

```
void *shmat(int shmid, const void *shmaddr, int shmflg);
```

- `shmid` : identificateur du segment
- `shmaddr` : adresse où attacher le segment, ou 0 pour laisser le système choisir l'adresse
- `shmflg` : 0 ou `SHM_RDONLY` (segment en lecture seule)
- Retourne l'adresse du premier octet du segment ou `(void *)-1` si erreur (avec `errno`)

- En général, l'adresse est toujours `(void *)0`

- Après un fork(), un processus fils hérite des segments attachés du processus père
- Apres un execve() ou pendant _exit(), tous les segments attachés sont détachés du processus

Mémoire partagée – shmdt()

- La fonction `shmdt()` détache un segment mémoire partagée de l'espace mémoire du processus courant

```
int *shmdt(const void *shmaddr);
```

- `shmaddr` : adresse du segment mémoire (pointeur retourné par `shmat()`)
- Retourne 0 ou -1 si erreur (avec `errno`)

Mémoire partagée – shmctl()

- La fonction `shmctl()` permet de contrôler ou effacer un segment

```
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

- `shmid` : identificateur du segment
- `cmd`: commande :
  - `IPC_RMID` : supprime le segment
  - `IPC_STAT` : renseigne la structure `buf`
  - `IPC_SET` : modifie les droits d'accès du segment
  - `SHM_LOCK` ou `SHM_UNLOCK` : prévient ou autorise le segment à être mis en swap (spécifique à Linux)
- `*buf` : structure comprenant les droits d'accès
- Retourne une valeur non négative (dépend de `cmd`) ou -1 si erreur (avec `errno`)

Mémoire partagée – Exemple

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

#define MEM_SZ 4096

struct shared_stuff_st {
   int a_number;
   char some_text[BUFSIZ];
};

int main(void)
{
   void *shared_memory = (void *)0;
   int sid;
   struct shared_stuff_st *shared_stuff;
   if ((sid = shmget((key_t)1234, MEM_SZ, 0666 | IPC_CREAT)) == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
   }
```

Mémoire partagée – Exemple (suite)

```
    if ((shared_memory = shmat(sid, (void *)0, 0)) == (void *)-1) {
          perror("shmat");
          exit(EXIT_FAILURE);
    }
    printf("Memory attached at : %p\n", shared_memory);

    shared_stuff = (struct shared_stuff_st *)shared_memory;

    /* do something */

    if (shmdt(shared_memory) == -1) {
          perror("shmdt");
          exit(EXIT_FAILURE);
    }
    if (shmctl(sid, IPC_RMID, 0) == -1) {
          perror("shmctl");
          exit(EXIT_FAILURE);
    }
    exit(EXIT_SUCCESS);
}
```

Threads

Threads - Introduction

- Un thread peut être vu comme une activité au sein d'un processus
  - Un thread existe au sein d'un processus et utilise ses ressources
  - Un thread ne peut exister hors d'un processus
- Un thread est schedulé en tant qu'entité indépendante (tant que le processus existe)
  - Sous Linux 2.6, le scheduleur ne fait pas de différence entre un thread et un processus (tous sont des "tasks")
- Les threads sont moins couteux à créer et à scheduler que les processus
  - Moins de overhead du système pour créer un thread
  - Moins de ressources système pour manager un thread
- La création d'un thread duplique seulement le minimum de ressources nécessaires pour que le thread existe comme code exécutable
  - Chaque thread d'un processus a sa propre pile
  - Les threads d'un même processus partagent le reste (code, données, descripteurs de fichiers, signaux)
  - Pas de protection mémoire entre threads d'un même processus
- La communication inter-threads est plus efficace qu'inter-processus

Threads – Introduction (suite)

- Comparaison entre la création de 50000 processus et la création de 50000 threads :

	ARM7 @ 265 MHz		Intel Core2 @ 2.99GHz	
50000	forks	threads	forks	threads
Real	3m 4.08s	0m 33.06s	0m7.488s	0m1.134s
user	0m 9.90s	0m 17.72s	0m0.991s	0m0.074s
sys	2m54.15s	0m 14.14s	0m7.387s	0m0.403s

- Dans l'environnement UNIX, l'interface thread a été standardisée et est spécifiée par IEEE POSIX 1003.1c standard (1995)
- Les implémentations suivant ce standard sont référées comme POSIX threads, ou Pthreads.
- La dernière version du standard Pthread est IEEE Std 1003.1, 2004 Edition.

Threads – Librairie NPTL

- Native Posix Threads Library :
  - glibc ≥ 2.3.2
  - Kernel 2.6
  - Tous les threads d'un processus partagent le même PID
  - La version NTPL peut être obtenue avec

```
$ getconf GNU_LIBPTHREAD_VERSION
```

- Les fonctions sont prototypées dans `<pthread.h>`
- La plupart des fonctions retournent 0 ou un code si erreur (avec `errno`)

- L'édition des liens se fait avec `-lpthread`
- Ceci déclare automatiquement la macro `_REETRANT` :
  - Utilise les fonctions qui ont une version réentrante (avec `_r` ajouté à leur nom)
  - Certaines fonctions de `stdio.h`, implémentées par macros, sont remplacées par des fonctions réentrantes
  - La variable `errno` est remplacée par une fonction

Threads – pthread_create()

- La fonction `pthread_create()` crée un nouveau thread

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

- `thread` : identificateur du thread
- `attr` : attributs du thread (souvent `NULL`)
- `start_routine` : fonction à exécuter
- `arg` : argument de la fonction `start_routine`
- Retourne 0 ou un numéro d'erreur (le contenu de `thread` est alors indéfini)

- Contrairement à `fork()`, un nouveau thread démarre donc dans une fonction spécifique
- Lorsque cette fonction retourne, le thread est terminé

Threads – pthread_create() - Exemple

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

void *printxs(void *unused)
{
   while (1)
        fputc('x', stderr);
}

int main(void)
{
   pthread_t thread_id;
   if (pthread_create(&thread_id, NULL, &printxs, NULL) != 0) {
        perror ("pthread_create");
        exit(EXIT_FAILURE);
   }
   while(1)
        fputc('o', stderr);
}
```

Threads – pthread_exit()

- La fonction `pthread_exit()` permet à thread de terminer

```
void pthread_exit(void *val);
```

- `val` : pointeur sur objet retourné
- Le thread peut donc retourner plus qu'un simple entier
- Par défaut, le noyau attend que la valeur de retour soit collectée pour réellement terminer un thread
  - Le thread est dit "joinable"
- Il est possible de rendre un thread "détachable"
  - Il est terminé automatiquement, le noyau n'attend pas que sa valeur de retour soit collectée (un peu comme un système de garbage collector)
  - Sa valeur de retour ne peut pas être récupérée par un autre thread

Threads – pthread_join()

- La fonction `pthread_join()` permet à un thread d'attendre la fin de l'exécution d'un autre thread

```
int pthread_join(pthread_t thread_id, void **thread_return);
```

- `thread_id` : identificateur du thread à attendre
- `thread_return` : pointeur sur valeur retournée par `return` ou objet retourné par `pthread_exit()`
- Retourne 0 ou un numéro d'erreur

- Un thread ne doit pas utiliser `pthread_join()` sur lui-même

Threads – pthread_join() – Exemple

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

void *thread_function(void *unused)
{
   sleep(3);
   pthread_exit("Bye bye");
}

int main(void)
{
   pthread_t thread_id;
   void *thread_result;
```

Threads – pthread_join() – Exemple (suite)

```
    if (pthread_create(&thread_id, NULL, &thread_function, NULL)) {
        perror("pthread_create");
        exit(EXIT_FAILURE);
    }

    fprintf(stderr, "Waiting for thread to end\n");
    if (pthread_join(thread_id, &thread_result)) {
        perror("pthread_join");
        exit(EXIT_FAILURE);
    }
    fprintf(stderr, "Thread returned : %s\n", (char *)thread_result);

    exit(EXIT_SUCCESS);
}
```

Threads – pthread_detach()

- La fonction `pthread_detach()` permet de détacher un thread

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

- Retourne 0 ou un numéro d'erreur

Threads – Attributs

- Les attributs d'un thread sont spécifiés dans un objet de type `pthread_attr_t`, qui doit être initialisé avant utilisation, puis détruit après utilisation
- Cet objet est passé à la fonction `pthread_create()`
- Chaque attribut d'un thread est spécifié ou lu avec une fonction dédiée

- Les attributs d'un thread et leurs fonctions sont :
  - Detach state — pthread_attr_setdetachstate()
  - Contention scope — pthread_attr_setscope()
  - Inherit scheduler — pthread_attr_setinheritsched()
  - Scheduling policy — pthread_attr_setschedpolicy()
  - Scheduling priority — pthread_attr_setschedparam
  - Guard size — pthread_attr_setguardsize()
  - Stack size and address — pthread_attr_setstack()

- A chaque fonction pthread_attr_set*() correspond une fonction pthread_attr_get*()

Threads – Attributs – Exemple

- Exemple : rendre un thread détachable dès sa création

```
pthread_attr_t attr;
pthread_t thread;
pthread_attr_init(&attr);
pthread_attr_setdetachstate(&attr, PTHREAD_CREATE_DETACHED);
pthread_create(&thread, &attr, &thread_func, NULL);
pthread_attr_destroy(&attr)
```

Threads – pthread_cancel()

- La fonction `pthread_cancel()` permet de demander au système la terminaison d'un thread avant la fin de son exécution

```
int pthread_cancel(pthread_t thread_id);
```

- `thread_id` : identificateur du thread
- Retourne 0 ou !0 si erreur

- Un thread terminé par `phread_cancel()` doit en principe être "joined" avec `phread_join()`
  - La valeur de retour est `PTHREAD_CANCELED`

- Les ressources allouées dans le thread ne sont pas automatiquement libérées

Threads – pthread_cancel() (Suite)

- Il est possible de contrôler dans un thread si et où il peut être terminé par `pthread_cancel()`
  - Le thread peut être "cancelable" de façon asynchrone
  - Le thread peut être "cancelable" de façon synchrone (défaut)
  - Le thread peut être "uncancelable"

- Un thread "cancelable" de façon synchrone ne peut l'être qu'à certains points de son exécution : "cancelation points"
  - Les demandes de cancelation sont mises en attente jusqu'à ce que l'exécution du thread atteint l'un de ces points

- Le thread peut désactiver toute "cancelation" (section critique)

- Voir `man 7 pthreads` ou www.kernel.org/doc/man-pages/online/pages/man7/pthreads.7.html pour la liste des fonctions pouvant être un cancelation point.

Threads – pthread_setcancel*()

- La fonction `pthread_setcancel*()` permettent de changer le type de cancelation, et d'activer ou désactiver la cancellation

```
int pthread_setcanceltype(int type, int *oldtypr);
```

- `type` :
  - `PTHREAD_CANCEL_DEFERRED` (défaut)
  - `PTHREAD_CANCEL_ASYNCHRONOUS`
- `*oldtype` : valeur précédente de `type`
- Retourne 0 ou !0 si erreur

```
int pthread_setcancelstate(int state, int *oldstate);
```

- `state` :
  - `PTHREAD_CANCEL_ENABLED` (défaut)
  - `PTHREAD_CANCEL_DISABLED`
- `*oldstate` : valeur précédente de `state`
- Retourne 0 ou !0 si erreur

Threads – pthread_setcancelstate() – Exemple

```
/* Toutes les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

float *account_balances;          /* array of balances in accounts */

int process_transaction(int from_acct, int to_acct, float dollars)
{
   int old_cancel_state;

   if (account_balances[from_acct] < dollars) {
        return 1;
   }

   /* begin critical section */
   pthread_setcancelstate(PTHREAD_CANCEL_DISABLE, &old_cancel_state);
   /* move the money */
   acount_balances[to_acct] += dollars;
   acount_balances[from_acct] -= dollars;
   /* end critical section */
   pthread_setcancelstate(old_cancel_state, NULL);

   return 0;
}
```

Threads – pthread_testcancel()

- La fonction `pthread_testcancel()` teste et crée un point de cancelation

```
void pthread_testcancel(void);
```

- Si la cancelation est désactivée, ou s'il n'y a pas de requête de cancelation en attente, la fonction est sans effet

- Si une requête de cancelation était en attente, le thread est terminé, la fonction `pthread_testcancel()` ne retourne pas

Threads – pthread_cleanup_...()

- Il est possible d'appeler automatiquement une liste de fonctions lorsqu'un thread est terminé par `pthread_cancel()` ou `pthread_exit()`
- Cette liste est une stack FIFO
- Une fonction est ajoutée au début de la liste avec `pthread_cleanup_push()`
- La fonction de la liste est enlevée avec `pthread_cleanup_pop()`

```
void pthread_cleanup_push(void (*func)(void *), void *arg);
```

- `func` : fonction à appeler
- `arg` : argument de la fonction à passer à la fonction

```
void pthread_cleanup_pop(int execute);
```

- `execute` : si différent de 0, exécute la fonction avant de l'enlever de la liste

Threads – pthread_cleanup_*() – Exemple

```
/* Toutes les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

void deallocate_buffer(void *buffer)
{
   free(buffer)
}

void *thread_function(void *unused)
{
   void *tmp_buf = malloc(1024);

   pthread_cleanup_push(deallocate_buffer, tmp_buf);
   do_work();
   pthread_cleanup_pop(1);
}
```

Threads - Mutexes

- Lorsque plusieurs threads accèdent et modifient une ressource commune, il est possible de se trouver dans une situation imprévue
- Pour garantir qu'un seul thread accède à une ressource, il faut utiliser un verrou d'exclusion mutuelle (MUTual Exclusion lock)
- Un seul thread peut verrouiller un mutex, les autres threads sont alors bloqués
- Lorsque le mutex est déverrouillé, un seul thread est alors débloqué

- Il y a 3 types de mutexes
  - Fast mutexes ou mutexes normaux (type par défaut, basés sur les futex)
    - Un thread peut verrouiller plusieurs fois le mutex, et se bloque alors lui-même (deadlock)
  - Recursive mutexes
    - Le mutex compte le nombre de fois qu'un même thread le verrouille (pas de deadlock)
  - Error-checking mutexes
    - Un deuxième verrouillage consécutif par un thread retourne une erreur (pas de deadlock)
    - Non POSIX, spécifique à Linux

Threads – pthread_mutex_init()

- La fonction `pthread_mutex_init()` initialise un mutex

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

- `*mutex` : mutex
- `attr` : attributs du mutex
  - `NULL` : type par défaut, sinon permet de spécifier le type (voir ci-après)
- Retourne 0 ou un numéro d'erreur

- Autre façon de créer un mutex avec attributs par défaut

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
```

- Lorsque le mutex n'est plus utile, on le détruit avec :

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

- `*mutex` : mutex
- Retourne 0 ou un numéro d'erreur

Threads – Type du mutex

- La fonction `pthread_mutexattr_settype()` permet de spécifier le type d'un mutex avant sa création.

```
int pthread_mutexattr_init(pthread_mutexattr_t *attr);
int pthread_mutexattr_settype(pthread_mutexattr_t *attr,
                              int type);
int pthread_mutexattr_destroy(pthread_mutexattr_t *attr);
```

- `type` : type du mutex :
  - `PTHREAD_MUTEX_NORMAL` ou `PTHREAD_MUTEX_DEFAULT`
  - `PTHREAD_MUTEX_ERRORCHECK`
  - `PTHREAD_MUTEX_RECURSIVE`
- Retournent 0 ou un numéro d'erreur

Threads – Type du mutex – Exemples

- Mutex par defaut

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

- Mutex d'un autre type

```
pthread_mutex_t mutex;
pthread_mutexattr_t attr;

pthread_mutexattr_init(&attr);
pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_RECURSIVE);
pthread_mutex_init(&mutex, &attr);
```

Threads – pthread_mutex_*lock()

- Les fonctions `pthread_mutex_*lock()` permettent de verrouiller ou déverrouiller un mutex

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

- Verrouille le mutex s'il n'est pas déjà verrouillé
- Sinon bloque le thread jusqu'à ce que le mutex soit déverrouillé

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

- Déverrouille le mutex

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

- Ne bloque pas si le mutex est déjà verrouillé
- Retourne 0 si le mutex peut être verrouillé, sinon retourne EBUSY

Threads – Variables conditionnelles

- Lorsqu'un thread attend qu'une condition spécifique arrive, il pourrait en boucle verrouiller un mutex, tester la condition, et déverrouiller le mutex
- Exemple :

```
while (1) {
    int flag_is_set;
    pthread_mutex_lock(&mutex);
    flag_is_set = thread_flag;    // thread_flag is global
    pthread_mutex_unlock(&mutex);
    if (flag_is_set)
        do_work();
}
```

- Busy-loop waiting est inefficace (consommation de CPU inutile)
- Le mécanisme de variable conditionnelle permet de bloquer un thread jusqu'à ce qu'une condition devienne vraie

Threads – Variables conditionnelles (suite)

- Une variable conditionnelle fonctionne avec un mutex :
  - Le mutex est déverrouillé pour que la variable soit accessible par d'autres threads, tandis que le thread courant est bloqué
  - Un autre thread peut alors verrouiller le mutex et manipuler la variable
  - Lorsqu'un autre thread a fini avec la variable, il envoie un signal, puis déverrouille le mutex
  - Le thread courant est débloqué et peut continuer

Threads – pthread_cond_init()

- La fonction `pthread_cond_init()` initialise une variable conditionnelle

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

- `cond` : variable conditionnelle
- `attr` : attributs de la variable (souvent `NULL`)
- Retourne 0 ou un numéro d'erreur

- Autre façon de créer une variable conditionnelle avec attributs par défaut

```
pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
```

- Lorsque la variable conditionnelle n'est plus utile, on la libère avec :

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

- `cond` : variable conditionnelle
- Retourne 0 ou un numéro d'erreur

Threads – pthread_cond_wait()

- La fonction `pthread_cond_wait()` permet d'attendre qu'un autre thread signale le changement de la variable conditionnelle

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

- `cond` : variable conditionnelle
- `mutex` : mutex associé
- Retourne 0 ou un numéro d'erreur

- Le mutex doit être déjà verrouillé par le thread
- Le mutex sera toujours verrouillé au retour de la fonction
- Plusieurs threads peuvent attendre et être bloqués

Threads – pthread_cond_signal()

- La fonction `pthread_cond_signal()` permet de débloquer un thread en attente de la variable conditionnelle

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

- `cond` : variable conditionnelle
- Retourne 0 ou un numéro d'erreur

- Si aucun thread n'est bloqué en attente de la variable conditionnelle, le signal est ignoré

- Tous les threads en attente peuvent être débloqués avec :

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

- `cond` : variable conditionnelle
- Retourne 0 ou un numéro d'erreur

Threads – Variables conditionnelles – Exemple

```
/* Toutes les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t condvar = PTHREAD_COND_INITIALIZER;
int flag = 0;

void *thread_spining(void *unused)
{
    while (1) {
          pthread_mutex_lock(&mutex);
          while (!flag)
                    pthread_cond_wait(&condvar, &mutex);
          pthread_mutex_unlock(&mutex);
          do_work();
    }
}
void thread_setflag(int value)
{
    pthread_mutex_lock(&mutex);
    flag = value;
    pthread_cond_signal(&condvar);
    pthread_mutex_unlock(&mutex);
}
```

Threads – RW Locks

- Les mutex offrent un mécanisme d'exclusion mutuelle
- Parfois il n'est pas nécessaire d'avoir une exclusion mutuelle sur une ressource
- Exemple :
  - Plusieurs threads accèdent à une variable commune en lecture seulement

- POSIX offre un mécanisme de verrou en lecture ou écriture
  - Le verrou est verrouillé en lecture par les threads accédant à un ressource
  - Le verrou est verrouillé en écriture (rarement) par un ou d'autres threads
  - Les threads verrouillant en lecture ne sont pas bloqués si aucun thread ne verrouille en écriture

- L'implémentation Linux favorise les threads verrouillant en écriture
  - Risque de situation de famine pour les threads verrouillant en lecture

Threads – pthread_rwlock_init()

- La fonction `pthread_rwlock_init()` initialise un verrou R/W

```
int pthread_rwlock_init(pthread_rwlock_t *rwlock,
                        pthread_rwlockattr_t *attr);
```

- `rwlock` : verrou
- `attr` : attribut du verrou, souvent `NULL`
- Retourne 0 ou un numéro d'erreur (avec `errno`)

- Il est possible de partager une verrou entre plusieurs processus, voir `pthread_rwlockattr_setpshared ()`

- Le verrou sera détruit avec `pthread_rwlock_destroy()`

```
int pthread_rwlockattr_destroy(pthread_rwlockattr_t *attr);
```

- Retourne 0 ou un numéro d'erreur (avec `errno`)

Threads – pthread_rwlock_*lock()

- Les fonctions `pthread_rwlock_wrlock()` et `pthread_rwlock_rdlock()` permettent de verrouiller ou déverrouiller un verrou R/W

```
int pthread_rwlock_wrlock(pthread_rwlock_t *rwlock);
int pthread_rwlock_rdlock(pthread_rwlock_t *rwlock);
```

- `rwlock` : verrou
- Retournent 0 ou un numéro d'erreur (avec `errno`)

Threads – Barrières

- Dans un programme, il peut être nécessaire d'attendre qu'un certain nombre de taches soient complétées avant de continuer
- Les POSIX threads offrent un mécanisme permettant de synchroniser ainsi des threads appelé barrière
- Une barrière spécifie le nombre de thread à attendre, bloque les threads à la barrière jusqu'à ce qu'ils soient tous arrivés
- Lorsque le dernier thread arrive, les threads bloqués sont libérés

Threads – pthread_barrier_init()

- La fonction `pthread_barrier_init()` initialise une barrière

```
int pthread_barrier_init(pthread_barrier_t *barrier,
            pthread_barrierattr_t *attr, unsigned count);
```

- `barrier` : barrière
- `attr` : attribut de la barrière, souvent `NULL`
- `count` : nombre de threads à attendre (doit être > 0)
- Retourne 0 ou un numéro d'erreur (avec `errno`)

- Il est possible de partager une barrière entre plusieurs processus, voir `pthread_barrierattr_setpshared()`

- La barrière sera détruite avec `pthread_barrier_destroy()`

```
int pthread_barrier_destroy(pthread_barrier_t *barrier);
```

- Retourne 0 ou un numéro d'erreur (avec `errno`)

Threads – pthread_barrier_wait()

- La fonction `pthread_barrier_wait()` bloque tout thread appelant cette fonction jusqu'à ce que le compte spécifié soit atteint

```
int pthread_barrier_wait(pthread_barrier_t *barrier);
```

- `barrier` : barrière
- Retourne 0 ou !0 si erreur (avec `errno`), pour tout les threads sauf un seul qui recevra `PTHREAD_BARRIER_SERIAL_THREAD`

- Le thread recevant `PTHREAD_BARRIER_SERIAL_THREAD` peut-être utilisé pour détruire la barrière (ou toute autre action…) :

```
int rc = pthread_barrier_wait(&barrier)
if ( rc == PTHREAD_BARRIER_SERIAL_THREAD )
{
    pthread_barrier_destroy(&barrier);
}
```

Threads – Private data

- L'implémentation Posix offre la possibilité à chaque thread d'un programme d'avoir une ou plusieurs zones de données privées
- Une zone de données privée est distinguée par une clé

```
int pthread_key_create(pthread_key_t *key, void
                                    (*destructor)(void*));
int pthread_key_delete(pthread_key_t key);
```

```
int pthread_setspecific(pthread_key_t key, void *value);
void *pthread_getspecific(pthread_key_t key);
```

- L'accès aux zones de données privées est relativement lent, et elles sont rarement utilisées

Threads – Ordonnancement et priorité

- Comme pour les processus, une politique d'ordonnancement et une priorité particulières peuvent être spécifiées pour un thread

- Par défaut, un thread hérite de la politique d'ordonnancement et de la priorité de son ancêtre (processus ou autre thread)
  - L'héritage peut-être désactivé dans un attribut du thread
  - Si l'attribut d'héritage n'est pas désactivé, les attributs spécifiant la politique et de la priorité sont sans effet

- Le PID du thread doit-être 0 (root) pour pouvoir changer la politique et la priorité

Threads – pthread_attr_setinheritsched()

- La fonction pthread_attr_setinheritsched( ) permet de spécifier si la politique d'ordonnancement et la priorité sont héritées ou spécifié dans les attributs du thread

```
int pthread_attr_setinheritsched(pthread_attr_t *attr,
                                 int inheritsched);
```

- `inherited :`
  - `PTHREAD_INHERIT_SCHED` : politique et priorité sont hérité du thread créateur, les attributs sont donc ignorés (défaut)
  - `PTHREAD_EXPLICIT_SCHED` : honore les attributs de politique et priorité à la création du thread
- Retourne 0 ou un numéro d'erreur

Threads – pthread_attr_setsched*()

- Les fonction `pthread_attr_setsched*()` permettent de spécifier la politique d'ordonnancement et la priorité d'un thread

```
int pthread_attr_setschedpolicy(pthread_attr_t *attr,
                                int policy);
```

- `policy` : `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`
- Retourne 0 ou un numéro d'erreur

```
int pthread_setschedparam(pthread_attr_t *attr, struct
                          sched_param *param);
```

- *param : structure ayant un membre :
  - `int sched_priority` : priorité pour `SCHED_FIFO` ou `SCHED_RR`
- Retourne 0 ou un numéro d'erreur

Threads – pthread_attr_setsched*() – Exemple

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

#include <pthread.h>
#include <sched.h>

pthread_attr_t attr;
sched_param param;
pthread_t thread_id;
pthread_attr_init(&attr);

param.sched_priority = 20;
pthread_attr_setschedparam(&attr, &param);
pthread_attr_setschedpolicy(&attr, SCHED_RR);
pthread_attr_setinheritsched(&attr, PTHREAD_EXPLICIT_SCHED);

pthread_create (&thread_id, &attr, func, NULL);

/* do some work */

pthread_attr_destroy(&attr);
```

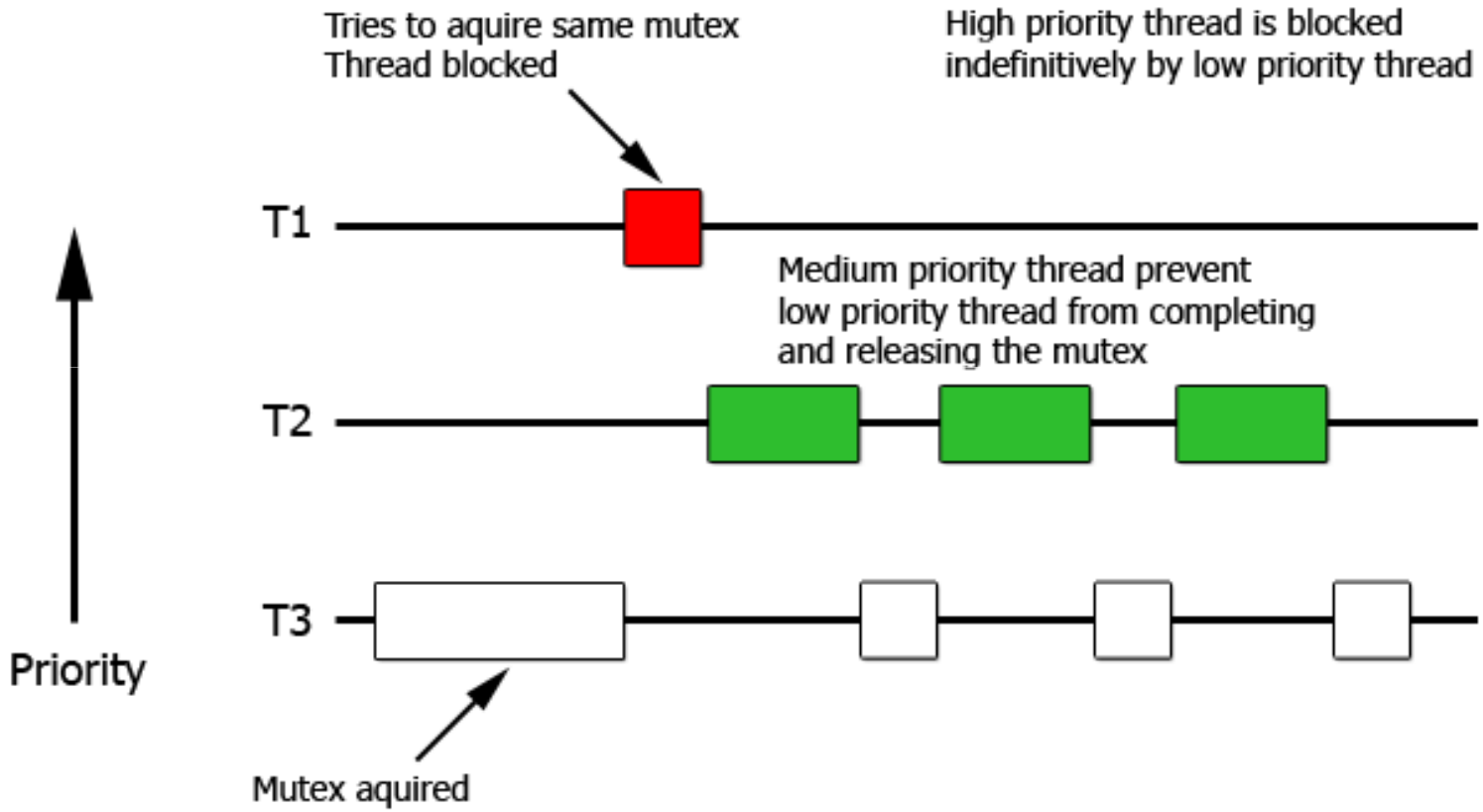
Threads – pthread_setschedparam()

- La fonction `pthread_setschedparam()` permet de changer la politique d'ordonnancement et la priorité d'un thread pendant son exécution

```
int pthread_setschedparam(pthread_t thread, int policy,
                          struct sched_param *param);
```

- `thread` : identificateur du thread
- `policy` : `SCHED_FIFO`, `SCHED_RR` ou `SCHED_OTHER`
- `param` : structure ayant un membre :
  - `int sched_priority` : priorité pour `SCHED_FIFO` ou `SCHED_RR`
- Retourne 0 ou un numéro d'erreur

Threads – Priority inversion



FORMATION

EUROGICIEL GROUPE

Threads – pthread_attr_*prioceiling()

- L'attribut `prioceiling` d'un mutex permet de specifier la priorité minimum d'un thread verrouillant ce mutex

```
int pthread_mutexattr_setprioceiling
                (pthread_mutexattr_t *attr, int prioceiling);
int pthread_mutexattr_getprioceiling
                (pthread_mutexattr_t *attr, int *prioceiling);
```

- `prioceiling` : priorité minimum (même valeurs que pour FIFO)
- Retourne 0 ou un numéro d'erreur

Threads – pthread_attr_*protocol()

- L'attribut `protocol` d'un mutex permet à un thread le verrouillant d'hériter de la priorité plus élevée d'un autre thread en attente sur le mutex

```
int pthread_mutexattr_setprotocol
            (pthread_mutexattr_t *attr, int protocol);
int pthread_mutexattr_getprotocol
            (pthread_mutexattr_t *attr, int *protocol);
```

- `protocol` :
  - `PTHREAD_PRIO_NONE` : pas de changement de priorité
  - `PTHREAD_PRIO_INHERIT` : thread verrouillant hérite de la priorité du thread en attente
  - `PTHREAD_PRIO_PROTECT` : thread verrouillant hérite de la plus haute des priorités (`prioceiling`) de tous les mutex de ce thread, même sans autre thread bloqué
- Retourne 0 ou un numéro d'erreur

Threads – pthread_attr_*stacksize()

- La fonction `pthread_attr_*stacksize()` permet de connaitre ou préciser la taille de la pile à la création d'un thread

```
int pthread_attr_getstacksize(pthread_attr_t *attr,
                              size_t *stacksize);
int pthread_attr_setstacksize(pthread_attr_t *attr,
                              size_t stacksize);
```

- `stacksize` : taille de la pile en octets
- Retournent 0 ou un numéro d'erreur

- La taille spécifiée par `pthread_attr_setstacksize()` sera la taille minimum allouée au thread

Threads – Autres fonctions

- `pthread_t pthread_self(void);`
  - Retourne l'ID du thread appelant

- `int pthread_equal(pthread_t t1, pthread_t t2);`
  - Retourne !0 si les deux IDs sont identiques, 0 sinon
  - Note : le type pthread_id est opaque, l'opérateur de comparaison == ne doit pas être utilisé

- `int pthread_once(pthread_once_t *once_control, void (*init_routine)(void));`
  - `pthread_once_t once_control = PTHREAD_ONCE_INIT;`
  - Appelle une seule fois la fonction `init_routine`

Sémaphores POSIX

Sémaphores Posix – Introduction

- Les sémaphores Posix ont un compteur d'une valeur entière positive ou nulle
  - Le compteur est incrémenté ou décrémenté de 1
  - La valeur maximum est SEM_VALUE_MAX
  - L'initialisation et l'utilisation est atomique
- Ils peuvent être utilisés pour la synchronisation de threads ou de processus
- Deux formes :
  - Anonymes
    - Résident en mémoire (variable globale ou segment de mémoire partagée)
  - Nommés :
    - Résident dans un système de fichiers virtuel (normalement `/dev/shm`)
    - Linux 2.6 et NPTL
- Les fonctions sont déclarées dans `<semaphore.h>`
- L'édition de liens se fait avec `-lrt` ou `-lpthread`

Sémaphores Posix – sem_init() et sem_destroy()

- La fonction `sem_init()` permet d'initialiser un sémaphore anonyme

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

- `*sem` : pointeur sur le type `sem_t`
- `pshared` : spécifie comment le sémaphore peut être partagé
  - `0` : partagé uniquement entre les threads d'un processus
    - `*sem` doit donc être visible de tous les threads (ex. variable globale)
  - `!0` : partagé entre les threads de différents processus
    - `*sem` doit donc être visible de tous les processus (ex. segment de mémoire partagée)
- `value` : valeur d'initialisation du sémaphore
- Retourne 0 ou -1 si erreur (avec `errno`)

- La fonction `sem_destroy()` permet de détruire un sémaphore

```
int sem_destroy(sem_t *sem);
```

- Retourne 0 ou -1 si erreur (avec `errno`)

Sémaphores Posix – sem_open()

- La fonction `sem_open()` permet d'initialiser un sémaphore nommé

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

- `*name` : nom du sémaphore (de la forme `/nom`)
- `oflag` : `O_CREAT` avec éventuellement `O_EXCL`
  - `O_CREAT` : crée le sémaphore si non existant, sinon ouvre le sémaphore
  - `O_EXCL` : retourne une erreur si un sémaphore de même nom existe déjà
- `mode` : permissions (voir `open()`)
- `value` : valeur initiale du compteur du sémaphore
- Retourne un pointeur du type `sem_t` ou `SEM_FAILED` si erreur (avec `errno`)

Sémaphores Posix – sem_close() et sem_unlink()

- Lorsqu'un processus a terminé d'utiliser un sémaphore nommé, il le ferme avec la fonction `sem_close()` (libération des ressources allouées à ce processus)

```
int sem_close(sem_t *sem);
```

- `*sem` : pointeur retourné par `sem_open()`
- Retourne 0 ou -1 si erreur (avec `errno`)

- Lorsque tous les processus ont terminé d'utiliser un sémaphore nommé, le sémaphore peut-être effacé du système avec la fonction `sem_unlink()`

```
int sem_unlink(sem_t *sem);
```

- `*sem` : pointeur retourné par `sem_open()`
- Retourne 0 ou -1 si erreur (avec `errno`)

- Le sémaphore est réellement détruit lorsque tout les processus l'ayant ouvert le ferment

Sémaphores Posix – sem_wait()

- La fonction `sem_wait()` verrouille un sémaphore
  - Si le sémaphore est supérieur à zéro, il est décrémenté et la fonction retourne
  - Sinon la fonction bloque tant que le sémaphore est égal à zéro

```
int sem_wait(sem_t *sem);
```

- `*sem` : pointeur sur le type `sem`
- Retourne 0 ou -1 si erreur (avec `errno`)

- Voir aussi les fonctions
  - `sem_trywait()` : retourne une erreur au lieu de bloquer
  - `sem_timedwait()` : limite le temps que la fonction peut bloquer
  - sem_getvalue() : récupère la valeur du compteur

Sémaphores Posix – sem_post()

- La fonction `sem_post()` déverrouille un sémaphore (il est incrémenté)

```
int sem_post(sem_t *sem);
```

- `*sem` : pointeur sur le type `sem`
- Retourne 0 ou -1 si la valeur du compteur dépasserait `SEM_VALUE_MAX` (avec `errno = EINVAL`)

Sockets

Sockets – Introduction

- Les sockets sont une interface système utilisée pour la communication entre processus, notamment à travers un réseau
- Cette API permet à un processus de spécifier un port et un protocole et d'y accéder pour émettre ou recevoir des données
- Couvre les protocoles AppleTalk, AX.25, IPX, NetRom, TCP/IP, IPC local, Netlink…
- Les ports privilégiés (en dessous de 1024) ne sont accessibles qu'aux processus avec un UID effectif de 0 (root)
- Les fonctions sur les sockets sont définies dans `<sys/socket.h>`

Sockets – Fichiers et commandes

- Le fichier `/etc/services` contient une liste des ports et services TCP et UDP communs
- Le fichier `/etc/host.conf` indique quels services (et leur ordre d'utilisation) seront utilisés pour la résolution de noms (DNS)
- Le fichier `/etc/resolv.conf` défini les domaines et adresses IP de serveurs DNS
- Le fichier `/etc/hosts` contient une table statique d'adresses IP et de noms symboliques
- Le fichier `/etc/protocols` contient une table de protocoles et leur numéros tels que définis par l'IANA

- Commandes utiles : `ifconfig`, `netstat`, `arp`, `rarp`, `route`

Sockets – Daemons

- Un démon (daemon) est un serveur qui fonctionne en permanence
- Il écoute un port / protocole spécifique et répond aux requêtes reçues

- `inetd` (ou `xinetd`) est un super-daemon chargé de l'écoute de plusieurs ports
- Il lance dynamiquement un processus serveur spécifique pour un port et un protocole donnés
- Il est configuré par `/etc/inetd.conf` (ou `/etc/xinetd.conf`)

Sockets – /proc/net

- L'interface `/proc/net` permet d'obtenir des statistiques réseau

- `/proc/net/dev`
  - Contient des informations sur les interfaces réseau (nombre de paquets, erreurs, collisions, etc)
- `/proc/net/tcp`, `/proc/net/udp`, `/proc/net/unix`, `/proc/net/raw`
  - Contiennent des informations pour chaque socket ouverte (adresses locale et distante, état, taille des queues, timers, UID, etc)
- `/proc/net/arp`, `/proc/net/rarp`, `/proc/net/route`
  - Contiennent les tables ARP et de routage
- `/proc/net/snmp`
  - Contient des statistiques protocoles (MIB-2 RFC)

- La plupart des adresses IP sont représentées en 4 octets hexadécimaux little-endian (ex : 0100007F:0017 -> 127 0 0 1 : 23)

Sockets – /proc/sys/net

- L'interface `/proc/sys/net` permet d'ajuster certains paramètres réseau

- `/proc/sys/net/core/(r|w)mem_(default|max)`
  - Tailles de buffer socket par défaut et maximum
- `/proc/sys/net/ipv4/tcp_timestamps`
- `/proc/sys/net/ipv4/tcp_window_scaling`
  - Ajoute les informations de temps et support de large TCP window (RFC 1323)
- `/proc/sys/net/ipv4/tcp_sack`
  - TCP selective acknoledgment
- `/proc/sys/net/ipv4/tcp_rmem`
- `/proc/sys/net/ipv4/tcp_wmem`
  - Taille de buffer réservée par connexion (mini, défaut, max)
- `/proc/sys/net/ipv4/tcp_fin_timeout`
- `/proc/sys/net/ipv4/tcp_keepalive_intvl`
- `/proc/sys/net/ipv4/tcp_keepalive_probes`
- `/proc/sys/net/ipv4/tcp_tw_recycle`
- `/proc/sys/net/ipv4/tcp_tw_reuse`

- Voir `Documentation/networking/ip-sysctl.txt` dans les sources du noyau

Sockets – Domaines

- Domaine Unix
  - Une socket peut être utilisé pour la communication entre processus d'une même machine
- Domaine Internet
  - Une socket peut être utilisé pour la communication entre processus de machines différentes
  - La socket est alors définie par
    - Le protocole
    - L'adresse IP de la machine A
    - Le port associé sur la machine A
    - L'adresse IP de la machine B
    - Le port associé sur la machine B
- Domaine Netlink
  - Système d'IPC entre l'espace noyau et l'espace user, pour transmettre des informations réseau (table de routage, de pare-feu, etc)

Sockets – Principaux types

- Type datagramme
  - Sockets en mode non connecté
  - Dans le domaine Internet, le protocole est UDP
  - Datagrammes de taille bornée
- Type flux (ou stream)
  - Socket en mode connecté
  - Dans le domaine Internet, le protocole est TCP
- Type raw
  - Fournit un accès bas-niveau direct avec la couche IP ou ICMP
  - L'UID effectif du processus doit être 0 (root)

Sockets – hton*() et ntoh*()

- Les protocoles fondés sur IP figent l'ordre des octets dans le réseau en big-endian (network byte-order)
- Sur une machine little-endian, il faut convertir l'ordre des données envoyées et reçues
- Il est de bonne pratique de systématiquement convertir les données quelque soit l'endianness de la machine (portabilité)

- Des fonctions de conversion sont déclarées dans `<netinet/in.h>`

```
unsigned long int htonl(unsigned long int val);
unsigned short int htons(unsigned short int val);
unsigned long int ntohl(unsigned long int val);
unsigned short int ntohs(unsigned short int val);
```

Sockets – sockaddr

- `sockaddr` est une structure définissant une adresse réseau
- La plupart des fonctions sur les sockets l'utilisent

```
struct sockaddr {
    unsigned short int sa_family;
    char sa_data[14];
};
```

- `sa_familly` : `AF_UNIX`, `AF_INET`, `AF_INET6`, `AF_IPX`, …
- `sa_data` : données propres au protocole

- Cette structure est générique (coquille vide)
- Chaque protocole utilise sa propre structure (`sockaddr_un`, `sockaddr_ipx`, `sockaddr_in6`, etc.)
- Dans la pratique on utilise une autre structure propre à un protocole, que l'on "cast" ensuite :

```
(struct sockaddr *) &sockaddr_un
```

Sockets – sockaddr_in

- Les structures adresses du domaine Internet IPv4 sont

```
struct in_addr {
    uint32_t s_addr;              // adresse IPv4
};
```

```
sockaddr_in {
    sa_family_t sin_family;       // toujours AF_INET
    in_port_t sin_port;           // port
    struct in_addr sin_addr;      // adresse IPv4
};
```

- L'adresse et le port doivent être sous la forme réseau (big-endian)
- L'adresse `INADDR_ANY` (0.0.0.0) dans `sin_addr.s_addr` associe une socket avec toutes les adresses IP de la machine
- La valeur 0 dans `sin_port` laisse le système choisir le port (client)

Sockets – sockaddr_in6

- Les structures adresse du domaine Internet IPv6 sont

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

```
sockaddr_in6 {
    sa_family_t sin6_family;      // toujours AF_INET6
    in_port_t sin6_port;          // port
    uint32_t sin6_flowinfo;       // information de flux
    struct in6_addr sin6_addr;    // adresse IPv6
    uint32_t sin6_scope_id;       // scope ID
};
```

- L'adresse et le port doivent être sous la forme réseau (big-endian)
- IPv6 partage les mêmes port qu'IPv4
- L'adresse in6addr_any associe une socket avec toutes les adresses IP de la machine (IPv4 et IPv6)

Sockets – inet_*()

- Les adresses IPv4 sont habituellement écrites sous la forme du chaine de caractères x.x.x.x (dotted quad)
- Des fonctions permettent de passer de cette forme à la forme réseau (big-endian) et vice-versa
- Il faut inclure les headers `<sys/socket.h>`, `<netinet/in.h>` et `<arpa/inet.h>`

```
int inet_aton(const char *cp, struct in_addr *inp);
char *inet_ntoa(struct in_addr in);
in_addr_t inet_addr(const char *cp);
int inet_pton(int af, const char *src, void *dst);
```

- NOTE : En cas d'erreur (adresse IP malformée), la fonction `inet_addr()` retourne `INADDR_NONE`, qui a pour valeur -1, qui est une valeur d'adresse IP légale (255.255.255.255)

Sockets – inet_pton()

- La fonction `inet_pton()` sert à convertir une adresse IPv4 ou IPv6 sous forme de chaine de caractères en une structure adresse

```
int inet_pton(int af, const char *src, void *dst);
```

- `af` : `AF_INET` ou `AF_INET6`
- `*src` : adresse IP :
  - IPv4 : "xxx.xxx.xxx.xxx"
  - IPv6 : "xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx:xxxx"
- `*dst` : structure adresse :
  - IPv4 : `struct in_addr`
  - IPv6 : `struct in6_addr`
- Retourne 1 si conversion réussie, 0 si `*src` ne représente pas une adresse valide, -1 si `af` ne contient pas une famille valide

Sockets – inet_ntop()

- La fonction `inet_ntop()` sert à convertir une adresse IPv4 ou IPv6 d'une structure adresse en une chaine de caractères

```
char *inet_ntop(int af, const void *restrict src,
                char *restrict dst, socklen_t size);
```

- `af` : `AF_INET` ou `AF_INET6`
- `*src` : structure adresse :
  - IPv4 : `struct in_addr`
  - IPv6 : `struct in6_addr`
- `*dst` : buffer qui contiendra l'adresse IP
- `size` : taille du buffer `dst` (`INET_ADDRSTRLEN` ou `INET6_ADDRSTRLEN`)
- Retourne un pointeur sur le buffer si conversion réussie, `NULL` sinon (avec `errno`)

Sockets – gethostbyname()

- La fonction `gethostbyname()`, déclarée dans `<netdb.h>`, permet de faire une résolution de nom DNS vers une adresse IPv4 ou IPv6

```
struct hostent *gethostbyname(const char *name);
```

- `name` : nom de domaine ou adresse IP
- Retourne un pointeur sur `hostent` ou `NULL` si erreur (avec la variable `h_errno`)

- La structure `hostent` :

```
struct hostent {
    char  *h_name;          nom DNS officiel
    char **h_aliases;       liste d'alias (terminée par NULL)
    int    h_addrtype;      AF_INET ou AF_INET6
    int    h_length;        longueur d'une adresse en octets
    char **h_addr_list;     listes des adresses (terminée par NULL)
};
```

- NOTE : cette fonction n'est pas réentrante

FORMATION

Sockets – gethostbyaddr()

- La fonction `gethostbyaddr()`, déclarée dans `<netdb.h>`, permet de faire une résolution inverse d'une adresse IP vers un nom de domaine

```
struct hostent *gethostbyaddr(const void *addr, int len,
                              int type);
```

- `addr` : pointeur sur une structure adresse (sin_addr pour `AF_INET`)
- `len` : longueur de la structure `addr` (4 pour IPv4, 16 pour IPv6)
- `type` : `AF_INET` ou `AF_INET6`
- Retourne un pointeur sur `hostent` ou `NULL` si erreur (avec la variable `h_errno`)

- NOTE : cette fonction n'est pas réentrante

Sockets – getaddrinfo()

- La fonction `getaddrinfo()` est l'équivalent (réentrant donc thread-safe) de `gethostbyname()` et fonctionne pour IPv4 et IPv6

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

- `*node` : nom de domaine (ex. "www.exemple.com") ou adresse IP
- `*service` : nom du service (ex. "http") ou port
- `*hints` : structure pré-remplie (voir ci-après) ou `NULL`
- `*res` : liste chainée (voir ci-après)
- Retourne 0 ou un code d'erreur

- La liste chainée `*res` doit être libérée avec :
  `freeaddrinfo(struct addrinfo *res);`

- Voir `getnameinfo()` pour la fonction inverse

Sockets – addrinfo

- `addrinfo` est une structure utilisée par la fonction `getaddrinfo()`

```
struct addrinfo {
    int ai_flags;          // AI_PASSIVE, AI_CANONNAME, etc
    int ai_family;         // AF_INET, AF_INET6, AF_UNSPEC
    int ai_socktype;       // SOCK_STREAM, SOCK_DGRAM
    int ai_protocol;       // souvent 0
    size_t ai_addrlen;     // taille de ai_addr
    struct sockaddr *ai_addr;     // sockaddr_in or _in6
    char *ai_canonname;    // canonical hostname
    struct addrinfo *ai_next;     // linked list, next node
};
```

- AF_UNSPEC permet de s'affranchir de la version IP

Sockets – getaddrinfo() – Exemple

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

int status;
struct addrinfo *servinfo;

if ((status = getaddrinfo("www.example.com", NULL, NULL, &servinfo)) !=
   0) {
    fprintf(stderr, "getaddrinfo error: %s\n", gai_strerror(status));
    exit(1);
}

// le domaine www.example.com est résolu, servinfo pointe sur une liste
   chainée d'une ou plusieurs struct addrinfos

// ... do work....

freeaddrinfo(servinfo);           // libère la liste chainée
```

Sockets – socket()

- La fonction `socket()` permet de créer une socket nécessaire pour la communications entre processus (locaux ou distants)

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

- `domain` : domaine de socket (`AF_UNIX`, `AF_INET`, `AF_INET6`, …)
- `type` : type de socket
  - `SOCK_STREAM` : TCP
  - `SOCK_DGRAM` : UDP
  - `SOCK_RAW` : raw IP
  - `SOCK_PACKET` : Link-Layer frames
- `protocol` : numéro du protocole utilisé, ou 0 pour le protocole par défaut
- Retourne un descripteur de fichier ou -1 si erreur (avec `errno`)

- Le descripteur de fichier retourné est utilisable par les fonctions telles que `read()` et `write()`

- La fonction `close()` permet de fermer le descripteur d'une socket (voir la page sur `shutdown()`)

Sockets – Sockets type datagramme

- Socket du type `SOCK_DGRAM`
- Protocole UDP dans le domaine `AF_INET`
- Aucune vérification qu'un message envoyé est bien reçu
- Les messages peuvent être reçu dans un ordre différent de l'ordre d'envoi
- Aucun contrôle de flux
- Aucun contrôle de congestion
- Checksum à la fin du message
- Deux modes de fonctionnement :
  - Socket non connectée :
    - Chaque message est envoyé ou reçu potentiellement à ou depuis une adresse différente
  - Socket connectée :
    - Le fonction `connect()` est utilisée avant l'envoi ou la réception de messages (avec `write()` et `read()` par exemple)

Sockets – sendto() et recvfrom()

- Les fonctions `sendto()` et `recvfrom()` permettent d'envoyer ou recevoir un message par une socket, en mode `SOCK_DGRAM`
- Les adresses de destination et provenance sont donc spécifiées (si socket non connectée)

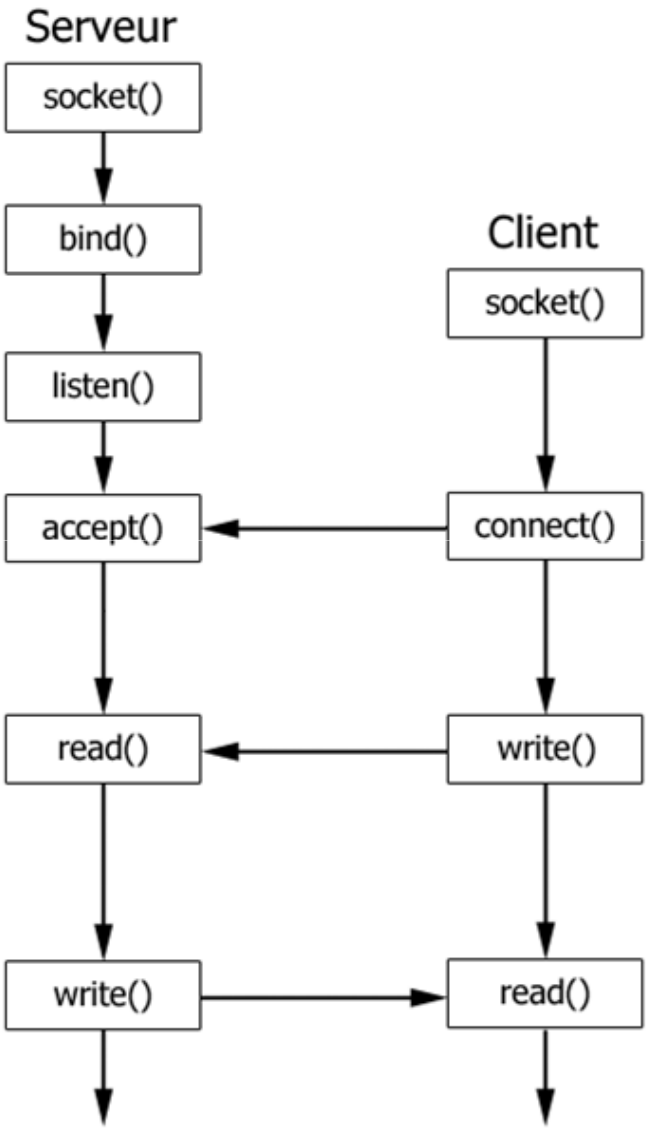
```
ssize_t sendto(int sockfd, const void *buf, size_t len, int
   flags, const struct sockaddr *addr, socklen_t addrlen);
ssize_t recvfrom(int sockfd, void *buf, size_t len, int
   flags, struct sockaddr *src_addr, socklen_t *addrlen);
```

- `sockfd` : descripteur de socket
- `buf` : buffer
- `len` : taille en octets de `buf`
- `flags` : options (normalement 0)
- `addr` : adresse du destinataire ou de l'émetteur
- `addr_len` : taille en octets de `addr`
- Retournent le nombre d'octets lus ou écrits ou -1 si erreur (avec `errno`)

Sockets – Streams

- Socket du type `SOCK_STREAM`
- Protocole TCP dans le domaine `AF_INET`
- Transport fiable, vérification qu'un message envoyé est bien reçu
- Contrôle de flux
- Contrôle de congestion
- Checksum
- L'établissement de la connexion entre machines se fait en plusieurs étapes
- La dissymétrie entre le serveur et le client est clairement marquée
  - Le serveur doit être en attente de connexion
  - Lorsque le serveur reçoit une connexion, une socket "de service" est créée
  - Le client prend l'initiative d'une connexion

Sockets – Streams : Client / Serveur



Sockets – bind()

- La fonction `bind()` permet d'attacher une sockets à une adresse
- Doit être utilisée avant de pouvoir recevoir des connexions

```
int bind(int sockfd, const struct sockaddr *addr, socklen_t
                                                  addr_len);
```

- `sockfd` : descripteur de socket
- `addr` : pointeur sur une structure d'adresse
- `addr_len` : taille en octets de la structure `addr`
- Retourne 0 ou -1 si erreur (avec `errno`)

- La structure `addr` employée dépend de la famille d'adresse utilisée (ex : `sockaddr_in` pour une adresse IPv4)

Sockets – listen()

- La fonction `listen()` permet à un serveur de créer une file d'attente de connexions en attente

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

- `sockfd` : descripteur de la socket
- `backlog` : longueur de la file d'attente
- Retourne 0 ou -1 si erreur (avec `errno`)

- Lorsque la file d'attente est pleine, les connections suivantes sont refusées

Sockets – accept()

- La fonction `accept()` permet a un serveur d'accepter une connexion en attente dans la file crée par `listen()`

```
int accept(int sockfd, struct sockaddr *addr, socklen_t
                                                addr_len);
```

- `sockfd` : descripteur de socket
- `addr` : adresse du client se connectant
- `addr_len` : taille en octets de la structure `addr`
- Retourne un descripteur sur une nouvelle socket ou -1 si erreur (avec `errno`)
  - Le descripteur original continue d'être utilisé pour les connexions suivantes

- S'il n'y a pas de connexion pendante, la fonction bloque (sauf si la socket est en mode non bloquant)

Sockets – connect()

- La fonction `connect()` permet a un client d'attenter une connexion vers un serveur

```
int connect(int sockfd, const struct sockaddr *addr,
                                        socklen_t addr_len);
```

- `sockfd` : descripteur de socket
- `addr` : adresse du serveur
- `addr_len` : taille en octets de la structure `addr`
- Retourne 0 ou -1 si erreur (avec `errno`)

- La fonction `connect()` bloque jusqu'à ce que le serveur accepte la connexion (ou timeout dépendant du protocole) (sauf si la socket est en mode non bloquant)

Sockets – connect() (suite)

- La fonction `connect()` peut être utilisé sur une socket `SOCK_DGRAM`
  - Doit être utilisée après `bind()` mais avant toute lecture ou écriture sur la socket
- Cela n'implique pas une connexion comme avec `SOCK_STREAM`
- Tous les paquets envoyés le seront vers l'adresse spécifiée
- Seuls les paquets venant de l'adresse spécifiée seront reçus
- Il est donc ensuite possible d'utiliser `read()` et `write()` au lieu de `recvfrom()` et `sendto()`
- Il est possible d'utiliser plusieurs fois `connect()` pour changer d'association
- Une association peut être rompue en utilisant `AF_UNSPEC`

Sockets – Streams – Exemple client

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

void tcp_client(void) {
    int sockfd;
    struct sockaddr_in address;
    char ch = 'A';

    sockfd = socket(AF_INET, SOCK_STREAM, 0);

    address.sin_family = AF_INET;
    address.sin_addr.s_addr = inet_addr("127.0.0.1");
    address.sin_port = htons(9734);
    connect(sockfd, (struct sockaddr *)&address, sizeof(address));
    write(sockfd, &ch, 1);
    read(sockfd, &ch, 1);
    printf("char from server = %c\n", ch);
    close(sockfd);
}
```

Sockets – Streams – Exemple serveur

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

void tcp_server(void) {
   int server_sockfd, client_sockfd, client_len;
   struct sockaddr_in server_address;
   struct sockaddr_in client_address;
   char ch;

   server_sockfd = socket(AF_INET, SOCK_STREAM, 0);

   server_address.sin_family = AF_INET;
   server_address.sin_addr.s_addr = htonl(INADDR_ANY);
   server_address.sin_port = htons(9734);
   bind(server_sockfd, (struct sockaddr *)&server_address,
                                          sizeof(server_address));
   listen(server_sockfd, 5);
   client_len = sizeof(client_address);
```

Sockets – Streams – Exemple serveur (suite)

```
    while(1) {
        client_sockfd = accept(server_sockfd, (struct sockaddr *)
                                        &client_address, &client_len);
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

Sockets – send() et recv()

- Les fonctions `send()` et `recv()` permettent d'envoyer et recevoir des données à travers une socket en mode `SOCK_STREAM`

```
int send(int sockfd, const void *msg, int len, int flags);
int recv(int sockfd, void *msg, int len, int flags);
```

- `sockfd` : descripteur de socket
- `*msg` : buffer de données à envoyer ou reçues
- `len` : taille du buffer
- `flags` : OU binaire entre :
  - `MSG_DONTROUTE` : **pas de routage à travers un gateway**
  - `MSG_DONTWAIT` : **opération non bloquante**
  - `MSG_OOB` : **message out-of-band (flag TCP** `URG`**)**
- Retourne le nombre d'octets envoyés ou lus ou -1 si erreur (avec `errno`)

- Les fonctions `send()`, `recv()`, `write()` et `read()` peuvent être interrompues par un signal (`errno` est égal à `EINTR`)

Sockets – Principales options des sockets

SOL_SOCKET	Description
SO_REUSADDR	Permet de réutiliser une socket immédiatement
SO_KEEPALIVE	Envoie des messages keep-alive (streams)
SO_LINGER	Délai entre messages non envoyés et retour de close()
SO_BROADCAST	Permet l'envoie ou reçoit des datagrammes broadcast
SO_SNDBUF	Taille du buffer d'émission
SO_RCVBUF	Taille du buffer de réception
SO_TYPE	Permet de lire le type de socket
SO_DONTROUTE	N'envoie pas de données à travers un gateway
TCP_NODELAY	Désactive l'algorithme de Nagle

Sockets – setsockopt()

- La fonction `setsockopt()` permet de spécifier des options d'une socket

```
int setsockopt(int sockfd, int level, int optname,
                           void *optval, socklen_t optlen);
```

- `sockfd` : descripteur de socket
- `level` : niveau de protocole (`SOL_SOCKET`, `SOL_TCP`, `IPPROTO_TCP`, …)
- `optname` : nom de l'option
- `optval` : valeur de l'option
- `optlen` : longueur de la valeur en octets
- Retourne 0 ou -1 si erreur (avec `errno`)

- Typiquement, `setsockopt()` s'utilise avant `bind()`

- La fonction `getsockopt()` permet de retrouver la valeur d'une option particulière

Sockets – setsockopt() – Multicast

- La fonction `setsockopt()` permet de joindre ou quitter un groupe multicast

```
setsockopt(sockfd, IPPROTO_IP, IP_ADD_MEMBERSHIP,
                   struct ip_mreq *mreq, sizeof (ip_mreq));
setsockopt(sockfd, IPPROTO_IP, IP_DROP_MEMBERSHIP,
                   struct ip_mreq *mreq, sizeof (ip_mreq));
```

- ```
  struct ip_mreq {
      struct in_addr imr_multiaddr; // Adresse IP multicast
      struct in_addr imr_interface; // Adresse IP locale
  };
  ```

- En multicast, il est nécessaire d'autoriser l'utilisation répétée d'une adresse IP :

```
char one = 1;
setsockopt(sd, SOL_SOCKET, SO_REUSEADDR, &one, sizeof(char));
```

FORMATION
```

Sockets – shutdown()

- La fonction `shutdown()` permet de spécifier comment une connexion full-duplex devrait être terminée

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

- `sockfd` : descripteur de socket
- `how` :
  - `SHUT_RD` : ferme en réception
  - `SHUT_WR` : ferme en émission
  - `SHUT_RDWR` : ferme en réception et en émission
- Retourne 0 ou -1 si erreur (avec `errno`)

- La fonction `close()` peut créer l'envoi d'un `RST` s'il reste des données dans les buffers (perte de données), même avec l'option `SO_LINGER`
- `shutdown()` vide les buffers et informe le correspondant de la fin de la connexion (envoi d'un `FIN`)

- La socket devra être fermée avec `close()`, après que `read()` retourne 0

Sockets – shutdown() – Exemple

```
sock = socket(AF_INET, SOCK_STREAM, 0);
connect(sock, &remote, sizeof(remote));
write(sock, buffer, 1000000);

shutdown(sock, SHUT_WR);

while(1) {
   res=read(sock, buffer, 4000);
   if(res < 0) {
      perror("reading");
      exit(1);
   }
   if(!res)
      break;
}
close(sock);
```

Sockets – Multiplexage

- Le multiplexage permet à un serveur de traiter plusieurs connexions clientes
- Plusieurs implémentations :
  - Utiliser un processus par client (`fork()` après `accept()`)
  - Utiliser une réserve de processus (plusieurs `fork()` avant `accept()`)
  - Utiliser un thread par client (`pthread_create()` après `accept()`)
  - Utiliser une réserve de thread (`pthread_create()` avant `accept()`)
  - Utiliser une réserve de sockets (`select()`)

Sockets – Multiplexage – Exemple forking

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

/* socket(), bind() et listen() ont été appelées à ce point */

while(1) {
   new_sock = accept(server_sock, NULL, NULL);
   if ((pid = fork()) == 0) {
        /* child process */
        close(server_sock);
        nread = read(new_sock, buffer, 25, 0);
        do_stuff();
        close(new_sock);
        exit(EXIT_SUCCESS);
   } else {
        /* parent process */
        close(new_sock);
   }
}
```

Sockets – Multiplexage – Exemple pre-forking

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

/* socket(), bind() et listen() ont été appelées à ce point */

for (x = o; x < MAX_CHILDREN; x++) {
    if ((pid = fork()) == 0) {
        /* child process */
        while(1) {
            new_sock = accept(server_sock, NULL, NULL);
            nread = read(new_sock, buffer, 25, 0);
            do_stuff();
            close(new_sock);
        }
    }
}
```

Sockets – Multiplexage – Exemple threaded

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

/* socket(), bind() et listen() ont été appelées à ce point */

while(1) {
   new_sock = accept(server_sock, NULL, NULL);
   pthread_create(&thread_id, NULL, &thread_proc, (void *) new_sock);
}

void thread_proc(void *arg)
{
   int sock = (int) arg;
   int nread = read(new_sock, buffer, 25, 0);
   do_stuff();
   close(sock);
}
```

Sockets – Multiplexage – select()

- La fonction `select()` permet de surveiller (attente d'un événement) plusieurs descripteurs (donc sockets) à la fois
- La fonction distingue trois ensembles de descripteurs à surveiller
  - Lecture
  - Ecriture
  - Exceptions
- En retour, la fonction ne garde dans chaque ensemble que les descripteurs ayant eu un événement

- Les ensembles de descripteurs sont placés dans une variable `fd_set`

- Des macros permettent de manipuler les variables `fd_set` :

```
FD_SET(int fd, fd_set *set);    // Ajoute fd à l'ensemble
FD_CLR(int fd, fd_set *set);    // Enlève fd de l'ensemble
FD_ISSET(int fd, fd_set *set);  // Vrai si fd est dans l'ensemble
FD_ZERO(fd_set *set);           // Efface l'ensemble
```

- Un ensemble ne peut contenir plus de `FD_SETSIZE` descripteurs

Sockets – select()

- La fonction `select()` d'attendre un ou des événements sur des ensembles de descripteurs de fichiers
- Elle est prototypée dans <sys/select.h>

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

- n : descripteur le plus élevé des trois ensembles + 1
- `readfds`, `writefds`, `exceptfds` : ensembles de descripteurs à surveiller
- `timeout` : délai d'attente ou `NULL` pour infini
- Retourne le nombre de descripteurs modifiés, 0 si timeout ou -1 si erreur (avec `errno`)

- La fonction `select()` peut être utilisée avec tout type de descripteur (socket, pipe, fichier)
- L'établissement d'une nouvelle connexion sur une socket est signalé dans l'ensemble de lecture
- Un descripteur sur fichier régulier est toujours prêt en lecture si pas à la fin du fichier, et est toujours prêt en écriture

Sockets – select() – Exemple 1

```
#include <stdio.h>
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h>

int main(void)
{
    struct timeval tv;
    fd_set readfds;

    tv.tv_sec = 2;
    tv.tv_usec = 500000;

    FD_ZERO(&readfds);
    FD_SET(STDIN, &readfds);

    select(STDIN_FILENO + 1, &readfds, NULL, NULL, &tv);

    if (FD_ISSET(STDIN, &readfds))
        printf("A key was pressed!\n");
    else
        printf("Timed out.\n");

    return 0;
}
```

Sockets – select() – Exemple 2

```
/* socket(), bind() et listen() ont été appelées à ce point */

fd_set readset;

do {
   FD_ZERO(&readfds);
   FD_SET(sockfd, &readfds);
   result = select(sockfd + 1, &readfds, NULL, NULL, NULL);
} while (result == -1 && errno == EINTR);
if (result > 0) {
   if (FD_ISSET(sockfd, &readfds)) {
      /* Data available to be read */
      result = recv(sockfd, some_buffer, some_length, 0);
      if (result == 0) {
         close(socket_fd);    // Other side closed its socket
      }
      else {
         /* Do work */
      }
   }
}
else if (result < 0) {
   /* An error ocurred, just print it to stdout */
   printf("Error on select(): %s\", strerror(errno));
}
```

Sockets – select() – Exemple 3

```
fd_set read_set, test_set;

FD_ZERO(&read_set);
FD_SET(sockfd, &read_set)
max_sockfd = sockfd;

while(1) {
   test_set = read_set;
   if (1 <= select(max_sockfd + 1, &test_set, NULL, NULL, NULL)) {
      for (var x = 0; x < max_sockfd; x++) {
         if (FD_ISSET(x, &test_set)) {
            if (x == sockfd) {
               int new_sock = accept(sockfd, NULL, NULL);
               FD_SET(new_sock, &read_set);
               max_sockdf = max(max_sockdf, new_sock)
            } else {
               nread = read(x, buffer, 25, 0)
               do_stuff();
               close(x);
               FD_CLR(x, $read_set);
            }
         }
      }
   }
}
```

Sockets – poll()

- La fonction `poll()` s'apparente à `select()`

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

- `*fds` : tableau de structure (voir ci-après):

```
struct pollfd {
    int fd;             // descripteur de fichier
    short events;       // événements à surveiller
    short revents;      // événements retournés
};
```

- `nfds` : nombre de structures dans `fds`
- `timeout` : délai d'attente ou `NULL` pour infini
- Retourne le nombre de structures avec événements, 0 si timeout ou -1 si erreur (avec `errno`)

FORMATION

Sockets – poll() (suite)

- Les événements sont définis avec un OU binaire de :
  - `POLLIN` : **données reçues**
  - `POLLPRI` : **données urgentes reçues (TCP OOB)**
  - `POLLOUT` : **écrire ne bloquerait pas**
  - `POLLRDHUP` : **l'autre coté de connexion a fermé sa socket**
  - `POLLHUP` : **déconnection de la connexion**
  - `POLLERR` : **condition d'erreur**
  - `POLLNVAL` : **descripteur invalide**

Sockets – poll() – Exemple

```
struct pollfd fds[2];
int ret;
int i;

fds[0].fd = open("/dev/dev0", ...);
fds[1].fd = open("/dev/dev1", ...);
fds[0].events = POLLOUT | POLLERR;
fds[1].events = POLLOUT | POLLERR;

if ((ret = poll(fds, 2, timeout_msecs)) > 0) {
   for (i=0; i<2; i++) {
      if (fds[i].revents & POLLERR) {
         /* handle error */
      }
      if (fds[i].revents & POLLOUT) {
         /* do work */
      }
   }
}
```

Sockets – Mode non bloquant

- Les sockets peuvent être configurées en mode non bloquant
- Les fonctions (`accept()`, `connect()`, `read()`, `write()`, etc) qui normalement bloquent retournent immédiatement
  - Si l'opération n'a pu réussir, elles retournent -1 et `errno` est égal à `EAGAIN` (or `EWOULDBLOCK`)

- La fonction `fcntl()` est utilisée pour configurer une socket en mode non bloquant :

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

Sockets – Domaine Unix

- Les sockets du domaine Unix sont similaires à un tube nommé
- L'adresse utilisée pour `bind()` est un nom de fichier
- Les sockets créées dans le système de fichiers ne peuvent être utilisées qu'avec l'API des sockets
- Le fichier socket ne doit pas exister avant la création d'une nouvelle socket (le supprimer avec `unlink()`)

- La structure adresse du domaine Unix est
  - ```
    struct sockaddr_un {
      sa_family_t sun_family;
      char sun_path[];
    };
    ```

    Domaine : `AF_UNIX` ou `AF_LOCAL`

    Nom de fichier

FORMATION
    ```

Sockets – Domaine Unix – Exemple client

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

#define SOCKET_FILE "/tmp/mysocket"

void client(void) {
   int sockfd;
   struct sockaddr_un address;
   char ch = 'A';

   sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

   address.sun_family = AF_UNIX;
   strcpy(address.sun_path, SOCKET_FILE);
   connect(sockfd, (struct sockaddr *)&address, sizeof(address));
   write(sockfd, &ch, 1);
   read(sockfd, &ch, 1);
   printf("char from server = %c\n", ch);

   close(sockfd);
}
```

Sockets – Domaine Unix – Exemple serveur

```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

#define SOCKET_FILE "/tmp/mysocket"

void server(void) {
   int server_sockfd, client_sockfd, client_len;
   struct sockaddr_un server_address;
   struct sockaddr_un client_address;
   char ch;

   unlink(SOCKET_FILE);

   server_sockfd = socket(AF_UNIX, SOCK_STREAM, 0);

   server_address.sun_family = AF_UNIX;
   strcpy(server_address.sun_path, SOCKET_FILE);
   bind(server_sockfd, (struct sockaddr *)&server_address,
                                          sizeof(server_address));
```

Sockets – Domaine Unix – Exemple serveur (suite)

```
    listen(server_sockfd, 5);
    client_len = sizeof(client_address);

    while(1) {
        client_sockfd = accept(server_sockfd, (struct sockaddr *)
                                &client_address, &client_len);
        read(client_sockfd, &ch, 1);
        ch++;
        write(client_sockfd, &ch, 1);
        close(client_sockfd);
    }
}
```

Sockets – socketpair()

- La fonction `socketpair()` permet de créer une paire de sockets connectées, full-duplex
- L'utilisation est similaire à `pipe()`

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

- `dom` : domaine, toujours `AF_UNIX` ou `AF_LOCAL`
- `type` : `SOCK_STREAM`
- `protocol` : normalement 0
- `sv` : tableau de descripteur de sockets
- Retourne 0 ou -1 si erreur (avec `errno`)

Sockets – socketpair() – Exemple

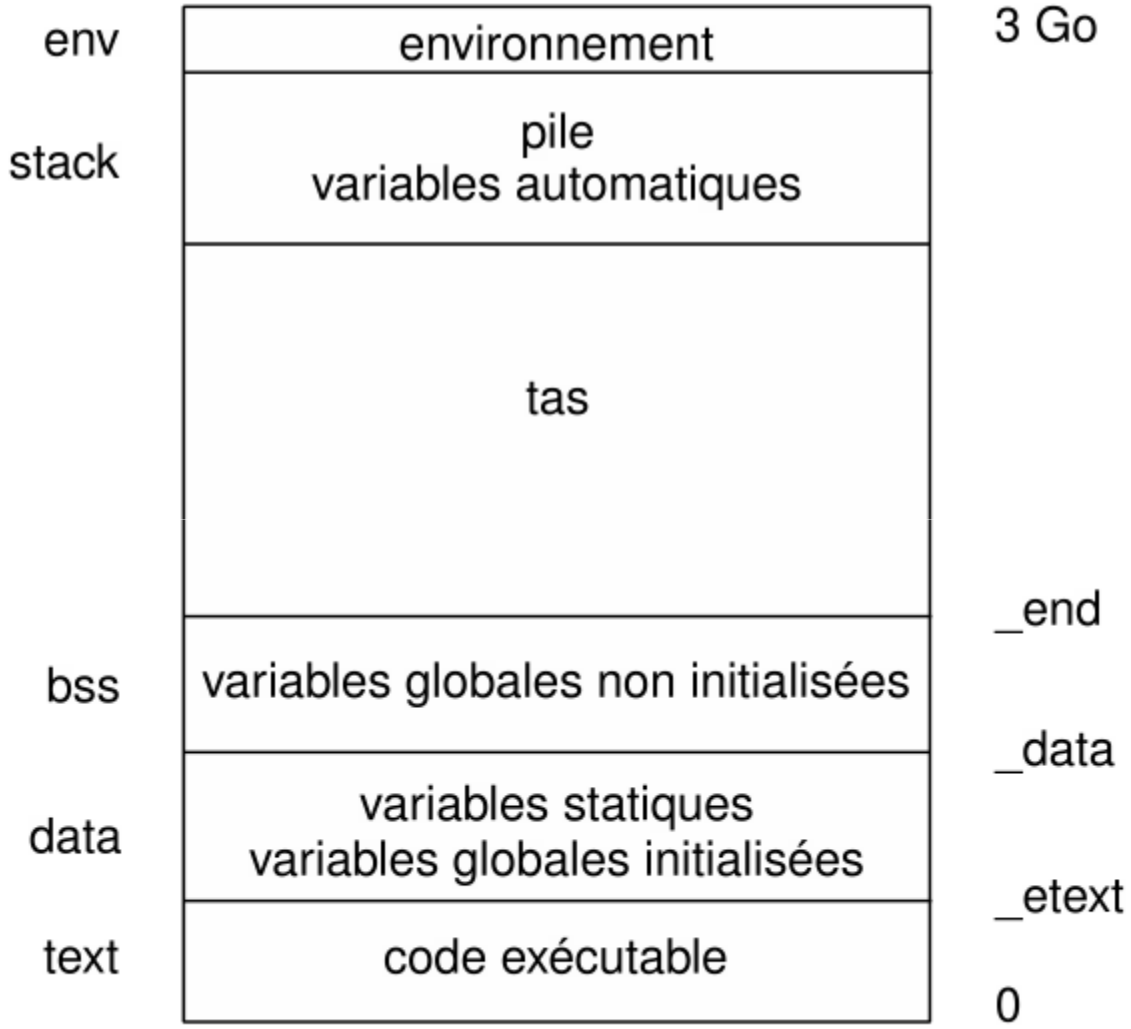
```
/* Les conditions d'erreurs ne sont pas vérifiées dans cet exemple !!! */

socketpair(AF_UNIX, SOCK_STREAM, 0, sockets);

child = fork();
if (child) {
   /* This is the parent. */
   close(sockets[0]);
   read(sockets[1], buf, 1024, 0);
   printf("-->%s\n", buf);
   write(sockets[1], DATA2, sizeof(DATA2));
   close(sockets[1]);
} else {
   /* This is the child. */
   close(sockets[1]);
   write(sockets[0], DATA1, sizeof(DATA1));
   read(sockets[0], buf, 1024, 0);
   printf("-->%s\n", buf);
   close(sockets[0]);
}
```

Gestion Mémoire

Mémoire – Règles

- Restreindre les blocs alloués dynamiquement qu'on ne libère pas à la fonction `main()` (toute la mémoire du processus est libérée)
- A chaque déclaration d'un pointeur, l'initialiser avec `NULL`
- Avant d'invoquer `malloc()`, vérifier que le pointeur est `NULL`
- Avant de libérer un pointeur, vérifier qu'il n'est pas `NULL`
- Vérifier les erreurs à tout appel de `malloc()`
  - `calloc(0)` et `malloc(0)` ne retourne pas toujours `NULL`
- Dès qu'un pointeur est libéré, le recharger avec `NULL`
- Ne pas oublier la place de `\0` lors d'un `malloc()` pour une chaine
  - Voir les fonctions `strlcat()` et `strlcpy()` d'OpenBSD

Mémoire – Espace mémoire processus



Mémoire – Espace mémoire processus (suite)

- `text` :
  - Contient le code exécutable du processus
  - Contient les routines des bibliothèques partagées utilisées par le processus
  - S'étend jusqu'à l'adresse contenue dans la variable `_etext`
- `data` :
  - Contient les données initialisées au chargement du processus
  - Contient les données locales statiques des fonctions
  - S'étend de l'adresse contenue dans la variable `_etext` jusqu'à celle contenue dans `_data`
- `bss` :
  - Contient les données non initialisées et des données allouées dynamiquement
  - S'étend de l'adresse contenue dans `_data` à celle contenue dans `_end`

Mémoire – Espace mémoire processus (suite)

- La taille maximale de la pile d'un processus est fixée par `RLIMIT_STACK`
- La fonction `getrlimit()` retourne la valeur de `RLIMIT_STACK`
- Voir aussi la commande `ulimit -s`

- Un processus dont la pile dépasse la limite reçoit un signal `SIGSEGV` (segmentation fault)

- La commande `ps` un permet de connaitre pour un processus la taille de mémoire virtuelle utilisée (VSZ) et la place utilisée en mémoire physique (RSS)

- L'outil `pmap` (ou `/proc/<pid>/maps`) montre la carte mémoire d'un processus

Mémoire – Over commit-memory

- `malloc()` augmente la taille du segment `bss`
  - `malloc()` change la position de `_end` avec les fonction `brk()` et `sbrk()`
- La place effective dans la mémoire physique n'est pas tout de suite allouée par le noyau
- Seulement lorsque le processus tente d'écrire dans la zone allouée qu'une page mémoire réelle est attribuée par le noyau
- Il est donc possible de réclamer beaucoup plus d'espace mémoire que ce que le système peut réellement fournir
- Lorsque toute la mémoire système est saturée, le noyau cherche le processus responsable et le termine (Out of memory killer)
  - Voir `Documentation/vm/overcommit-accounting`
- Il est donc recommandé d'écrire dans la zone mémoire après chaque `malloc()` ou `calloc()` avec `memset()` par exemple

Mémoire – alloca()

- La fonction `alloca()` est semblable à `malloc()` mais alloue un bloc mémoire dans la pile (au lieu du segment `bss`)

```
void * alloca (size_t size);
```

- `size` : taille en octets
- Retourne un pointeur sur la zone allouée (jamais `NULL`)

- Le bloc mémoire est donc automatiquement libéré lorsque la fonction invoquant `alloca()` retourne
  - Seules des variables utilisées dans la fonction ou dans des sous routines peuvent donc être allouées

- Le noyau alloue automatiquement les pages nécessaires pour la pile (donc pas de risque de over committed memory)

- Si l'allocation échoue, soit le système n'a plus de mémoire libre ou la pile a dépassé la limite `RLIMIT_STACK`

Mémoire – mlock()

- Un processus peut voir une page de mémoire allouée déplacée en swap si le noyau a besoin de recouvrer de la mémoire physique
- Le rechargement d'une page mémoire lors d'une faute de page est lent
- Des données confidentielles peuvent être accessible sur le périphérique de swap
- Les fonctions `mlock()` et `mlockall()` permettent à un processus de verrouiller une page mémoire ou l'ensemble de son espace d'adressage

```
int mlock(const void *addr, size_t len);
int munlock(const void *addr, size_t len);
int mlockall(int flags);
int munlockall(void);
```

- `addr` : adresse de la zone mémoire à verrouiller
- `len` : taille de la zone (le noyau arrondi à la frontière de page inférieure)
- `flags` : `MCL_CURRENT` et/ou `MCL_FUTURE`
- Retournent 0 ou -1 si erreur (avec `errno`)

- Le processus doit avoir un UID effectif de 0
- Le verrouillage d'une page mémoire n'est pas hérité lors d'un `fork()`

Mémoire – mprotect()

- La fonction `mprotect()` permet de limiter l'accès à une page mémoire, en lecture, écriture ou en exécution

```
int mprotect(const void *addr, size_t len, int prot);
```

- `addr` : adresse de la zone mémoire à verrouiller, doit être aligné sur une frontière de page
- `len` : taille de la zone
- `prot` : soit `PROT_NONE` ou un OU binaire entre
  - `PROT_READ`
  - `PROT_WRITE`
  - `PROT_EXEC`
- Retourne 0 ou -1 si erreur (avec `errno`)

- La zone mémoire à protéger doit être alignée sur une frontière de page
- Si un processus essaie un accès illégal à une page protégée, il reçoit un signal `SIGSEGV`
- La taille d'une page est obtenue avec `sysconf(_SC_PAGE_SIZE);`

Mémoire – mtrace() et muntrace()

- La fonction `mtrace()` de Glibc permet de suivre les allocations et libérations de mémoire (`malloc()`, `calloc()`, `realloc()` et `free()`)
- La fonction `muntrace()` arrête le suivi

```
void mtrace(void);
void muntrace(void);
```

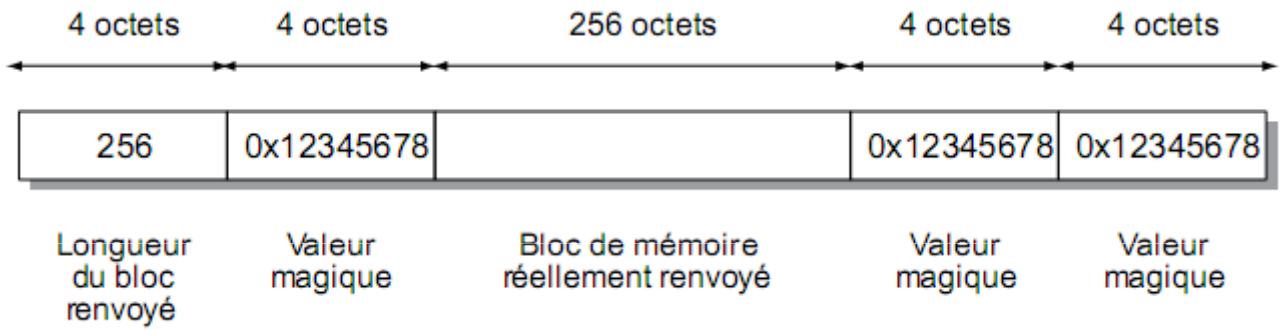
- Inclure `<mcheck.h>`
- Le fichier a générer est spécifié dans la variable d'environnement `MALLOC_TRACE`
- Le fichier généré est lisible avec le script `/usr/bin/mtrace`

- Exemple :

```
$ export MALLOC_TRACE="trace.out"
$ ./exemple_mtrace
$ mtrace exemple_mtrace trace.out

Memory not freed:
-----------------
Address     Size     Caller
0x08049750  0x200    at /home/arno/exemple_mtrace.c:13
```

Mémoire – Encadrement des zones allouées

- Des outils permettent de faire des vérifications d'intégrité poussées, et de détecter non seulement les fuites mémoire, mais aussi les débordements de buffers



- Electric-fence : http://perens.com/FreeSoftware/ElectricFence/
- Valgrind : http://valgrind.org/

- Purify : http://www-01.ibm.com/software/awdtools/purify (commercial)
- Insure++ : http://www.parasoft.com (commercial)

Mémoire – Valgrind – Memcheck

- L'outil memcheck de la suite Valgrind intercepte les appels à `malloc()`, `new()`, `free()` et `delete()`
- Il peut détecter :
  - Accès mémoire invalides (mémoire non allouée ou libérée, overruns)
  - Utilisation de variables non initialisées
  - Fuites mémoire
  - Double libération de mémoire
  - Ecrasement de blocs copié par `memcpy()`

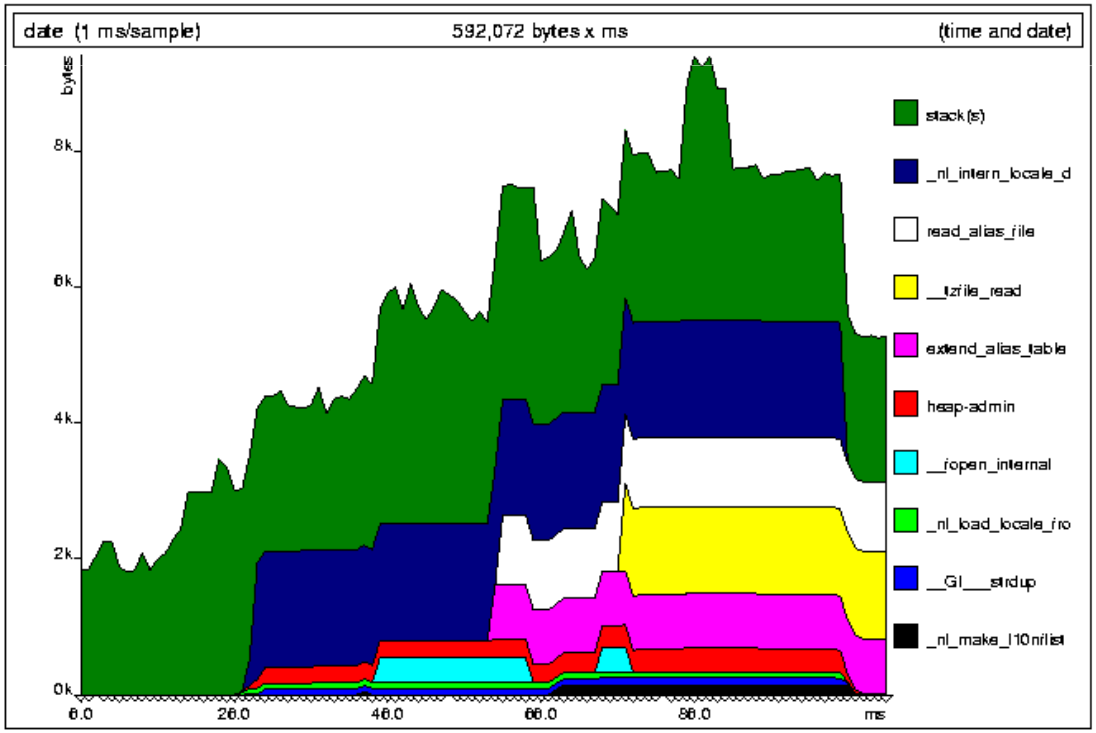
- Exemple :

```
$ gcc –g prog.c –o prog
$ valgrind --tool=memcheck --leak-check=yes ./prog
==11323== Invalid write of size 4
==11323==     at0x8048518:main(prog.c:9)
==11323==   Address 0x1BB261B8 is 0 bytes after a block
==11323==     of size 400 alloc'd
==11323==     at 0x1B903F40: malloc
==11323==    (in/usr/lib/valgrind/vgpreload_memcheck.so)
==11323==    by 0x80484F2: main(prog.c:6)
```

Mémoire – Valgrind – Massif

- L'outil massif de la suite Valgrind permet de profiler l'utilisation du tas et de la pile d'un programme
- Le résultat est produit sous la forme d'un graphe montrant les usages mémoires durant le temps d'exécution
- Exemple :

```
$ gcc –g prog.c –o prog
$ valgrind --tool=massif ./prog
```



Mémoire – Fichier core

- Sous Unix, un fichier core est créé lorsqu'un programme est tué par certains signaux (`SIGABRT`, `SIGSEGV`, `SIGILL`, …)
- Le fichier contient une image de la mémoire du processus

- Par défaut, le fichier `core` est créé dans le répertoire courant
  - Peut être défini dans `/proc/sys/kernel/core_pattern`

- La création d'un fichier core peut être désactivée :
  - Les attributs `RLIMIT_CORE` ou `RLIMIT_FSIZE` sont égaux à 0
  - Le processus ne peut pas créer de fichier

- La plupart des distributions Linux désactive la création de fichiers core
- Un utilisateur peut voir et changer la limite de taille du fichier :
  - `$ ulimit -a`
  - `$ ulimit -c 1024`

- La commande `gcore` de gdb peut créer un fichier core au cours de l'exécution d'un programme

Mémoire – Fichier core - gdb

- gdb peut lire un fichier core et permet de faire un backtrace remontant à l'instruction coupable

- Exemple :

```
$ ./buggy
Segmentation fault (core dumped)
$ gdb buggy core
[...]
Core was generated by `./buggy'.
Program terminated with signal 11, Segmentation fault.
#0  0x00007f0c877c7001 in strncpy () from /lib/libc.so.6
(gdb) bt
#0  0x00007ff68fade001 in strncpy () from /lib/libc.so.6
#1  0x0000000000400680 in main () at buggy.c:23
(gdb) up
#1  0x0000000000400680 in main () at buggy.c:23
23                  strncpy(temp->data, test_str, strlen(test_str));
(gdb)
```

Profiling – gprof

- gprof est un profiler ; outil mesurant combien de fois chaque fonction d'un programme est appelée et combien de temps y est passé
- gprof fait une mesure par échantillonnage : à intervalles réguliers, il relève dans quelle fonction le programme s'exécute

- Exemple :

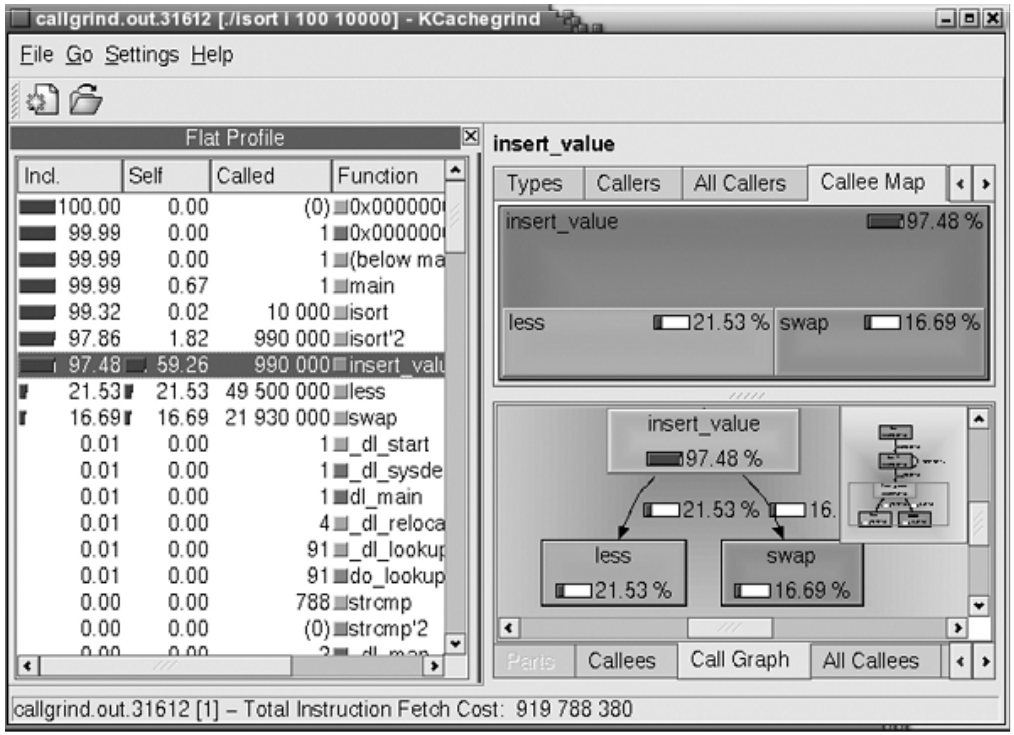
```
$ gcc –pg –g isort.c –o isort
$ ./isort i 100 1 00000
$ gprof isort gmon.out
Each sample counts as 0.00195312 seconds.
%  cumulative    self              self     total
time  seconds seconds     calls  ns/call   ns/call name
66.79   22.47   22.47 495000000    45.40     45.40 less
25.56   31.07    8.60   9900000   868.45   3356.02 insert_value
 6.41   33.22    2.16 219300000     9.83      9.83 swap
 1.03   33.57    0.35    100000  3476.56 335722.66 isort
 0.21   33.64    0.07                       main
```

Profiling – Valgrind – Callgrind

- L'outil callgrind de la suite Valgrind est un profiler
- Le résultat est plus précis qu'avec gprof (Valgrind utilise une machine virtuelle)
- L'outil kcachegrind permet de voir les résultats de graphiquement

- Exemple :

```
$ gcc isort.c –o isort
$ valgrind --tool=callgrind ./isort i 100 10000
$ callgrind_annotate callgrind.out.31612
919,788,380  PROGRAMTOTALS
545,100,000  insert_value
198,000,000  less
153,510,000  swap
16,770,000   isort'2
```



Profiling – Valgrind - Helgrind

- L'outil helgrind de la suite Valgrind permet de détecter des erreurs dans un programme multithreads
  - Verrouillages / déverrouillages illégaux de mutex
  - Libération de mémoire contenant un mutex vérrouillé
  - Terminaison de threads vérrouillant un mutex
  - Report des erreurs retournées par les fonctions pthread
  - Détection de conditions de races

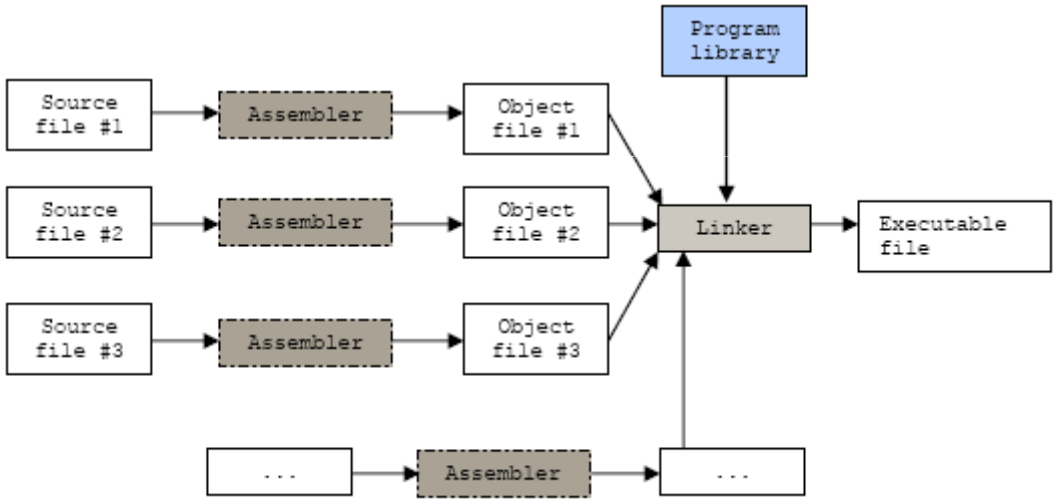
- Exemple :

```
$ gcc –g prog.c –lpthread –o prog
$ valgrind --tool=helgrind prog
...
== Possible data race during write of size 4 at 0xFEFF6D4C
==     at 0x8048475: prog(prog.c:11)
==     by 0x48CDD19: clone(in/lib/tls/libc-2.3.2.so)
==   Oldstate: shared-readonly by threads #2, #3
==   Newstate: shared-modified by threads #2, #3
==   Reason: this thread, #3, holds no consistent locks
==   Location 0xFEFF6D4C has never been protected by any lock
```

Librairies

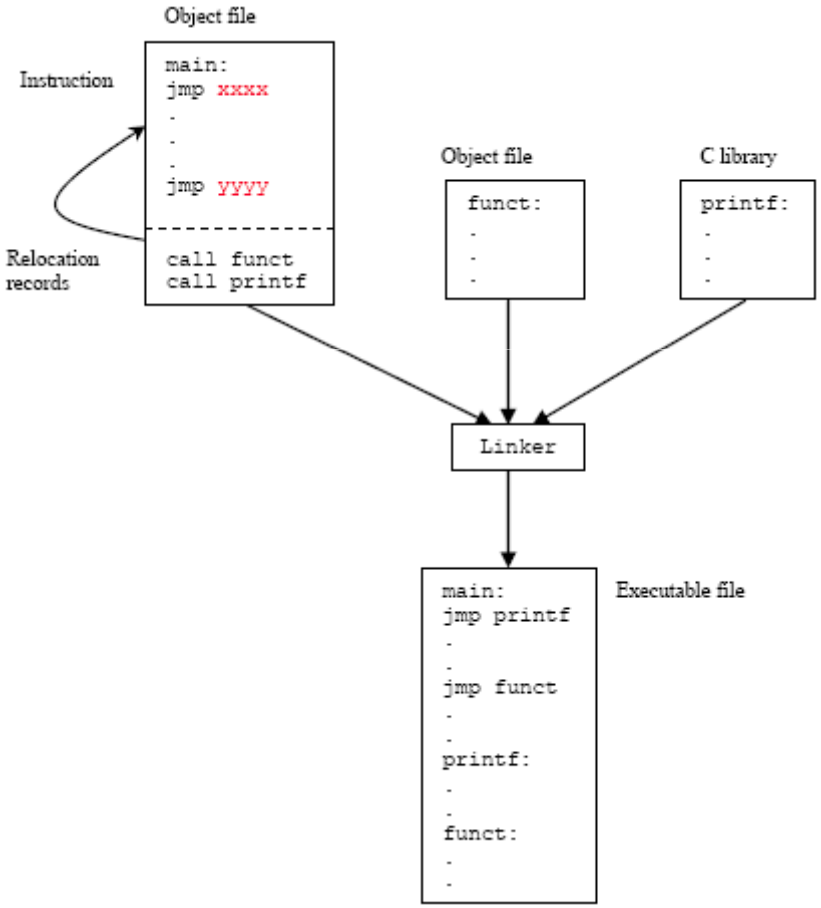
Edition de liens

- Un fichier exécutable est souvent créé depuis plusieurs fichiers sources
- Ces fichiers sources sont indépendamment compilés et assemblés en fichiers objets
- L'édition de lien consiste à combiner ces fichiers objets



Edition de liens (suite)

- Les fichiers objet réfèrent souvent entre eux du code et des données :



Edition de liens (suite)

- Ces références sont externes sont placées dans des zones de relocation (relocation records) par l'assembleur

- Chaque relocation consiste en :
  - Un index dans une table de symboles (de sorte à savoir quel symbole est référencé)
  - Un offset qui réfère à l'adresse utilisant ce symbole
  - Un tag qui indique le type de la relocation

- Après avoir combinés les fichiers objets, le linker doit résoudre ces relocations

Edition de liens statique

- L'édition de lien statique consiste à combiner ensemble du code source et du code d'une librairie particulière
- Le linker résout les relocations vers la librairie dans l'exécutable lui-même
  - Le code objet de la librairie est copié dans l'exécutable
  - Les symboles référencés dans le programme sont associés avec le code copié de la librairie
- Un changement dans la librairie nécessite une réédition de liens
- Sous Linux, les programmes qui sont liés statiquement le sont avec des archives de fichiers objets (librairies statiques) qui ont l'extension `.a`

Librairies statiques

- Pour créer une librairie statique, on compile d'abord les sources en fichiers objets :

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

- Les fichiers objets sont combinés en une librairie statique par l'utilitaire `ar` (archiver) :

```
$ ar rcs libstuff.a foo.o bar.o
```

- Le nom de la librairie doit commencer par `lib` et avoir l'extension `.a`

- La liste des fichiers objets contenus dans une librairie statique peut être affichée :

```
$ ar t libstuff.a
foo.o
bar.o
```

Librairies statiques (suite)

- Pour compiler un programme avec une librairie statique :

```
$ gcc fubar.c libstuff.a -o fubar
```

- Si la librairie est installée dans les répertoires standards de librairies (comme `/usr/lib`), utiliser l'option `-l` de `gcc` avec le nom de la librairie (sans `lib` et sans l'extension `.a`) :

```
$ gcc fubar.c -o fubar -lstuff
```

- Si une librairie dynamique de même nom existe dans les répertoires standards, celle-ci sera préférée par défaut
- Pour forcer l'édition de liens avec une libraire statique, il faut utiliser l'option `-static` :

```
$ gcc -static fubar.c -o fubar -lstuff
```

- L'option `-L` permet de spécifier un répertoire non standard dans lequel est la librairie :

```
$ gcc fubar.c -o fubar -L. -lstuff
```

Edition de liens dynamique

- Typiquement dans un système, un certain nombre de programmes utilisent les mêmes fonctions issues de librairies
- Si tout ces programmes sont compilés statiquement, chacun aura une copie du code de ces fonctions dans son exécutable
  - Gaspillage d'espace disque et de mémoire

- Il est donc préférable que chaque programme référence une seule instance mémoire de chaque fonction
- Certaines relocations ne seront pas résolues durant l'édition de liens mais durant l'exécution (édition de liens différée ou dynamique) :
  - Les symboles des objets référencés sont seulement vérifiés pour être certain qu'ils existent quelque part
  - Le linker enregistre dans l'exécutable le nom des librairies dans lesquelles il a trouvé les symboles manquants
  - Les librairies sont dites "position indépendante" ; le code peut être chargé n'importe où en mémoire

- Sous Linux, les programmes qui sont liés dynamiquement le sont avec des archives de fichiers objets (librairies dynamiques) qui ont l'extension `.so`

Librairies partagées

- Les fichiers objets destinés à être combinés dans une librairie partagée doivent compilés en “position indépendante” :

```
$ gcc -fpic -c foo.c bar.c
```

- Pour créer une librairie partagée :

```
$ gcc -shared -o libstuff.so foo.o bar.o
```

- Le nom de la librairie doit commencer par `lib` et avoir l’extension `.so`

Librairies partagées – Versions

- L'un des intérêts des librairies partagées est qu'il est possible de les modifier sans pour autant devoir recompiler les programmes les utilisant (tant que l'API ne change pas)
- Pour cela, chaque librairie se voit assignée un nom spécial, le `soname`, qui inclus les nom et version majeure de la librairie
- Le `soname` n'est changé que lorsque l'API de la librairie n'est plus compatible avec la version précédente (changement de version)
- Le nom de fichier de la librairie inclus généralement les versions majeure et mineure

- La librairie doit être compilée avec le `soname` passé au linker :

```
$ gcc -shared -Wl,-soname,libstuff.so.1 -o libstuff.so.1.0.1
                                                foo.o bar.o
```

- Un lien symbolique est souvent utilisé pour pointer sur la bonne version :

```
$ ls –l /lib/libstuff*
 -rwxr-xr-x  1.6M /lib/libstuff.so.1.0.1
 Lrwxrwxrwx    14 /lib/libstuff.so.1 -> libstuff.so.1.0.1
```

Librairies partagées – Linker

- L'option `-l` de gcc indique au linker ld qu'un programme doit être lié avec une librairie
- Par défaut, ld cherche la librairie dans les répertoires `/lib` et `/usr/lib`
- L'option `-L` permet d'indiquer à ld de chercher dans un répertoire particulier

```
$ gcc fubar.c –o fubar –L. -L/usr/local/lib/ -lstuff
```

- L'emplacement de la librairie peut être explicitement indiqué

```
$ gcc fubar.c –o fubar /usr/local/lib/libstuff.so
```

- Le linker inclus le `soname` de la librairie dans le fichier exécutable
- La commande `ldd` permet de vérifier les librairies à charger pour un programme donné

```
$ ldd fubar
libstuff.so.1 (0x0000002a95557000)
libc.so.6 => /lib64/tls/libc.so.6 (0x00000352e100000)
/lib64/ld-linux-x86-64.so.2 (0x000000352dd00000)
```

Librairies partagées – Installation

- Le loader `/lib/ld.so` charge les librairies partagées nécessaires à un programme, le prépare et lance son exécution

- Les librairies chargées par `ld.so` sont cherchées dans l'ordre suivant :
  - Dans les répertoires indiqués dans la variable d'environnement `LD_LIBRARY_PATH` (séparés par ':')
  - Dans le fichier cache `/etc/ld.so.cache`
  - Dans les répertoires `/lib` et `/usr/lib`

- La commande `ldconfig` crée les liens symboliques et met à jour le cache avec les versions les plus récentes des librairies (spécifiées en argument, dans le fichier `/etc/ld.so.conf`, et dans les répertoires `/lib` et `/usr/lib`)

- Quand on ajoute une librairie partagée qui n'est pas dans `/lib` ou `/usr/lib`, il faut donc ajouter le répertoire dans `/etc/ld.so.conf` et exécuter `ldconfig`

Chargement dynamique – dlopen()

- Les librairies partagées peuvent être chargées et déchargées dynamiquement pendant l'exécution d'un programme (et non au lancement)
  - Permet de créer une architecture de plugins
- Les fonctions supportant ce mécanisme sont prototypées dans `<dlfcn.h>`
- L'exécutable doit être compilé avec l'option `-ldl` de gcc (libdl)

- La fonction `dlopen()` permet de charger en mémoire une libraire partagée

```
void *dlopen(const char *filename, int flag);
```

- `filename` : nom de la librairie
- `flag` :
  - `RTLD_LAZY` : résolution des symboles seulement à l'exécution
  - `RTLD_NOW` : résolution des symboles au chargement de la librairie
  - Flags optionnels :
    - `RTLD_GLOBAL` **: les symboles seront disponibles à d'autres librairies**
    - `RTLD_LOCAL` **: les symboles ne seront pas disponibles à d'autres librairies**
- Retourne un handler ou `NULL` si erreur

Chargement dynamique – dlsym()

- La fonction `dlsym()` retourne l'adresse mémoire d'un symbole

```
void *dlsym(void *handle, const char *symbol);
```

- `handle` : pointeur retourné par `dlopen()`
- `symbol` : nom de la fonction recherchée
- Retourne un pointeur sur le symbole (en théorie peut être `NULL`)

- Les compilateurs C++ décorent les noms de symboles, il faudra donc les déclarer extern C pour qu'ils soient utilisables par `dlsym()` :

```
extern "C" {
    extern int foo;
    extern void bar();
}
```

Chargement dynamique – dlclose() et dlerror()

- La fonction `dlclose()` décrémente le nombre de références sur une librairie ouverte par `dlopen()`

```
int dlclose(void *handle);
```

- `handle` : pointeur retourné par `dlopen()`
- Retourne 0 ou !0 si erreur

- La fonction `dlerror()` affiche sur un message d'erreur décrivant l'erreur la plus récente suite à un appel à l'une des fonctions `dlopen()` ou `dlsym()`

```
char *dlerror(void);
```

- Retourne une chaine de caractère ou `NULL` s'il n'y a pas eu d'erreur

Chargement dynamique – Exemple

```
int main(void)
{
    typedef double (*cos_t)(double);

    void* handle = dlopen("libm.so", RTLD_LAZY);
    if (!handle) {
          fprintf(stderr, "Cannot open library: %s\n", dlerror());
          return 1;
    }

    dlerror();           // reset errors
    cos_t cos = (cos_t) dlsym(handle, "cos");
    const char *dlsym_error = dlerror();
    if (dlsym_error) {
          fprintf(stderr, "Cannot load symbol 'cos': %s\n", dlsym_error);
          dlclose(handle);
          return 1;
    }

    printf("cos 2 is %f\n", cos(2));

    dlclose(handle);
    return 0;
}
```

Format ELF

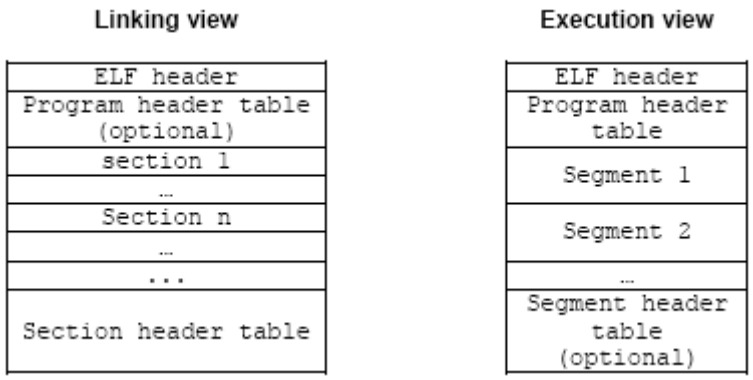
- Le format ELF est un format de fichier (exécutables, objets, librairies, cores) commun dans le monde Unix (Solaris, *BSD et Linux)
- Il remplace l'ancien format a.out
- Supporte "Position Independent Code", même pour les exécutables (PIE)
- Supporte les architectures 32 et 64 bits
- Designé pour le mapping mémoire direct

- Typiquement, les fichiers objets, exécutables et librairies utilisent un format de fichier basé sur des sections :
  - `.text` : code exécutable
  - `.data` : variables initialisées
  - `.bss` : variables non initialisées

- L'assemblage en code machine enlève tout les labels du code
- Les fichiers objets et librairies doivent donc garder ces labels dans une table de symboles
  - La table des symboles est une liste de noms et de leurs offsets correspondant vers les segment de code et de données

Format ELF (suite)

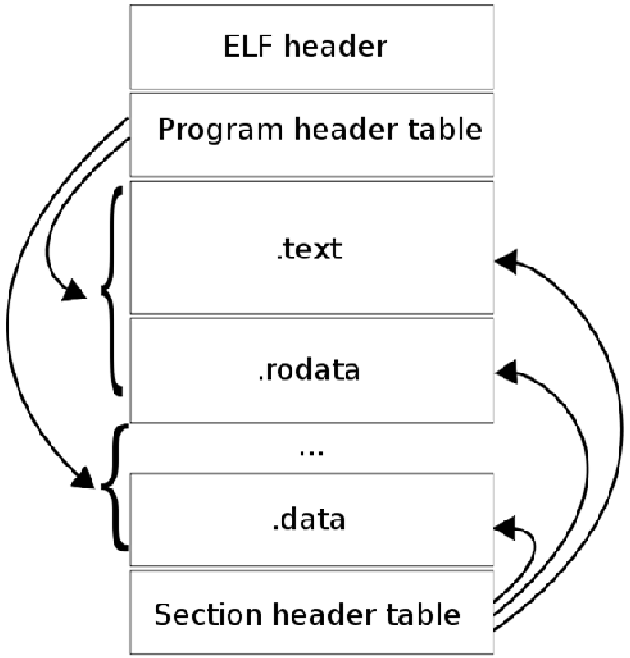
- Le format ELF définit les sections d'un exécutable, fichier objet, ou librairie
- Les principales sections sont :
  - `Global Offset Table` : table des symboles
  - `Procedure Linkage Table` : liens indirects vers la GOT
  - `.init / .fini` : initialisation et shutdown internes
  - `.ctors / .dtors` : constructeurs et destructeurs internes
  - `.rodata` : données en lecture seule
  - `.data` : données initialisées
  - `.bss` : données non initialisées
  - `.text` : code exécutable



Format ELF (suite)

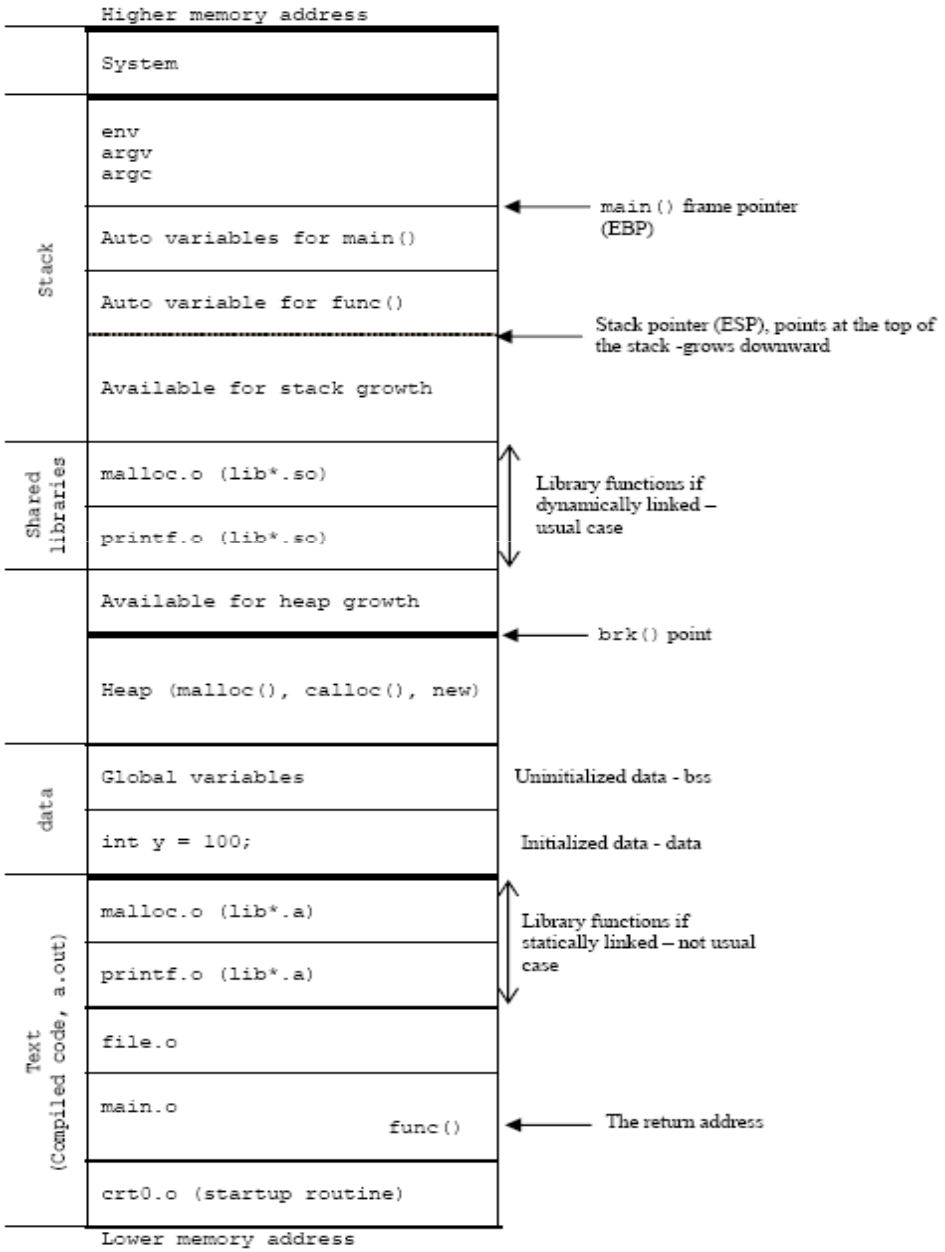
- Dans un exécutable, contrairement à un fichier objet ou à une librairie, les sections de même type sont regroupées en segments

- Les sections `.text` et `.rodata` sont combinées en un segment `.text`
- Les sections `.data` et `.bss` sont combinées en un segment `.data`
- D'autres sections sont regroupées en segments (symboles, debug, …)



- http://refspecs.freestandards.org/elf/elf.pdf
- http://people.redhat.com/drepper/dsohowto.pdf

Format ELF (suite)



- Le loader crée une image processus en chargeant les segments `.text` et `.data` en mémoire (load segments), selon les informations de la table `Program Header`
- Les références non résolues sont recherchées par le loader
- Le loader change ces références en références mémoire, relatives au début de l'espace d'adressage
- `/proc/<pid>/maps`
- `/proc/<pid>/smaps`
- `$ pmap –x <pid>`

FORMATION

EUROGICIEL GROUPE

Outils ELF

- Il est possible d'obtenir des statistiques sur le coût CPU des relocations :

```
$ env LD_DEBUG=statistics ./prog
18550:  runtime linker statistics:
18550:     total startup time in dynamic loader: 2656525 clock cycles
18550:     time needed for relocation: 41798 clock cycles (1.5%)
18550:     number of relocations: 0
18550:     number of relocations from cache: 26
18550:     number of relative relocations: 0
18550:     time needed to load objects: 2430410 clock cycles (91.4%)
18550:  runtime linker statistics:
18550:     final number of relocations: 0
18550:     final number of relocations from cache: 26
```

Outils ELF – readelf

- `readelf` est un utilitaire pouvant décortiquer un fichier ELF
- Afficher le header ELF :
```
$ readelf -h prog
```
- Afficher les informations contenues dans les sections header :
```
$ readelf -S prog
```
- Afficher les entrées du program header :
```
$ readelf -l prog
```
- Afficher les entrées de la table des symboles :
```
$ readelf -s prog
```
- Afficher les informations contenues dans la section dynamic :
```
$ readelf -d prog
```
- Afficher les relocations :
```
$ readelf -r prog
```

Outils ELF – objdump

- `objdump` affiche le contenu de fichiers ELF
- Afficher le code désassemblé de la section `.text` :

```
$ objdump -d -j .text prog
```

- Pour complètement désassembler un programme :

```
$ objdump -Dslx prog
```

- Pour désassembler une librairie :

```
$ objdump –dR libstuff.so
```

Outils ELF – nm

- `nm` affiche les symboles contenus dans un fichier objet ou une librairie partagée
- Afficher tout les symboles :

```
$ nm prog
```

- Un type est donné pour chaque symbole :
  - A : la valeur est absolue
  - B : section .bss
  - D : section .data
  - T : section .text
  - U : symbole non défini (dépend d'une autre librairie)