PG204 - Programmation système

2eme année Informatique

M. Faverge - mfaverge@enseirb-matmeca.fr

https://cours-mf.gitlabpages.inria.fr/pg204/

2025 - 2026

ENSEIRB-MatMeca

Objectifs du cours

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

- Principes généraux
- Exemples pratiques de programmation
- Etudier les couches les plus basses d'un logiciel sur un système informatique

Objectifs du cours

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

- Principes généraux
- Exemples pratiques de programmation
- Etudier les couches les plus basses d'un logiciel sur un système informatique

Ce que le cours n'est pas:

- cours de programmation C
- apprenez le C ! Lisez le C, de B. Kernighan et D. Richie
- cours de Shell
- apprenez votre shell ! Utilisez `man`

Plan

Introduction

Appels systèmes

Fichiers

Processus et gestion mémoire

Signaux

Mémoire partagée et synchronisation

Threads

Concurrence

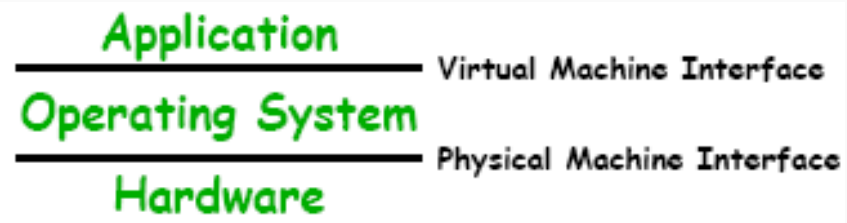
Réseau - sockets

Introduction

Qu’est ce qu’un système d’exploitation?

Un système d’exploitation:

- gère et cache la complexité du matériel au programmeur
- fournit une interface virtuelle de la machine
- fournit une protection entre les utilisateurs (souvent)



Qu’est ce qu’un système d’exploitation?

Services proposés par l’OS

- Gestionnaire de ressources matérielles
  - cache la complexité du matériel
  - gère les conflits de requêtes d’accès aux ressources, permet un usage équitable
  - gère les erreurs
  - empêche les usages impropres de la machine
- Facilite l’utilisation de la machine
  - ensemble de bibliothèques standard (fenêtrage par ex.)
  - rend la programmation plus facile, plus simple

Qu’est ce qu’un système d’exploitation?

Quelles ressources, quels services?

- Temps CPU:
  - Ordonnancement des tâches sur le processeur (scheduling)
- Mémoire:
  - Allocation
  - Protection mémoire
- Entrées/sorties:
  - Système de fichiers et droits,
  - Gestion des diverses cartes et protocoles réseau,
  - Fenêtrage
- Consommation d’énergie: réglage fréquence, . . .

Qu’est ce qu’un système d’exploitation?

Structurer les données

- Multiplexer le réseau
  - Un seul flux de bits qui contient des informations de différents programmes vers différents serveurs
  - Comme un tuyau qui contiendrait plusieurs produits qu’on peut séparer à la sortie
- Organiser le disque dur
  - Un immense tableau de blocs où on veut stocker une arborescence de dossiers et fichiers
  - Comme un grande feuille de papier sur laquelle on écrirait à plusieurs en même temps

Où sont les systèmes d'exploitation?

Dans les super-calculateurs:

Frontier SuperComputer, 1,1 Eflops (10^{18} flops/s), ordinateur le plus puissant du monde (Depuis juin 2022).

`http://www.top500.org` (MaJ tous les 6 mois)

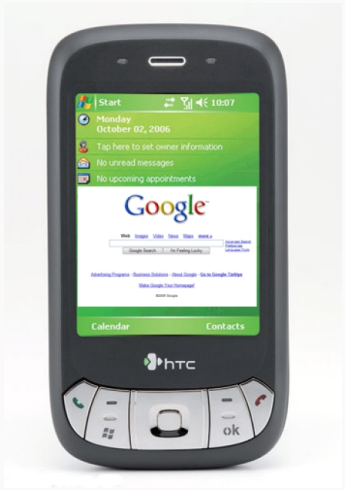
Où sont les systèmes d'exploitation?

Dans les centres de données (datacenter): banques, assurances, . . .

Où sont les systèmes d'exploitation?

Dans les systèmes embarqués: internet box, équipements multimédia

Dans les équipements mobiles: téléphones android, symbian, dans les kits mains libres, . . .



Où sont les systèmes d'exploitation?

Avec le cloud computing $\rightarrow$ plus de localisation

- Le programme Google: navigateur (Chrome OS)
- L'ordinateur Google: des milliers d'ordinateurs google dans les centres de données et de calcul répartis dans le monde
- Services: système de fichier (web), mail/chat (communication réseau), ...

Où sont les systèmes d'exploitation?

Certains systèmes critiques n'utilisent pas d'OS

- Systèmes qui n'exécutent qu'un seul programme
  - et ce programme n'a pas besoin de s'exécuter ailleurs
- Systèmes temps réels :
  - OS spécifique, juste pour un ensemble de tâches et d'événements (centrale nucléaire)
  - pas d'OS (Airbus A320-380)
- Langages de programmation, compilateurs spécifiques temps réel
  - essayer de garantir certaines propriétés (temps de réaction, pas de crash, . . .)
  - Lustres, Esterel, Oasis

Cacher la complexité du matériel

Les machines sont très différentes

- Type et nombre de processeurs/cœurs
- Quantité de mémoire
- Type de disque dur ou carte réseau

On veut qu'un même programme fonctionne partout sans devoir être modifié

- Peu importe la quantité de mémoire disponible, je peux lancer mon programme
- J'ai une carte réseau, peu importe ce qu'elle sait faire, je l'utilise de la même façon
- Peu importe si mon disque dur est un SSD ou un NAS connecté par le réseau, j'utilise mes fichiers de la même façon

Un peu d'histoire

Années 50-60

- Un seul programme à la fois, sur un seul processeur

Années 70

- Un autre programme peut s'exécuter quand le premier est bloqué
  - attente de la fin d'une lecture sur le disque dur

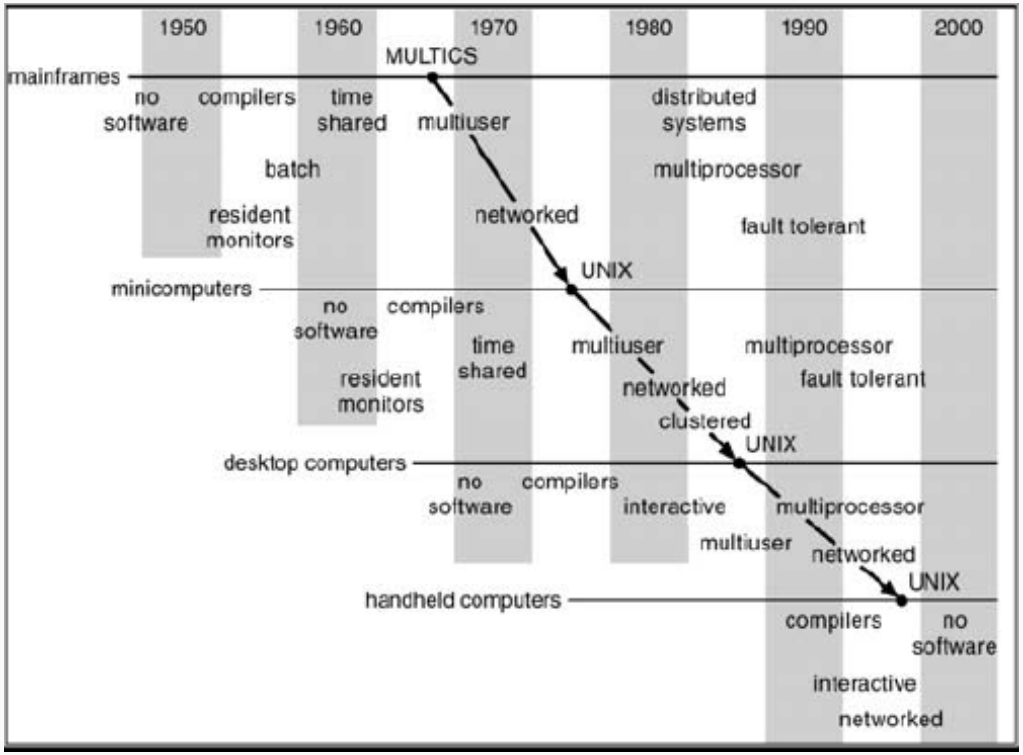
Années 80

- Interactivité
  - Les programmes réagissent aux actions de l'utilisateur
  - L'utilisateur peut alterner entre plusieurs programmes

Années 90

- Plusieurs processeurs puis plusieurs cœurs par processeur
  - Plusieurs programmes s'exécutent en même temps

Un peu d'histoire



Complexité $\rightarrow$ Besoin d'OS

Un peu d'histoire

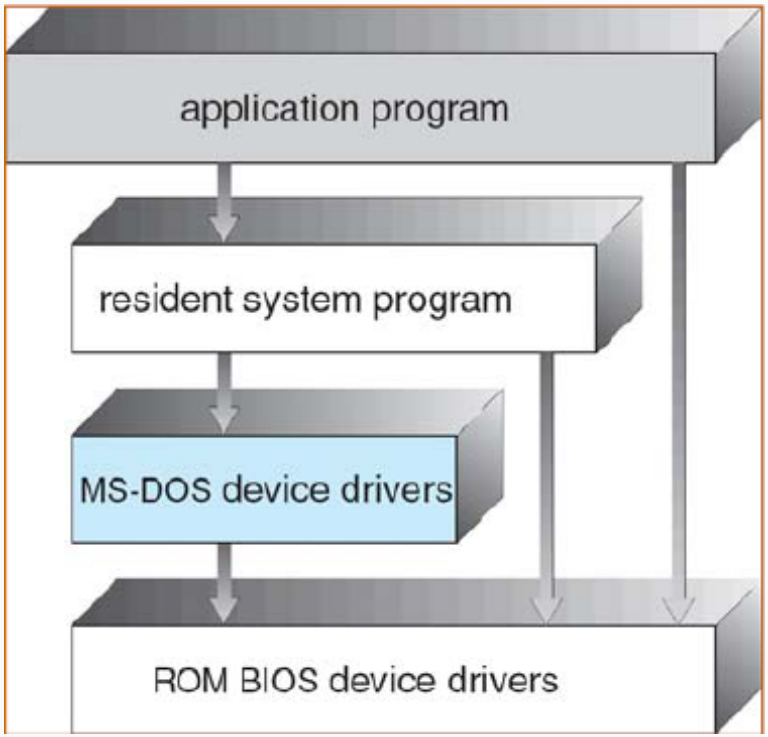
Au début, un OS=quelques bibliothèques

Hypothèses simplificatrices:

- pas de mauvais programmeurs/utilisateurs
- un programme à la fois

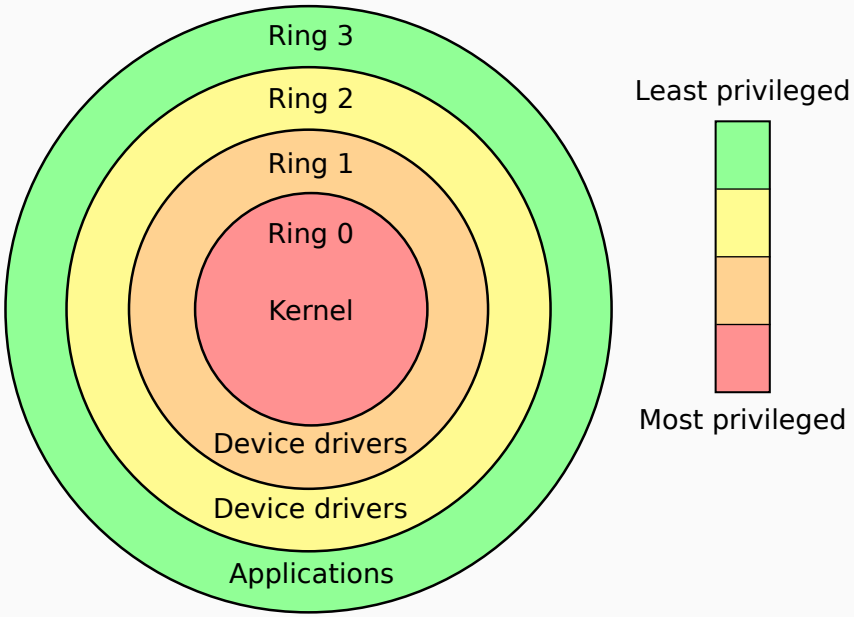
$\Rightarrow$ Mauvaise utilisation:

- du temps machine
- du temps de l'utilisateur

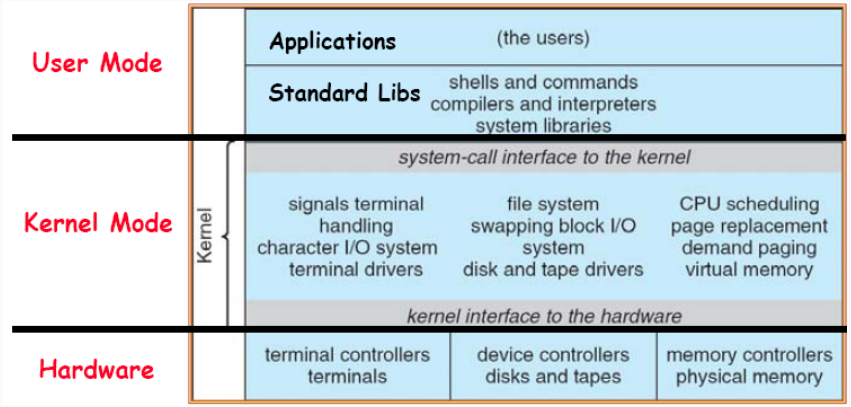


Structure d'un OS moderne

Une structure en oignon



Structure d'un OS moderne



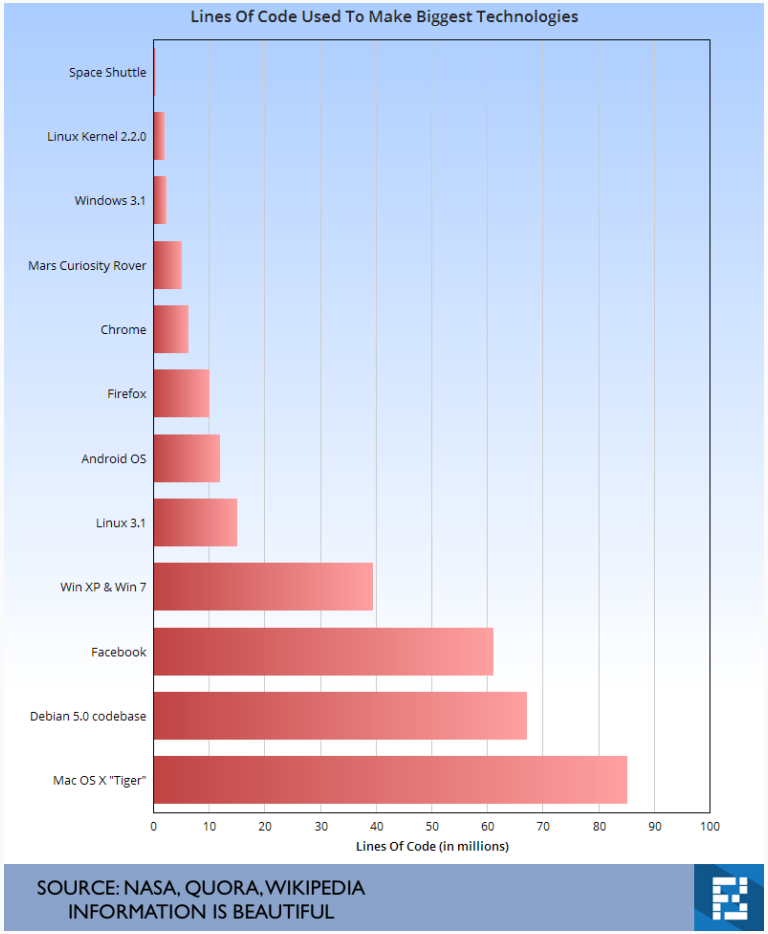
https://blog.xmco.fr/info-minix-un-systeme-dexploitation-cache-dans-vos-processeurs-intel/

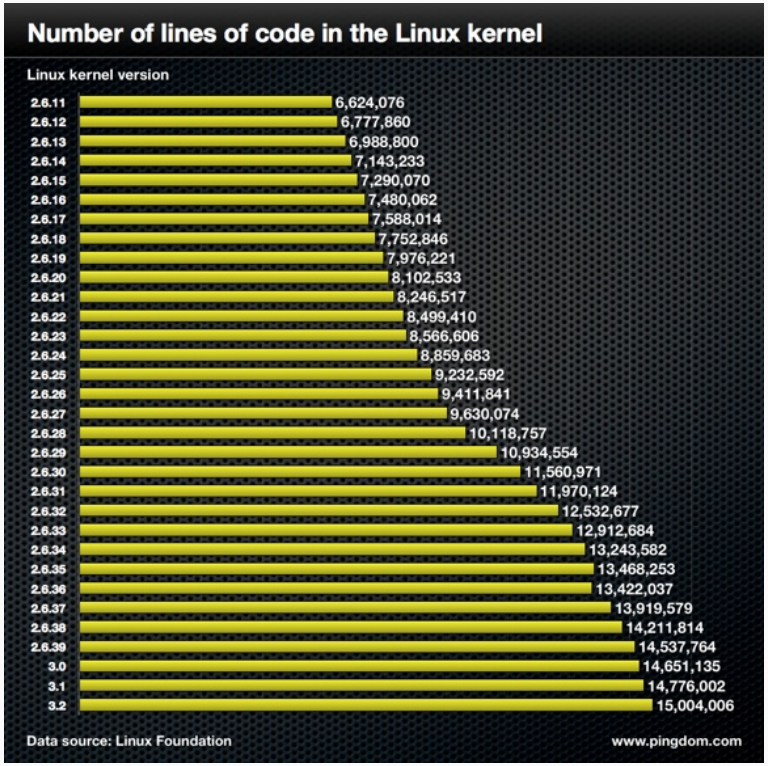
Structure d'un OS moderne

Différents composants logiciels

- Un noyau qui s'exécute en mode **privilégié** et peut tout faire
  - Modifier la mémoire, parler aux périphériques, etc.
  - Ce n'est pas le super-utilisateur/administrateur/root
    - Il a juste un peu plus de droits que les autres
- Des programmes qui s'exécutent en mode **non-privilégié**
  - Ils peuvent uniquement utiliser/modifier leur mémoire
    - Sauf s'ils font un appel système (voir plus loin)
  - Par exemple, des outils en ligne de commande (cp, mv, ...)
  - Plein d'autres programmes!
    - Par exemple un compilateur est juste un programme qui lit un fichier source et écrit un fichier binaire

Un logiciel très complexe



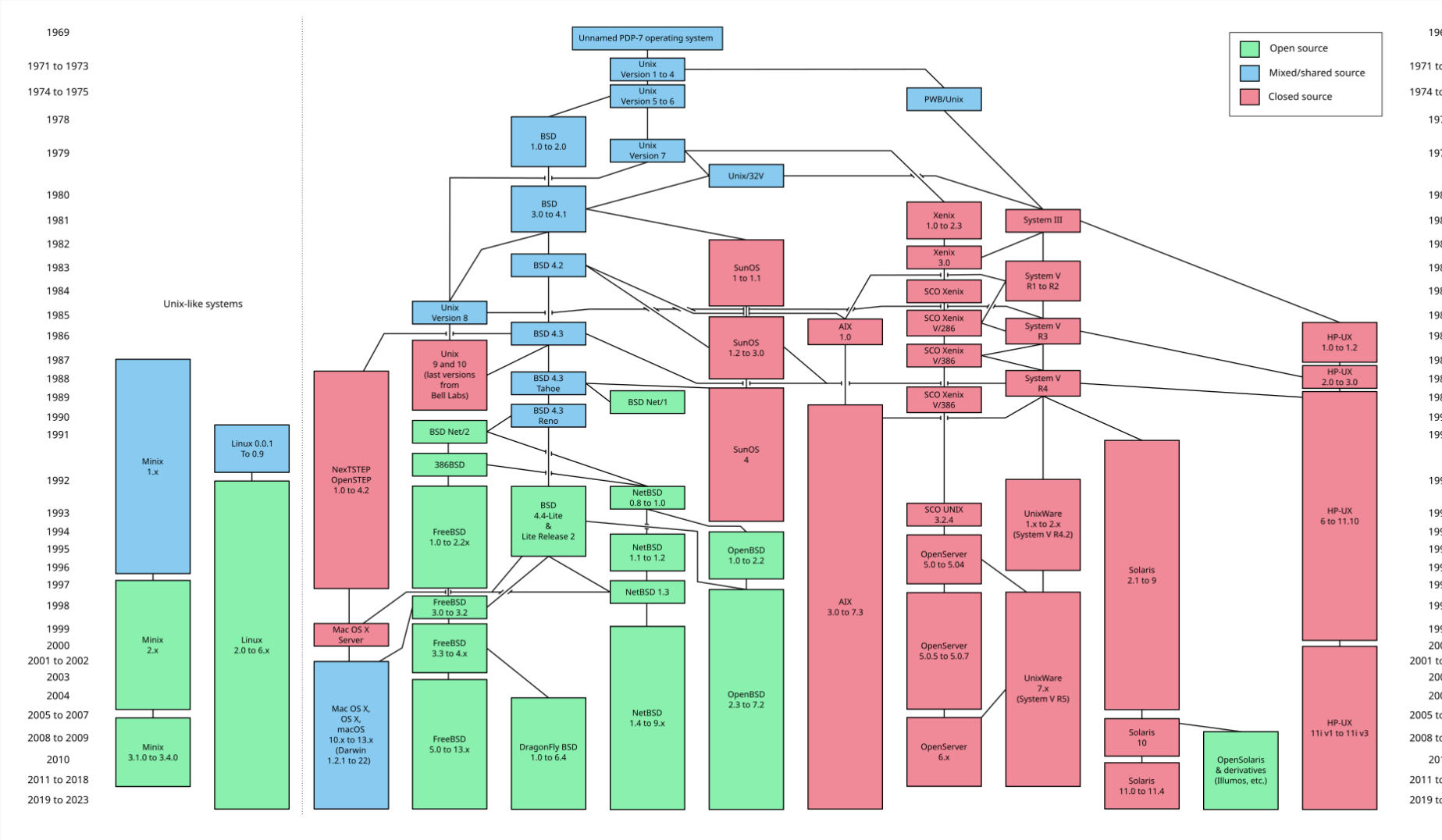


- supporter de nombreuses plates-formes différentes
- efficacement, pour satisfaire les utilisateurs
- Linux en 2020: 28M de lignes de code, 21K auteurs.

Exemples

- Linux, sur de nombreuses architectures
- De nombreux autres Unix (Solaris, BSD, etc)
- MacOS X qui dérive de BSD
- Android qui dérive de Linux
- Windows
- GNU/Hurd, basé sur un micro-noyau, avec une organisation assez différente
- De nombreux autres OS spécifiques (Plan9, Symbian, VxWorks, etc)

Généalogie


Open source
Mixed/shared source
Closed source
Unix-like systems
1969
1971 to 1973
1974 to 1975
1978
1979
1980
1981
1982
1983
1984
1985
1986
1987
1988
1989
1990
1991
1992
1993
1994
1995
1996
1997
1998
1999
2000
2001 to 2002
2003
2004
2005 to 2007
2008 to 2009
2010
2011 to 2018
2019 to 2023
Unnamed PDP-7 operating system
Unix Version 1 to 4
Unix Version 5 to 6
Unix Version 7
PWB/Unix
BSD 1.0 to 2.0
Unix/32V
BSD 3.0 to 4.1
Xenix 1.0 to 2.3
Xenix 3.0
System III
BSD 4.2
SunOS 1 to 1.1
SCO Xenix
System V R1 to R2
Unix Version 8
SCO Xenix V/286
BSD 4.3
SunOS 1.2 to 3.0
AIX 1.0
System V R3
SCO Xenix V/386
HP-UX 1.0 to 1.2
HP-UX 2.0 to 3.0
BSD 4.3 Tahoe
BSD Net/1
System V R4
Unix 9 and 10 (last versions from Bell Labs)
BSD 4.3 Reno
SCO Xenix V/386
BSD Net/2
SunOS 4
386BSD
NetBSD 0.8 to 1.0
SCO UNIX 3.2.4
UnixWare 1.x to 2.x (System V R4.2)
Solaris 2.1 to 9
HP-UX 6 to 11.10
Minix 1.x
Linux 0.0.1 To 0.9
NexTSTEP OpenSTEP 1.0 to 4.2
FreeBSD 1.0 to 2.2x
BSD 4.4-Lite & Lite Release 2
NetBSD 1.1 to 1.2
OpenBSD 1.0 to 2.2
OpenServer 5.0 to 5.04
NetBSD 1.3
FreeBSD 3.0 to 3.2
AIX 3.0 to 7.3
Minix 2.x
Linux 2.0 to 6.x
Mac OS X Server
FreeBSD 3.3 to 4.x
OpenBSD 2.3 to 7.2
OpenServer 5.0.5 to 5.0.7
UnixWare 7.x (System V R5)
Solaris 10
Solaris 11.0 to 11.4
OpenSolaris & derivatives (Illumos, etc.)
HP-UX 11i v1 to 11i v3
Minix 3.1.0 to 3.4.0
Mac OS X, OS X, macOS 10.x to 13.x (Darwin 1.2.1 to 22)
FreeBSD 5.0 to 13.x
DragonFly BSD 1.0 to 6.4
NetBSD 1.4 to 9.x
OpenServer 6.x

Références

- Livre
  - Operating System Concepts, Silbershatz, Galvin, Gagne (8^{eme} édition)
- Cours en ligne
  - J. Kubiatowicz, CS 162, Berkeley University
  - D. Maziere, CS 140, Stanford University
  - A. Cohen, INF570, école Polytechnique
- Autres ressources en ligne
  - `http://www.top500.org`
  - `http://www.wikipedia.org`
  - `http://www.linuxmanpages.com` ou `http://linux.die.net`
  - `http://www.osdata.com`

Appels systèmes

Abstraction:

- Simplifier et standardiser
- Portabilité du noyau
- Fonctions POSIX
- Faciliter le développement de drivers
- Stabiliser l'environnement d'exécution

Abstractions étudiées dans le cours:
Fichiers, Processus, Mémoire virtuelle, Threads, Communication

Protection

- Identification, droits d'utilisation/d'accès
- Isolation entre processus, entre utilisateurs
- Encryptage de données
- Définition de politiques de sécurité (réseau, fichiers, ...)

Virtualisation

- Favorise portabilité, programmation
  - Système de fichiers VFS sur Linux
  - Drivers SCSI pour tout un ensemble de périphériques (même si pas SCSI)
  - Vue homogène des caractéristiques du matériel
- Favorise stabilité, évolution:
  - Pas de crash de machine,
  - Isolation des processus
  - Simulation d'une machine/OS sur une autre (qemu)
  - Ipv4 sur Ipv6
- POSIX: normalise fonctions systèmes

Notion de processus

- Un processus est associé à un contexte qui lui est propre
  - Il a une vision de la mémoire qui lui est propre
  - Il ne peut pas voir la mémoire des autres processus
- Il est isolé sur la machine et a l'impression d'être seul
  - Les autres processus sont cachés, de même que le système.

Modes d'exécution (processus/processeur)

Un processus peut s'exécuter au travers de deux modes d'exécution du processeur.

Mode utilisateur (non-privilégié)

- Par défaut le processeur exécute le code en mode non privilégié
- Les programmes peuvent juste faire des additions, lectures, écritures, . . . dans leur mémoire
- Quand ils exécutent ces instructions basiques, le processeur ne touche que les données privées du processus (cf mémoire virtuelle plus loin)

Modes d'exécution (processus/processeur)

Un processus peut s'exécuter au travers de deux modes d'exécution du processeur.

Mode noyau (privilégié)

- Si un processus veut faire autre chose que les opérations précédentes, il doit demander l'autorisation à l'OS par une instruction spéciale : un appel système (*syscall*)
- Le matériel passe en mode privilégié et s'assure via l'OS que l'opération demandée par le processus est possible :
  - ouvrir un fichier, le modifier, le renommer, allouer de la mémoire, créer une tâche, terminer une tâche, . . .
  - tout ce qui est potentiellement sensible et qui pourrait nuire aux autres tâches et utilisateurs

Appels système

Mode kernel (noyau)

- Certaines instructions machines autorisées (gestion de la mémoire et interruptions) sont interdites en mode utilisateur

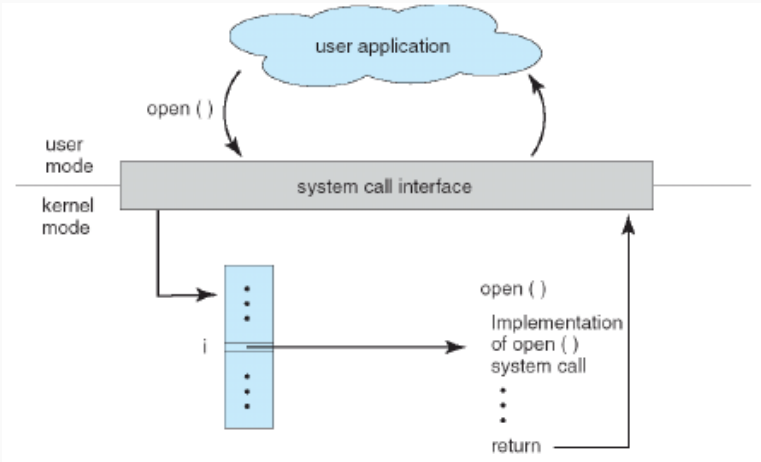
Noyau:

- Le noyau est un gestionnaire de processus, n'est pas un processus.
- Gère ressources et accès matériel.
- Fait pour être toujours en mémoire
- Chargé au boot de la machine
- Processus utilisateur passe en mode noyau pour utiliser un service proposé par le noyau

Appels système

Appels système:

- A l'interface entre mode utilisateur et mode noyau
- Changement de mode: instruction assembleur d'interruption (ta, int, . . .)
- Le noyau a un gestionnaire d'interruption
- Fonction noyau permet de faire des choses que l'utilisateur ne peut pas faire directement
- Les appels systèmes sont coûteux!!!
  Plusieurs centaines de cycles



Appels système

Appel système:

- retourne d'éventuels code d'erreur dans variable `errno`
- retourne valeur `-1` lorsqu'il y a une erreur
- affichage texte d'erreur avec:
  `void perror(const char *s);`
  ou `%m` dans `fprintf()`
- trace des appels système d'un programme avec
  `strace executable`

strace call

Example

```
> strace echo Hello World!
execve("/bin/echo", ["echo", "Hello", "World!"],
                    [/* 30 vars */]) = 0
uname({sys="Linux", node="debian", ...}) = 0
brk(0)                                  = 0x804d000
access("/etc/ld.so.nohwcap", F_OK)      = -1 ENOENT
                            (No such file or directory)
mmap2(NULL, 8192, PROT_READ|PROT_WRITE,
                    MAP_PRIVATE|MAP_ANONYMOUS, -1, 0)
      = 0xb7f35000access("/etc/ld.so.preload", R_OK)
      = -1 ENOENT (No such file or directory)
open("/etc/ld.so.cache", O_RDONLY)      = 3
...
write(1,"Hello World!",13)    = 13
...
```

Fichiers

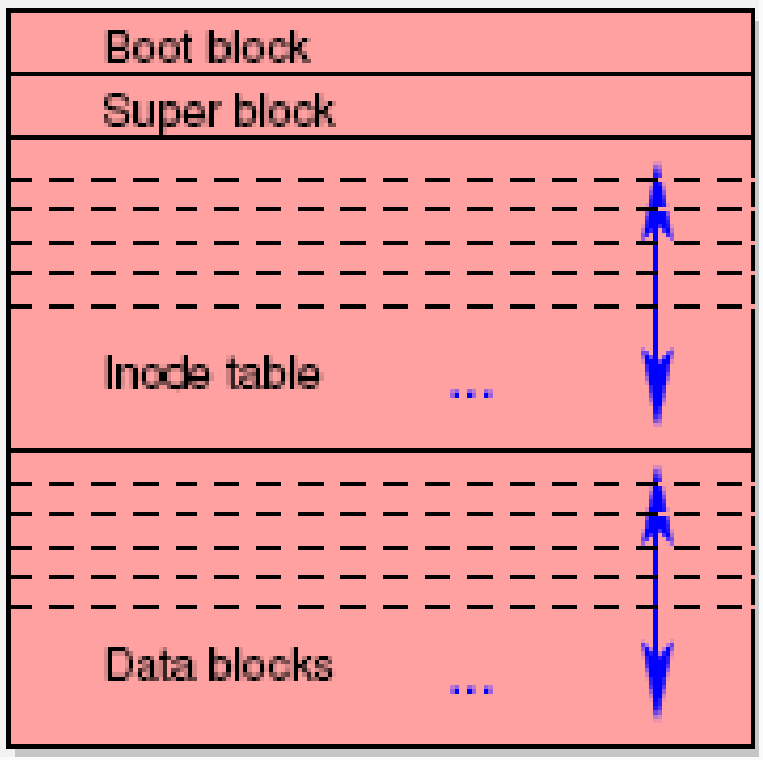
Comment accéder aux fichiers?

Il existe plusieurs niveaux de gestion des fichiers.

- stockage mémoire
  - Structure du système de fichier
  - Structure des fichiers, inodes.
- stockage du système
  - table des descripteurs (processus)
  - table des fichiers ouverts
  - table des inodes

Stockage: structure du système de fichiers

- Boot bloc:
  - block de données qui permet de booter la machine. Flag *bootable*
- Superbloc:
  - point de montage du système de fichiers
  - nombre de noeuds alloués/libres
  - liste des noeuds alloués/libres
- Table des inodes
- Blocs de données
  - contient les données des fichiers.
  - Pour un répertoire, liste des noms des fichiers



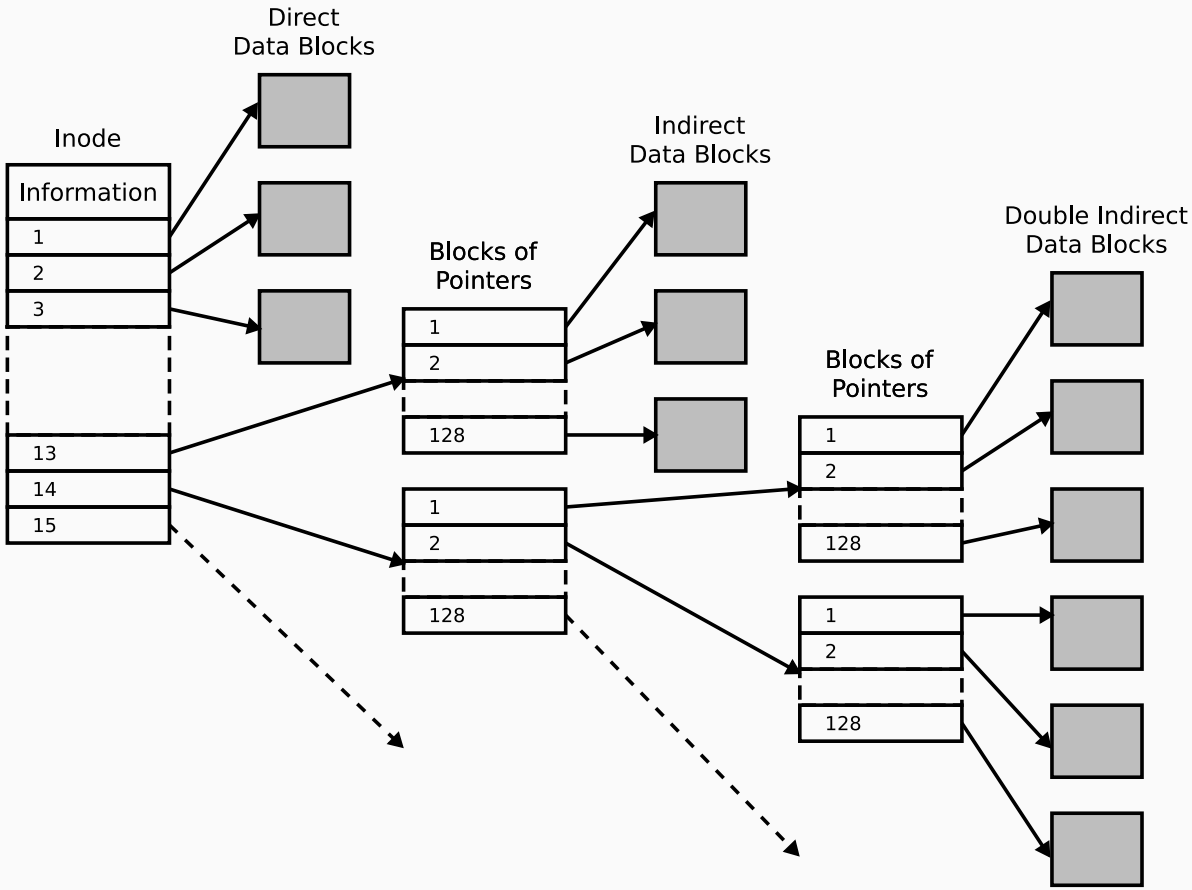
Fichiers / inodes

- 2 structures en Unix:
  - le contenu du fichier
  - les informations sur le fichier (metadata): inode
- Informations d'un inode:
  - Type de fichier
  - Nombre de liens durs (hard links) partageant l'inode
  - Longueur du fichier en octets
  - Identifiant de device
  - Identifiant de l'utilisateur (UID) propriétaire, de son groupe (GID)
  - Date de création, modification
  - Droits d'accès

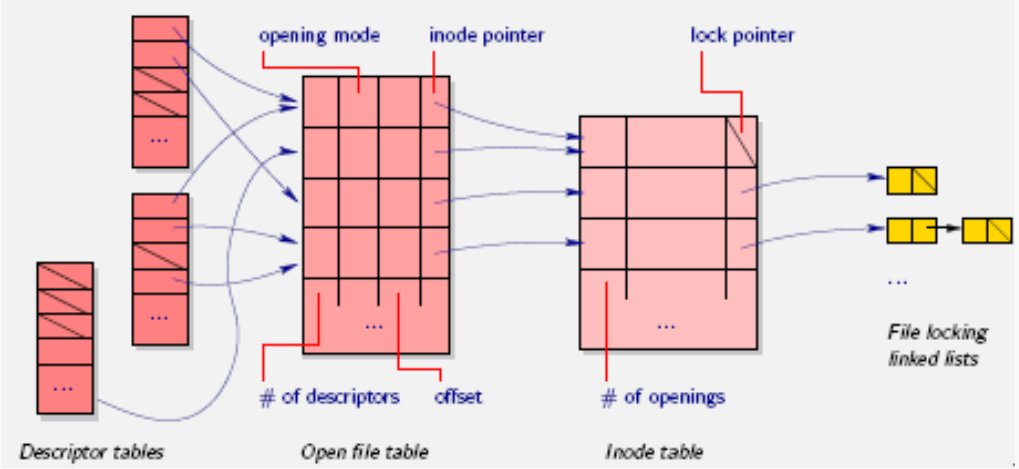
Structure des inodes

inode: regroupe les informations d'un fichier

table des blocs de données, avec accès directs ou indirects

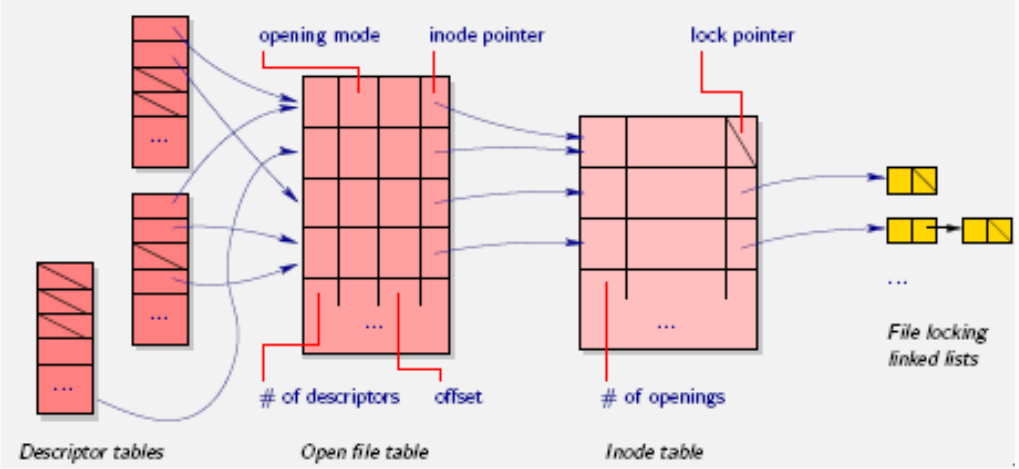


Gestion des fichiers par le système



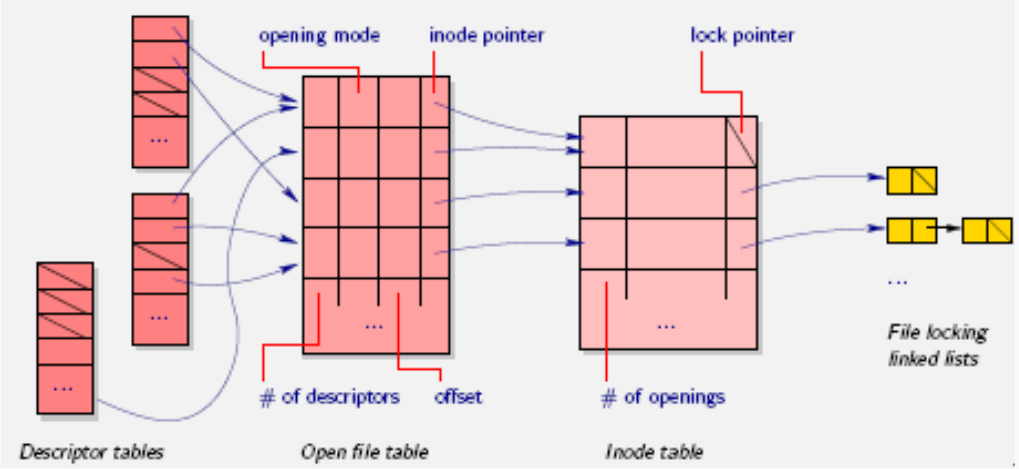
- Une table des descripteurs de fichiers ouverts/accédés par processus

Gestion des fichiers par le système



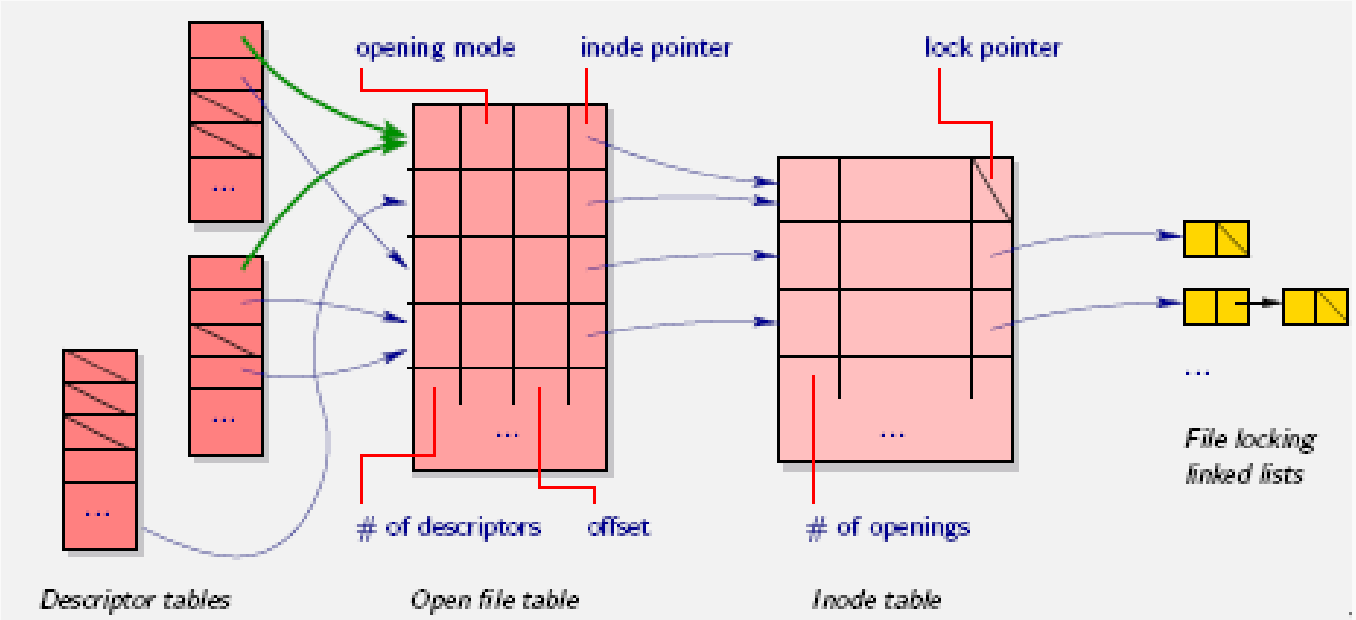
- Une table des descripteurs de fichiers ouverts/accédés par processus
- Une table des fichiers ouverts pour chaque `open` appelé. Il peut y avoir le même fichier plusieurs fois dans cette table. Il en existe une seule pour le système.

Gestion des fichiers par le système



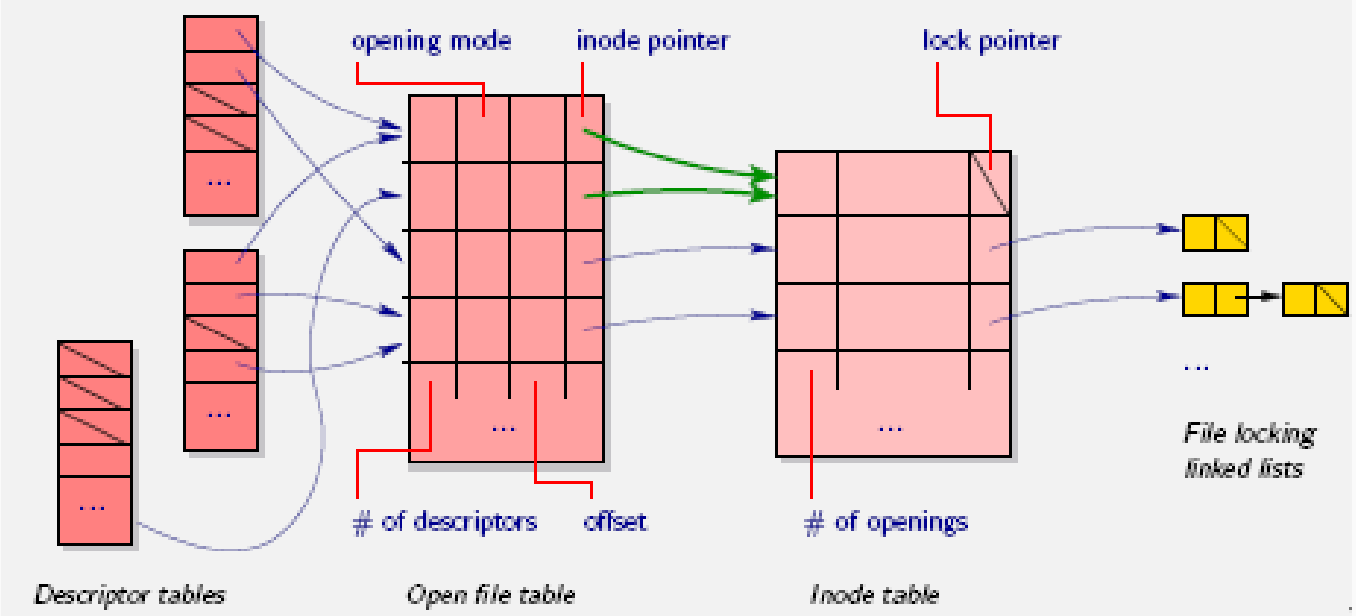
- Une table des descripteurs de fichiers ouverts/accédés par processus
- Une table des fichiers ouverts pour chaque `open` appelé. Il peut y avoir le même fichier plusieurs fois dans cette table. Il en existe une seule pour le système.
- Une seule table des inodes.
  Gère le nombre d'ouvertures de chaque fichier, et stocke les informations des fichiers en mémoire.

Gestion des fichiers par le système



Aliasing sur le descripteur de fichier: par `dup()` ou `fork()`

Gestion des fichiers par le système



Aliasing sur les inodes (fichiers ouverts), en faisant de multiples `open()` sur le même fichier.

Quelques fonctions de base

- Manipulation des inodes

  `stat()`, `unlink()`, `chown()`, `chmod()`, `mknod()`, `access()`, `link()`, `...`
- Manipulation des descripteurs de fichier

  `read()`, `write()`, `creat()`, `open()`, `lseek()`, `fcntl()`, `...`

  open crée un nouveau fichier, donc crée un nouvel inode également.
- Fonctions de bibliothèque C:

  `fopen()`, `fclose()`, `fread()`, `...`

  Ce ne sont pas des appels systèmes!!!

  Utilisent des `FILE*` au lieu d'un `id` de descripteur de fichier (`fileno()` permet de passer de l'un à l'autre).

Informations sur les fichiers

Récupérer des informations sur un inode:

```
int stat(const char * path, struct stat *buf)
```

Retourne: `0` si pas d'erreur, `-1` sinon

- `EACCES`: pb de permission dans les chemins d'accès
- `ENOENT`: chemin n'existe pas
- `ELOOP`: trop de liens symboliques dans le chemin
  (un lien pointe typiquement vers un des répertoires parent)

Structure définie par stat

```
struct stat {
 dev_t     st_dev;     /* Peripherique                  */
 ino_t     st_ino;     /* Numero i-noeud                */
 mode_t    st_mode;    /* Protection et type            */
 nlink_t   st_nlink;   /* Nb liens materiels            */
 uid_t     st_uid;     /* UID proprietaire              */
 gid_t     st_gid;     /* GID proprietaire              */
 dev_t     st_rdev;    /* Type peripherique             */
 off_t     st_size;    /* Taille totale en octets       */
 blksize_t st_blksize; /* Taille de bloc pour E/S       */
 blkcnt_t  st_blocks;  /* Nombre de blocs alloues       */
 time_t    st_atime;   /* Heure dernier acces           */
 time_t    st_mtime;   /* Heure derniere modification   */
 time_t    st_ctime;   /* Heure dernier changement etat */
};
```

Macros pour `st_mode`

Comment tester le type d'un fichier avec le champ `st_mode`?

- `S_ISREG(m)` : vrai si c'est un fichier normal
- `S_ISDIR(m)` : vrai si c'est un répertoire
- `S_ISCHR(m)` : vrai si c'est un device de type caractère
- `S_ISBLK(m)` : vrai si c'est un device de type bloc
- `S_ISFIFO(m)` : vrai si c'est une fifo
- `S_ISLNK(m)` : vrai si c'est un lien symbolique
- `S_ISSOCK(m)` : vrai si c'est une socket

Tester les autorisations d'accès

Solution alternative sans passer par `stat()` :

```
int access(const char *pathname, int mode);
```

Modes:

- `R_OK`: lecture ?
- `W_OK`: ecriture ?
- `X_OK`: exécution ?
- `F_OK`: le fichier existe ?

Retourne: `-1` en cas d'erreur ou si demande interdite, `0` sinon.

- `EROFS`: demande d'accès d'écriture à un système de fichier read-only
- `ETXTBSY`: demande d'accès d'écriture à un exécutable en cours d'exécution.

Créer un fichier (N'importe quel type)

```
int mknod(const char *pathname,
          mode_t mode, dev_t dev);
```

Mode: l'un des modes possibles de fichier, avec un *ou* logique avec les permissions

Example
`S_IFREG|S_IRUSR|S_IXUSR` pour un fichier normal

Créer/Détruire des liens

- Créer un nouveau lien vers un fichier:

```
int link(const char *oldpath, const char
                    *newpath);
```

  Pas possible de faire de nouveaux liens vers des répertoires.

- Détruire un lien dur (et le fichier si c'est le seul lien)

```
int unlink(const char *pathname);
```

  Renvoie: `0` si pas d'erreur, `-1` sinon

  - `EISDIR`: le fichier est un répertoire

Ouvrir/Créer un fichier

Ouvrir et créer un nouveau fichier:

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

Retourne: une valeur non négative de descripteur de fichier si pas d'erreur, `-1` sinon.

Flags:

- `O_RDONLY` (read), `O_WRONLY` (write), `O_RDWR` (read/write)

Mode:

- `O_CREAT`: crée le fichier si n'existe pas
- `O_APPEND`: ajoute au fichier existant
- `O_EXCL`: échoue si le fichier existe déjà
- `O_TRUNC`: si le fichier existe déjà, le vider de son contenu

Fermer un fichier

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

Ferme un descripteur de fichier.

Lorsqu'il s'agit du dernier descripteur de fichier pointant sur un fichier donné, et que celui-ci a été détruit avec unlink, alors il est effectivement détruit après cet appel.

Lire des données

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

- Lit au plus *count* octets du fichier décrit par le descripteur de fichier *fd* et les place dans le buffer *buf*
- Le buffer doit être alloué avant cet appel
- *count* ne doit pas dépasser la taille du buffer
- l'appel retourne le nombre d'octets effectivement lus ou -1 si erreur

Erreurs:

- `EINTR`: interrompu par un signal avant toute lecture

Écrire des données

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

Écrit dans le fichier donné par *fd* les *count* premiers octets du tableau *buffer*.

Retourne: `-1` si erreur ou le nombre d'octets écrits. Si ce nombre est $<$ à *count*, alors c'est une erreur.

Exemple: lecture d'un fichier

Example

```
void readfile(const char *pathname, int count, char *buf){
  int fd, byte, length=count;
  fd = open(pathname, O_RDONLY);
  if (fd == -1) {
    perror(''open() failed''); exit(1);
  }
  /* Read count bytes from the file */
  while ((byte = read(fd, buf, length)) != 0) {
    if (byte ==-1) {
      perror(''read() failed''); exit(1);
    }
    length = length-byte;
    buf = buf+length
  }
  close(fd);
}
```

Manipulation des répertoires

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

```
struct dirent {
  /* numero de l'inode */
  ino_t          d_ino;
  /* decalage vers le prochain dirent */
  off_t          d_off;
  /* longueur de cet enregistrement */
  unsigned short d_reclen;
  /* type du fichier */
  unsigned char  d_type;
  /* nom du fichier */
  char           d_name[256];
};
```

Ne pas utiliser les 4 premiers champs.

Exemple: parcours d'un répertoire

```
pDIR = opendir(".");
if ( pDIR == NULL ) {
  perror("opendir() failed");
  exit( -1 );
}
/* Get each directory entry from pDIR
   and print its name */
pDirEnt = readdir( pDIR );
while ( pDirEnt != NULL ) {
  printf( "%s\n", pDirEnt->d_name );
  pDirEnt = readdir( pDIR );
}
/* Release the open directory */
closedir( pDIR );
```

Processus et gestion mémoire

Processus et gestion mémoire

Abstraction pour le processus et la mémoire

Abstraction de processus

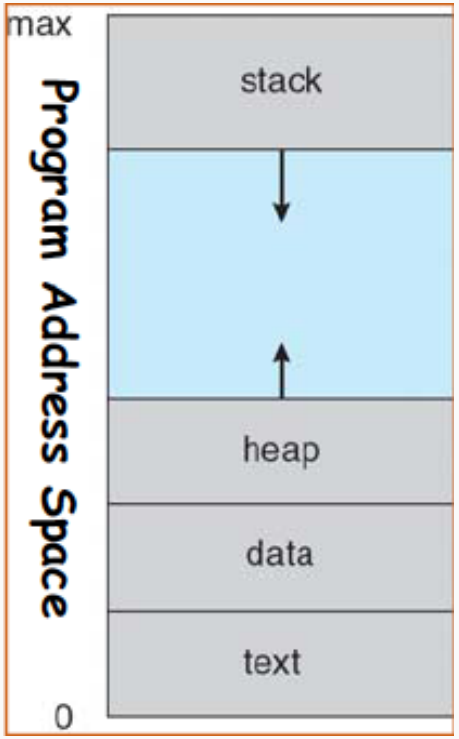
Processus
Un programme qui s'exécute ainsi que son contexte (mémoire, état des descripteurs de fichiers)

Dans les systèmes d'exploitation:

- Support pour de multiples processus
- Chaque processus a un espace d'adressage privé
  - Garantit le cloisonnement entre processus (sécurité et protection)
  - Abstraction de la mémoire (mémoire virtuelle)
  - Cache le noyau et les autres processus !

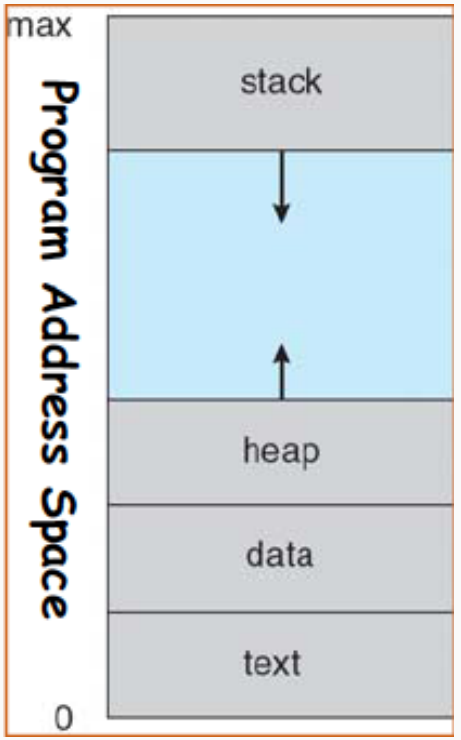
Abstraction de mémoire: adressage virtuel

- Adresse sur n bits $\rightarrow 2^n$ adresse possibles
- Espace d'adressage:
  - les adresses et cellules mémoires
  - les états associés à ces cellules
- Chaque processus ne voit que son espace d'adressage



Abstraction de mémoire: adressage virtuel

- La mémoire est initialisée pour chaque processus en différentes zones ou segments
- Stack: pile où sont stockées les variables locales
- Heap (tas): allocation dynamique, par malloc
- Data: zone de données globales ou statiques
- Texte: zone de code, des instructions du programme



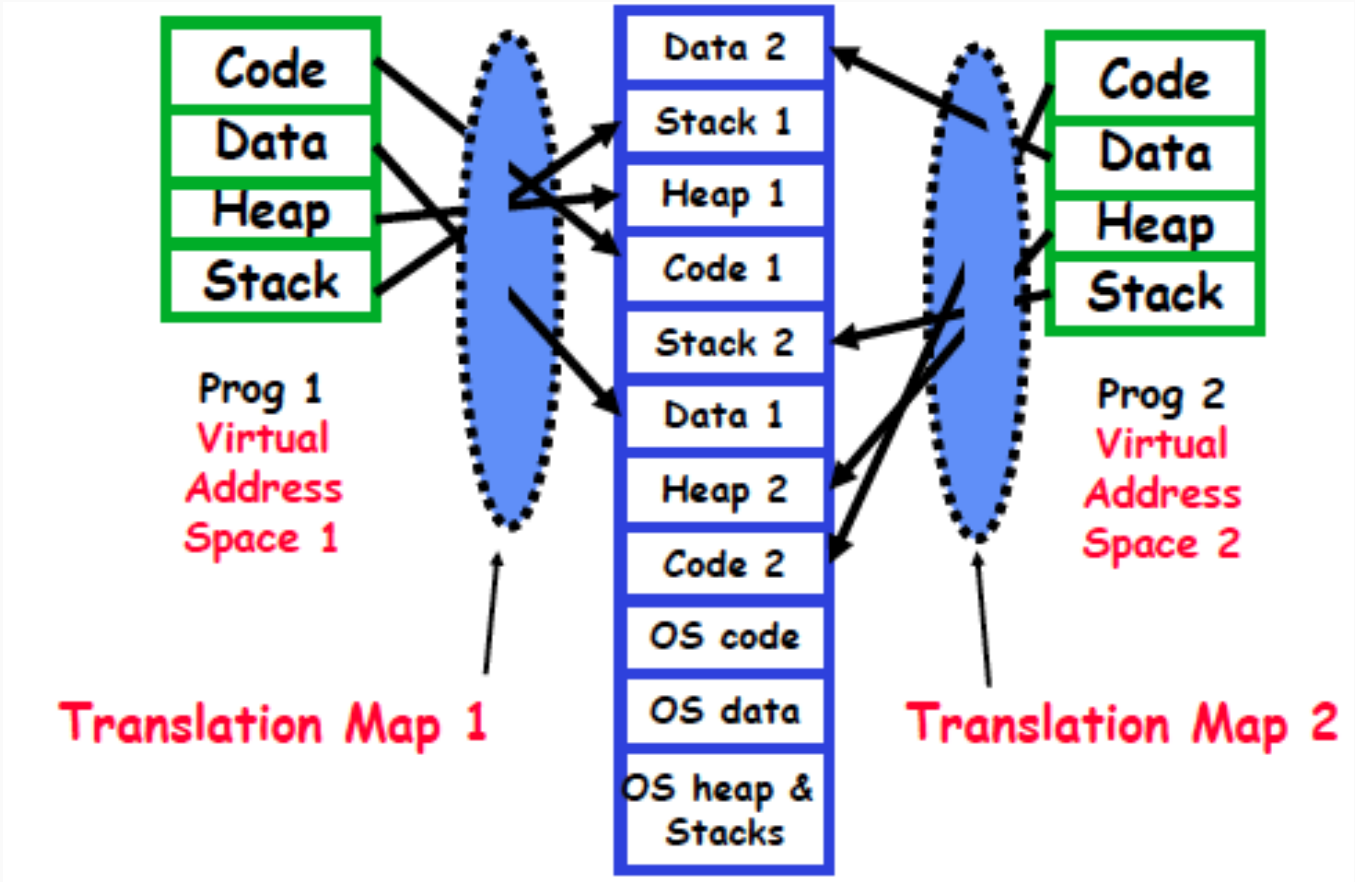
Processus et gestion mémoire

Gestion mémoire par l'OS

Lien entre mémoire virtuelle et mémoire réelle

- Traduction mémoire virtuelle $\rightarrow$ mémoire réelle faite par le CPU (MMU/TLB)
- Mécanisme de pages:
  - La mémoire est gérée par blocs de 2ko de mémoire (4Ko ou 2Mo sur x86), des pages
  - Chaque page virtuelle correspond à une page réelle
- La traduction s'appuie sur un mapping des pages (traduction par dictionnaire)
- L'adresse virtuelle est composée de différents champs permettant de construire l'adresse réelle

Lien entre mémoire virtuelle et mémoire réelle



Allocation mémoire

- Un ensemble de pages physiques sont allouée à un processus (pour les différentes zones)
- Allocation paresseuse
  - L'allocation de la page est réellement faite que lorsqu'elle est réellement accédée
  - Permet gestion plus économique de la mémoire (plutôt qu'une allocation gloutonne)
- Politique de gestion mémoire:
  - Que faire si plus de pages physiques disponibles ?
  - Rôle du swap:
    - Si plus de pages physiques, stocke sur disque les pages de processus en attente

Processus et gestion mémoire

Interface de programmation pour la gestion mémoire

Changer la taille du tas (heap)

```
int brk(void *end_segment);
int sbrk(intptr_t displacement);
```

Définit l'adresse virtuelle de la fin du tas:

- soit directement avec `brk`
- soit avec un déplacement avec `sbrk`

`brk` retourne 0 si succès, -1 sinon.
`sbrk` retourne l'adresse de fin du tas.

Ces fonctions ne sont pas à utiliser *directement*, utiliser `malloc` plutôt.

Allouer de la mémoire dans le tas

```
void *malloc(size_t size);
void *calloc(size_t count, size_t size);
void *realloc(void *p, size_t size);
```

Retourne: un pointeur sur une nouvelle zone libre de *size* octets sur le tas, *NULL* si erreur.

- Beaucoup d'OS font de l'allocation paresseuse:
  - Comportement optimiste et retourne souvent non null
- L'OS peut être amené à terminer certains processus pour libérer de la place

Affichage des différentes zones mémoire

```
int t[10];
void zones() {
  int s=0;
  void *p = malloc(10);
  printf("Code: %p Data: %p Heap: %p,%p Stack: %p",
         zones, t,
         sbrk(0), p,
         &s);
}
```

Libérer une zone mémoire allouée sur le tas

```
void free(void *ptr)
```

- Libère une zone allouée avec `malloc`
- Comportement indéfini si déjà libéré (segfault, . . .)
- ou si pas alloué avec `malloc`
- Si *ptr* est *NULL*, aucune opération n'est effectuée

Pour débugger les programmes et les pointeurs fantômes

- Gdb: précise l'endroit problématique
- Valgrind: trouve tous les mallocs sans free (fuites mémoires)

Toute la mémoire d'un processus est libérée lorsqu'il termine

Processus et gestion mémoire

Gestion de processus par l'OS

Notion de processus

Caractéristiques statiques

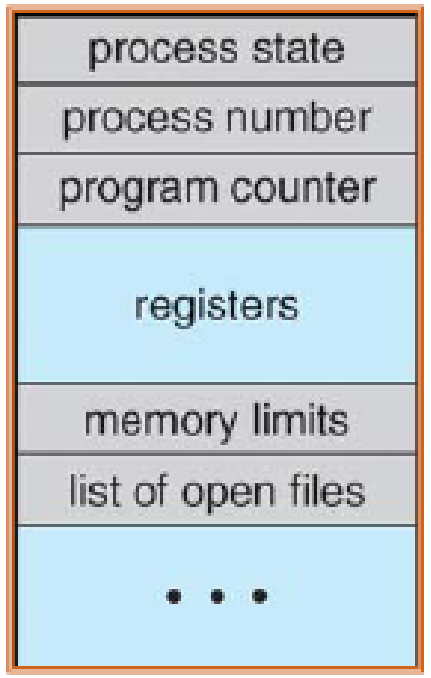
- *PID: Process Identifier*, identifiant unique d'un processus
- *PPID: Parent Process Identifier*, identifiant de l'unique du parent dans l'arborescence des processus
- Utilisateur propriétaire
- Droits d'accès aux ressources (fichiers, . . .)

Caractéristiques dynamique

- Priorité, état des registres, . . .
- Données statistiques : temps CPU consommé, temps utilisateur, temps système, . . .
- Table des descripteurs de fichiers ouverts

Structure d'un processus (PCB)

- Etat: en cours d'exécution, en attente, prêt, zombie, ...
- Etat de la machine: registres, PC
- ID, parent: pointeur vers le processus qui l'a lancé (PPID)
- Fichiers: liste des descripteurs de fichiers ouverts
- Limites: définit une limite sur le nombre de processus fils, le nombre de descripteurs de fichiers ouverts, ...
  (`man getrlimit`)



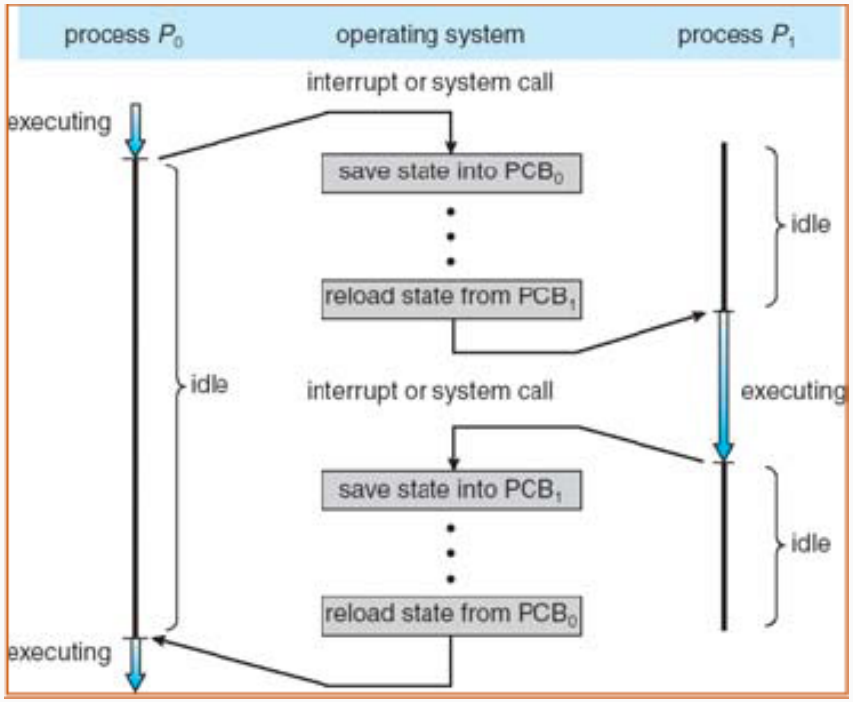
Autres informations

- Signaux: quels comportements sont prévus, pour quels signaux (sera vu après)
- Table des pages mémoire: quelles pages mémoires sont allouées par le processus
- Threads: information sur les threads (sera vu après)

Changement de contexte

Comment le processeur passe d'un processus à un autre?

- Cela s'appelle un *context switch*.
- Le temps du *context switch* doit être réduit au minimum (cf commande `time`)
- L'OS définit une politique d'ordonnancement (scheduling)



Changement de contexte

Mécanismes classiques pour le scheduling

- Mécanisme de préemption:
  - Un processus est interrompu au bout d'un quota de temps (10ms par ex.)
  - Certains OS (temps réel) ne sont pas préemptifs, les processus doivent laisser la main à l'OS volontairement.
- Algorithme d'ordonnancement:
  - Utilise des priorités
  - Tourniquet (round robin): tous les processus ont droit à un quota de temps à tour de rôle

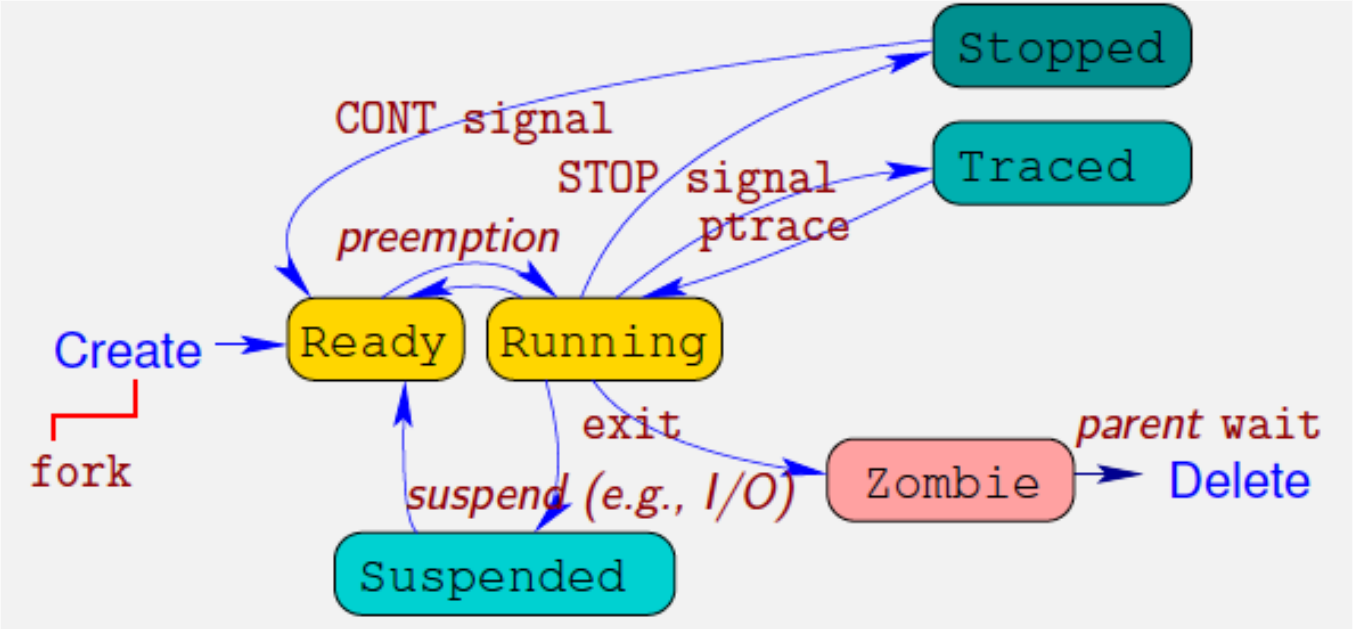
État d'un processus

- Prêt: est en attente du processeur, dépend de l'ordonnancement
- En exécution: occupe le processeur
- Suspendu: en attente d'une entrée sortie
- Stoppé, tracé: en attente d'un signal, en position d'attente (pour synchronisation avec d'autres processus par ex.)
- Zombie puis arrêté: attend que le processus parent recoive le signal de terminaison du fils puis termine (PCB détruit)

La commande `ps` donne la liste des processus en cours et leur état.
`pstree` donne la filiation entre processus.

État d'un processus

- Création: créer par clonage avec `fork()`
- Prêt: prêt à être exécuté, en attente du processeur



Processus et gestion mémoire

Interface de programmation des processus

Comment créer un processus

Sous Windows

- On lance un nouveau processus avec `CreateProcess()` en disant quel programme on veut lancer avec quelles options
- Quand on clique sur l'icône d'un programme, c'est cette fonction qui est appelée.

Sous Unix

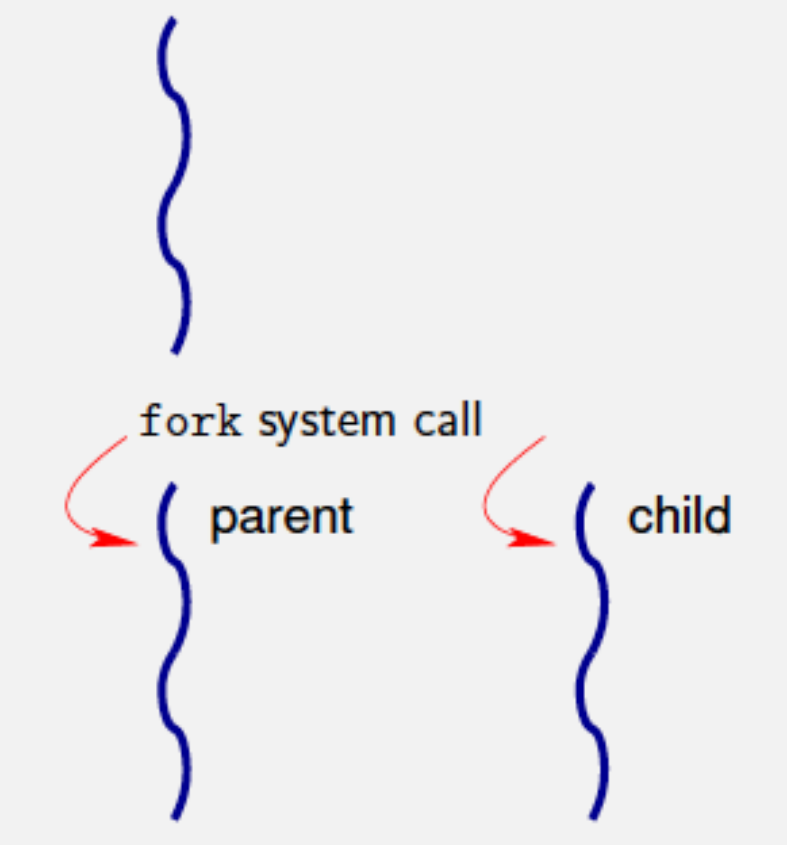
1. On crée un clone du processus courant avec `fork()`
2. Éventuellement, on change quelques paramètres
3. On appelle `exec()` pour transformer le processus en un programme différent
4. Les icônes et le shell Unix utilisent ces commandes.

Interface de programmation des processus

- Création:
  `fork()` et similaires
- Exécution:
  `exec()` et similaires
- Envoie de signaux et arrêt:
  `kill()` et autres
- Mise en attente, synchronisation:
  `wait()` et autres

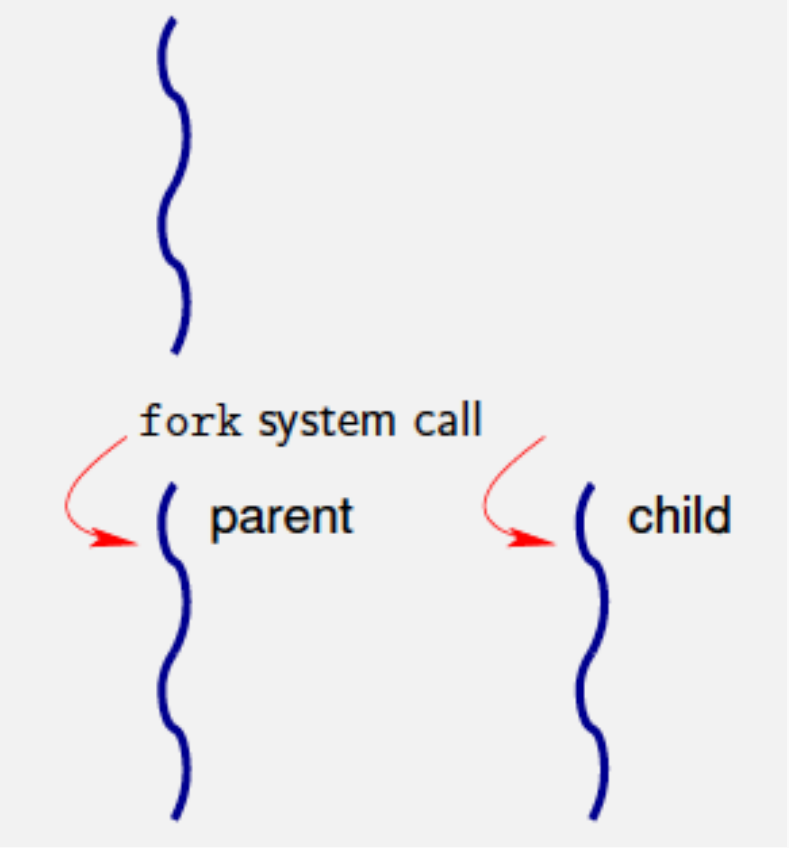
Création de processus par clonage

- Fonctions `fork()` ou `clone()`
- Le processus fils:
  - exécute le même programme
  - possède une copie de l'espace virtuel du père
- Sous linux, seule méthode de création !



Coût de création

- Création d'un Process Control Block (pas cher)
- Copie des pages (Très cher si on les copies toutes au lancement!)
- Utilisation de copie à l'écriture (copy on write): on marque la page comme devant être copiée, mais elle n'est copiée qu'à la prochaine écriture (du père ou du fils)



Création de clones: fork

```
pid_t fork();
```

Duplique le processus courant. Les 2 processus sont identiques sauf pour leur processus id respectifs: PID pour le fils, PPID pour le parent (voir `getpid()` et `getppid()`).

Le processus fils:

- NE récupère PAS les signaux en cours
- récupère les handlers et les signaux bloquant du parent

Retourne:

- 0 si c'est le fils
- Le `PID` du fils ($>$ 0) si c'est le père
- -1 si erreur

Autres variantes: `clone` (sert aussi aux threads), `vfork`.

Création de clones: fork (exemple)

```
pid_t id=fork();
switch(id) {
  case 0: // code du fils
    ...
    break;
  case -1: // erreur
    break;
  default: // code du pere
}
```

Lancement d'un exécutable (execve)

```
int execve(const char *name, char *const argv[],
           char *const envp[]);
```

Paramètres:

- *name*: le chemin et nom de l'exécutable
- *argv*: tableau des arguments qui sera passé au main du programme
- *envp*: pointeur sur environnement, obtenu par `getenv()` et manipulé par `setenv()`. Définit les variables d'environnements: CC, PATH, LD_LIBRARY_PATH, ...

Lancement d'un exécutable (execve)

```
int execve(const char *name, const *const argv[],
                    char *const envp[]);
```

Si réussi:

- Remplace les segments heap, data (statique), texte (le code) avec celles du programme chargé
- Garde les PID, PPID, descripteurs de fichiers ouverts
- Si le fichier a le bit SUID, change l'ID effective à celle du fichier
- L'appel ne retourne pas
- Sinon retourne -1

Variantes: `execl`, `execv`, `execvp`, `execlp`, `execle`, `execve`

Boot

Si tous les processus sont créés par clonage, qui est le premier?

Boot

Si tous les processus sont créés par clonage, qui est le premier?

Processus 0

Boot

Si tous les processus sont créés par clonage, qui est le premier?

Processus 0:

- Un par CPU, partagent leurs données
- Construit complètement par le noyau et tourne en mode noyau
- N'utilise que de la mémoire statique
- Construit et initialise les structures pour la gestion de la mémoire virtuelle
- Crée des threads noyaux (swap, log, . . .)
- Crée le processus 1

Boot

Si tous les processus sont crées par clonage, qui est le premier?

Processus 1:

- Un seul
- Partage ses données avec le(s) processus 0
- Finit l'initialisation du système
- Passe en mode utilisateur, devient un processus visible
- Exécute /sbin/init
- Adopte tous les processus orphelins
- Tous les autres processus sont crées à partir d'init

Terminaison d'un processus

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

Étapes de la terminaison

- Ferme tous les fichiers ouverts
- Libère toutes les pages allouées (sauf celles partagées)
- Tous les processus fils sont adoptés par le processus 1 (init)
- Le père reçoit le signal SIGCHLD (ignoré par défaut)
- Si le processus est le resp. d'une session, envoie SIGHUP à tous les autres processus de la session (les termine par défaut)

Terminaison d'un processus

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

- L'entier `status` est le code de terminaison
- Récupéré par `wait` par exemple d'un processus parent
- Récupéré par le shell. Si différent de 0, signifie une erreur

L'appel ne revient jamais

Solution plus haut niveau

```
void exit(int status);
```

Étapes de la terminaison

- Appelle toute fonction enregistrée avec `atexit()`
- Exécute les mêmes étapes que `_exit()`.

Plutôt utiliser cette fonction que l'autre qui est plus bas niveau.

Session de processus

Une session:

- Groupe de processus, indépendant de la hiérarchie de processus. Il y a un responsable de session.
- Lorsque le responsable de la session termine, les autres reçoivent SIGHUP (les termine par défaut).
- Associé aux login, terminaux, démons.

Fonctions:

`setsid()`, `getsid()`, `setgsid()`, `tcgetsid()`, ...

Créer une session

```
pid_t setsid();
```

Si le processus n'est pas responsable de session:

- cela crée une session dont il est le responsable
- le processus n'a plus de terminal le contrôlant
- retourne l'ID de la session (le PID du processus)

S'il est déjà responsable de session:

- retourne -1.

Créer un service (daemon)

- Un processus de service est détaché de tout terminal et session
- Procédure pour transformer un processus:
  - Appeler `fork()` dans le processus
  - Terminer le processus père (`exit()`)
  - Appeler `setsid()` pour rendre le processus fils indépendant

Signaux

Signaux

Evénements gérés par les processus et générés par les processus ou l'OS.

Provoque des changements d'états d'un processus

Signaux

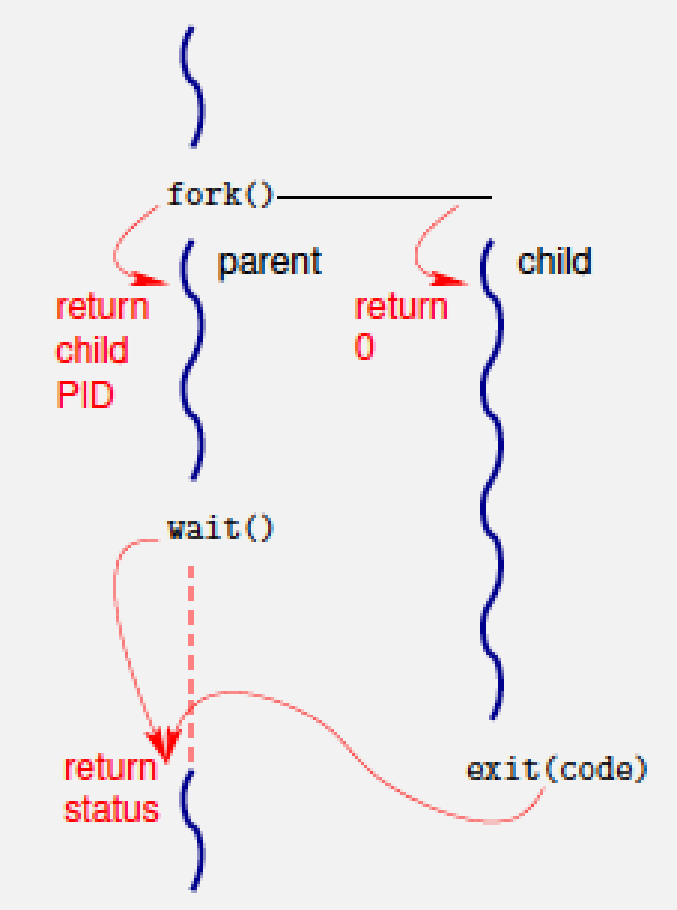
Surveillance de processus fils, gestion erreurs

Attente de signal d'un processus fils

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

- Attend les changements d'état d'un processus fils et retourne le PID
- D'un processus fils ayant terminé
- D'un fils stoppé ou ayant repris l'exécution par un signal
- Un processus termine $\rightarrow$ passe dans l'état zombie jusqu'à ce que son père recoive son signal de terminaison
- Si pas de père, init l'adopte
- init attend ses enfants

Attente de signal d'un processus fils



Valeur de pid dans `waitpid(pid, &status, option)` ?

- *pid* $>$ 0: waitpid suspend l'exécution jusqu'à un changement d'état du fils *pid*
- *pid* $=$ 0: attend pour n'importe quel processus dans le même groupe (`setsid()`)
- *pid* <-1: attend pour n'importe quel processus dans le groupe $-$*pid*
- *pid* $==-1$: attend pour n'importe quel processus fils.

`wait(&status)` *est équivalent à* `waitpid(-1, &status, 0)`.

waitpid: quelles options?

- `WNOHANG`: ne pas bloquer si aucun fils ne change d'état. Retourne 0 dans ce cas.
- `WUNTRACED`: détecte qu'un fils est arrêté (avec un signal `SIGSTOP, SIGTSTP, SIGTTIN ou SIGTTOU`)
- `WCONTINUED`: détecte qu'un fils a repris l'exécution (avec un signal `SIGCONT`)

waitpid: quels status?

Si valeur non nulle:

- `WIFEXITED(status)` : vrai si terminaison normale (`_exit()`)
  `WEXITSTATUS(status)` =valeur retournée par `_exit`
- `WIFSIGNALED(status)` : vrai si terminé par signal
  `WTERMSIG(status)` =signal ayant terminé le processus
- `WIFSTOPPED(status)` : vrai si stoppé
  `WSTOPSIG(status)` =signal ayant stoppé le processus
- `WIFCONTINUED(status)` : vrai si exécution reprise par le signal `SIGCONT`.

waitpid: exemple

```
int status;
p = fork();
switch(p) {
  case -1: perror("fork"); exit(1);
  case 0: pause(); // Attend les signaux
  default:
  if ( waitpid(p, &status, WUNTRACE | WCONTINUED) == -1 ) {
    perror("waitpid"); exit(1);
  }
  if ( WIFEXITED(status) ) {
    printf("exited with value %d\n", WEXITSTATUS(status));
  }
  else if ( WIFCONTINUED(status) ) {
    printf("continued\n"); // Marche sous linux 2.6
  }
  else if ( WIFSTOPPED(status) ) {
    printf("stopped by signal %d\n", WSTOPSIG(status));
  }
  else if ( WIFSIGNALED(status) ) {
    printf("killed by signal %d\n", WTERMSIG(status));
  }
}
```

Signaux

Synchronisation entre processus

Synchronisation

Principe de l'envoi/reception de signaux

- Envoi et la réception sont asynchrones
- Envoi pendant exécution du processus cible ou pas
- Réception du signal n'importe où dans l'exécution

Gestionnaire de signal (signal handler)

- Est appelé lors de la réception du signal

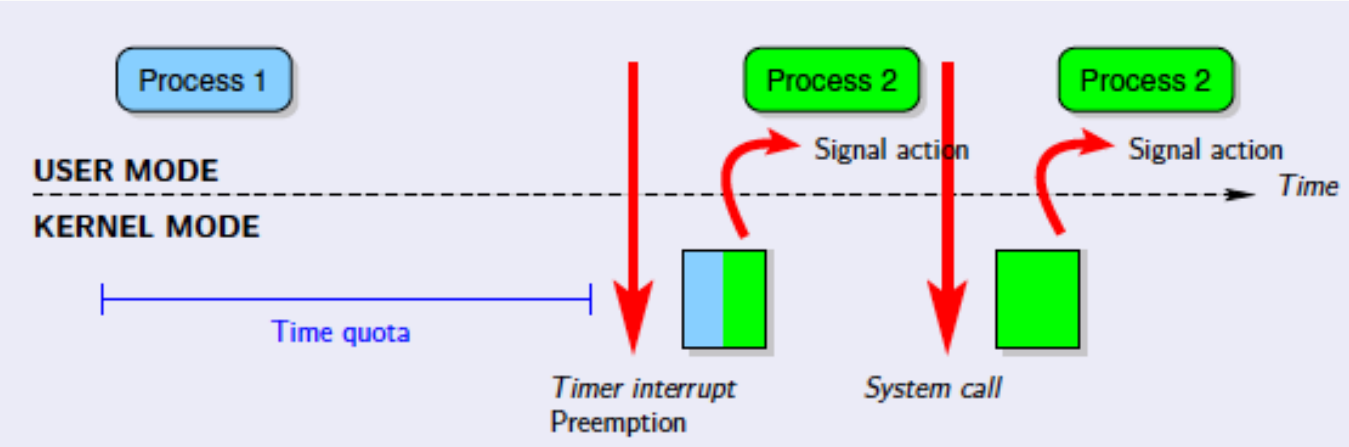
Signal en attente Reçu mais pas géré

- Soit bloqué
- Soit le noyau n'a pas vérifié son statut (bloqué ou pas)
  !! Pas de file d'attente pour les signaux!!

Capture du signal (Quand?)

Lors d'un changement du mode noyau vers mode utilisateur:

- Retour appel système
- Retour d'un context switch (changement de contexte)



Envoyer un signal

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

Envoie le signal `sig` au(x) processus `pid`

Quelle valeur possible pour pid ?

- $pid > 0$: envoie au processus pid
- $pid = 0$: envoie à tous les processus du groupe du processus courant
- $pid < -1$: envoie à tous les processus du groupe -pid
- $pid = -1$: envoie à tous les processus auxquels on a le droit d'envoyer un signal, sauf soi même et init

Valeurs de retour

- 0 si succès, -1 si échec, avec comme errno:
- `EINVAL`: numéro de signal invalide
- `EPERM`: pas le droit d'envoyer ce signal à ce processus
- `ESRCH`: le processus (ou groupe) n'existe pas

Envoyer un signal

Valeurs de signaux:

terminent le processus, non-masquables et non bloquables

- `SIGHUP`: signal de terminaison du terminal
- `SIGINT`: Ctrl-C
- `SIGQUIT`: Ctrl-
- `SIGKILL`: terminaison. Ne peut pas être bloqué
- `SIGBUS / SIGSEGV`: memory bus error, segment violation
- `SIGPIPE`: FIFO cassée (écriture sans lecture)
- `SIGALRM`: signal d'alarme
- `SIGSTOP, SIGSTP`: suspend l'exécution
- `SIGCONT`: reprend l'exécution
- `SIGCHLD`: fils terminé ou stoppé
- `SIGUSR1 / SIGUSR2`: signaux utilisateurs

Réception des signaux (deprecated)

```
sig_t signal(int sig, sig_t func);
      typedef void (*sig_t) (int);
```

Installe un nouveau gestionnaire pour le signal *sig*.

Retourne l'ancien gestionnaire de ce signal (ou SIG_ERR)

Gestionnaires prédéfinis:

- `SIG_DFL`: gestionnaire par défaut
- `SIG_IGN`: signal ignoré

Gestion des signaux

Exécution du gestionnaire de signal

- Reçoit la valeur du signal responsable
- Bloque la réception d'autres signaux identiques
- Attention aux appels système utilisés dans le gestionnaire (certains ne sont pas *safe* dans gestionnaire de signaux, man 2 signal, dépend de l'OS)

Retour du gestionnaire:

- L'exécution du processus reprend où elle était
- Certains appels système interrompus sont automatiquement relancés (open, write, . . . dépend de l'OS)

Attente de signaux

```
int pause();
```

- Suspend l'exécution jusqu'à la reception d'un signal:
  - Soit le signal termine le processus (pause ne revient pas),
  - Soit il appelle un gestionnaire de signal.
- Les signaux ignorés (SIG_IGN) n'interrompent pas pause.

Retourne toujours -1.

Programmer une alarme / Mettre en sommeil

```
int alarm(unsigned int s);
```

- Envoie le signal SIGALRM au processus appelant après s secondes.
- Par défaut: le signal termine le processus

```
int sleep(unsigned int s);
```

- Mets le processus en attente pendant *s* secondes.
- Utilise pause, signal et alarm
  $\rightarrow$ utilise le même timer qu'alarm, ne pas mélanger les deux.

Signaux

Exemples

Sleep avec pause()

```
// Gestionnaire
void rien(int sig) {
}

void mysleep(unsigned int s) {
  // Installe le nouveau gestionnaire
  signal(SIGALRM, rien);
  // Lance le signal dans s secondes
  alarm(s);
  // Attend le signal d'alarme
  pause();
  // Restaure l'ancien gestionnaire
  signal(SIGALRM,SIG_DFL);
}
```

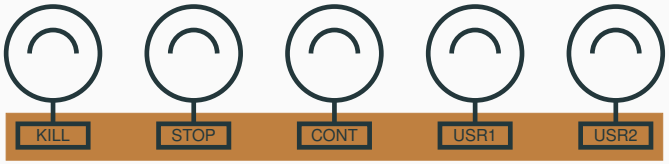
Ping-pong

```c
char *string;

void ping(pid_t p) {
  // Gestionnaire
  signal(SIGUSR1, ping_sh);
  for (int i=0; i<10; i++) {
    if (p==0) {
      // Processus fils
      kill(getppid(), SIGUSR1);
      pause(); sleep(1);
    }
    else {
      // Processus pere
      pause(); sleep(1);
      kill(p, SIGUSR1);
    }
  }
}
```

```c
void ping_sh(int s){
  printf("%s\n", string);
}

int main(){
  pid_t p;
  // Creation 2 processus
  p = fork();
  switch(p) {
  case -1:
    perror("fork");
    exit(1);
  case 0:
    string="pong";
    break;
  default:
    string="ping";
  }
  ping(p);
  return 0;
}
```

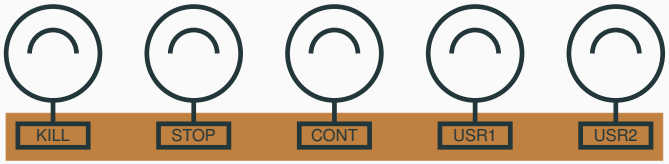
Explications (Double réception)

1. Etat Initial:

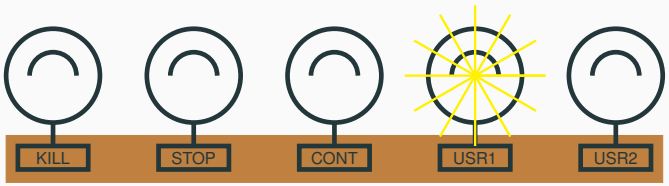


Explications (Double réception)

1. Etat Initial:

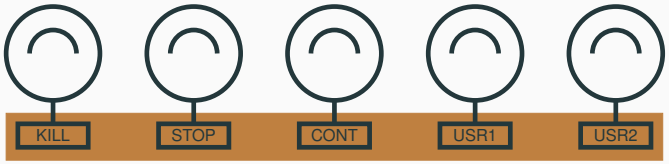


2. Réception de SIGUSR1:

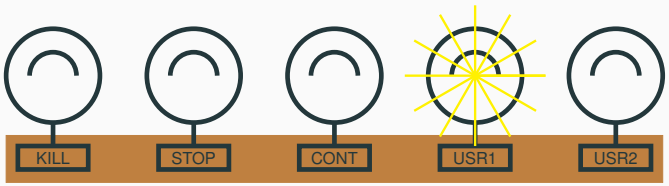


Explications (Double réception)

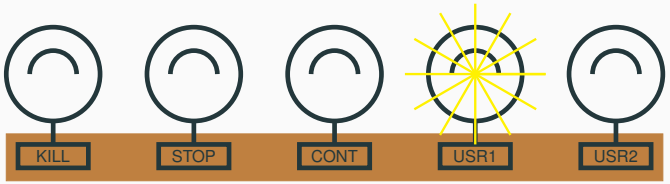
1. Etat Initial:



2. Réception de SIGUSR1:

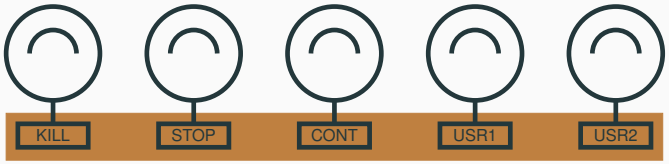


3. Seconde réception de SIGUSR1:

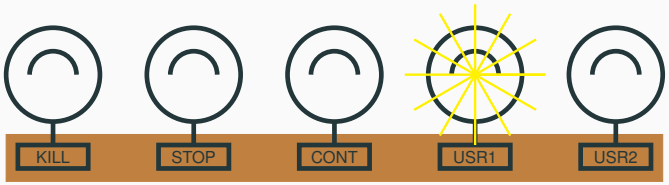


Explications (Double réception)

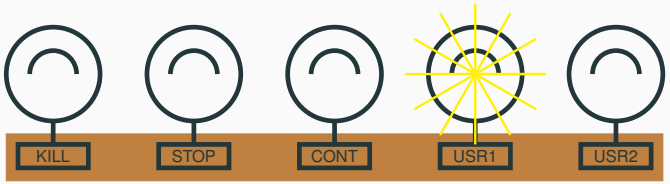
1. Etat Initial:



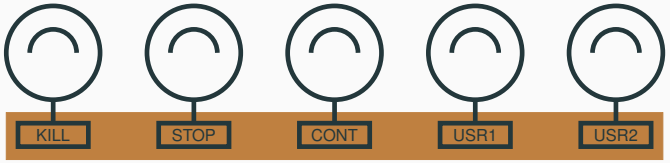
2. Réception de SIGUSR1:



3. Seconde réception de SIGUSR1:

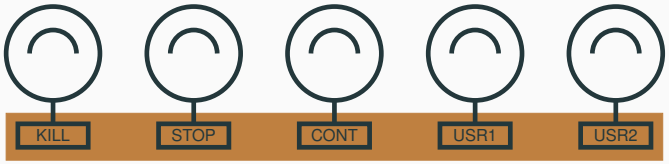


4. Traitement de SIGUSR1:

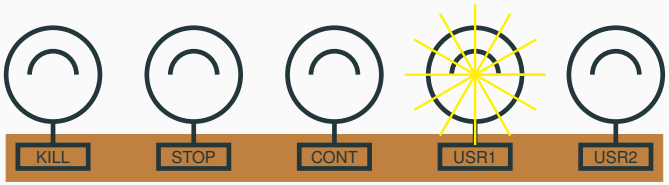


Explications (Double réception)

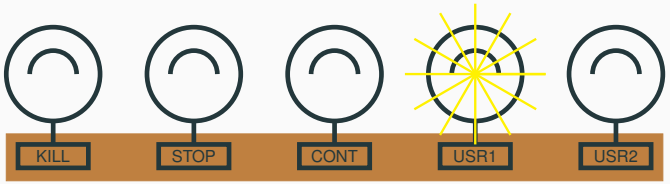
1. Etat Initial:



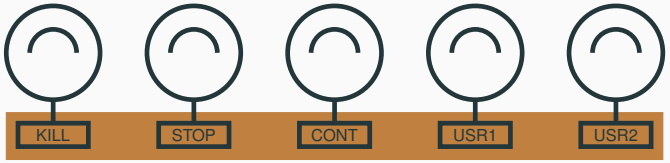
2. Réception de SIGUSR1:



3. Seconde réception de SIGUSR1:



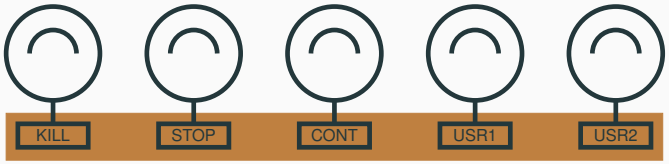
4. Traitement de SIGUSR1:



Signal reçu deux fois, mais traité **une seule** fois

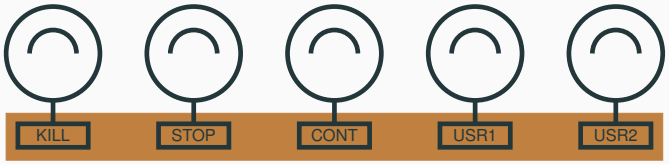
Explications (Installation tardive)

1. Etat initial

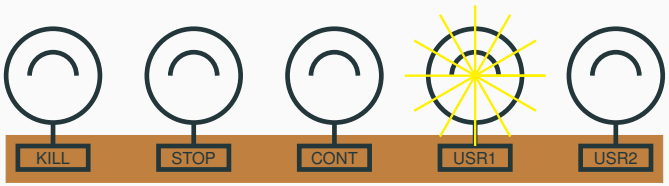


Explications (Installation tardive)

1. Etat initial

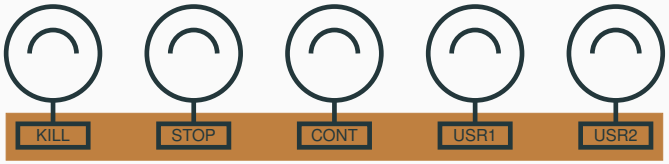


2. Réception de SIGUSR1

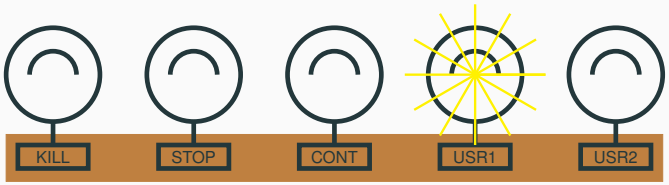


Explications (Installation tardive)

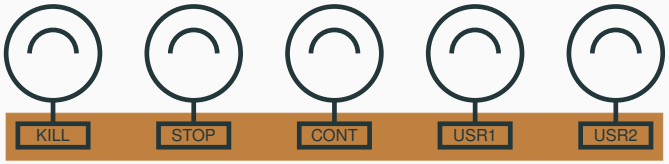
1. Etat initial



2. Réception de SIGUSR1

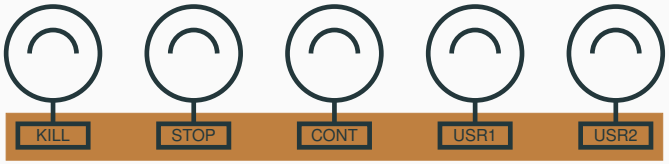


3. Traitement de SIGUSR1

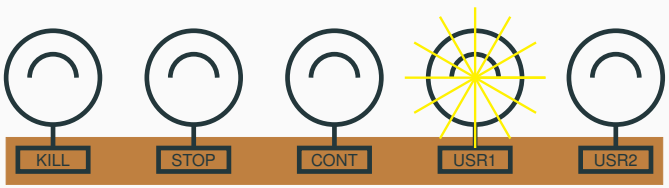


Explications (Installation tardive)

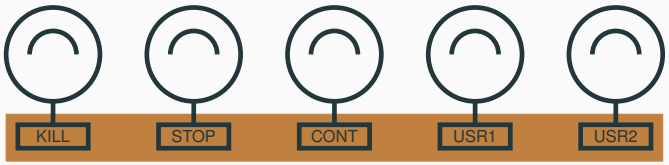
1. Etat initial



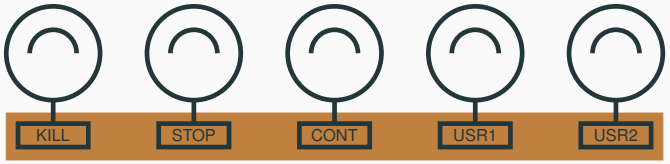
2. Réception de SIGUSR1



3. Traitement de SIGUSR1

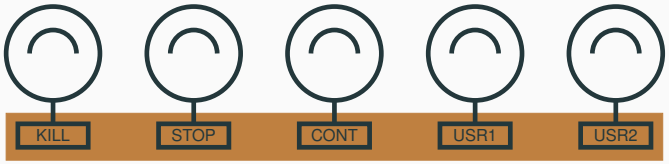


4. Installation du handler ou appel à PAUSE()

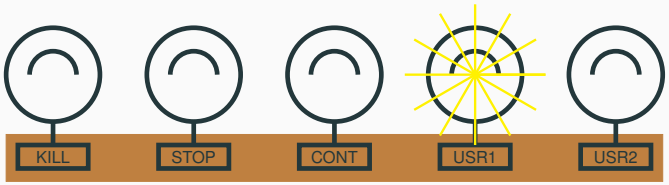


Explications (Installation tardive)

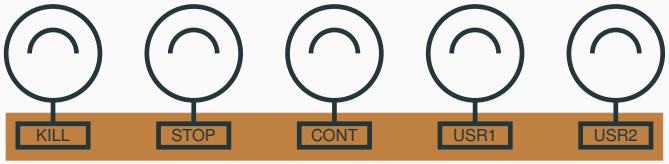
1. Etat initial



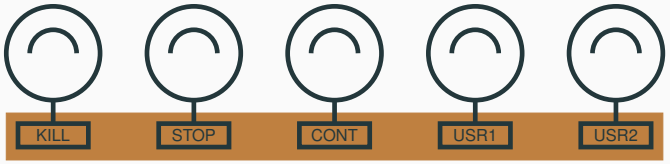
2. Réception de SIGUSR1



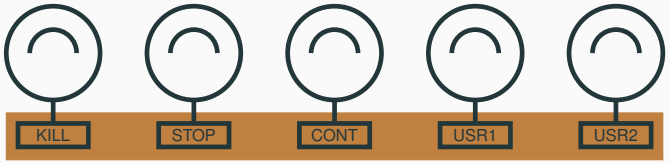
3. Traitement de SIGUSR1



4. Installation du handler ou appel à PAUSE()

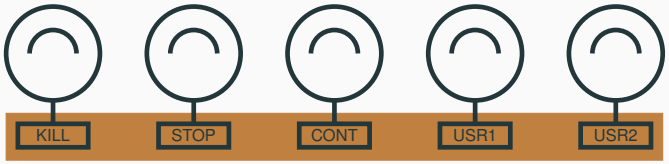


5. Attente du signal SIGUSR1

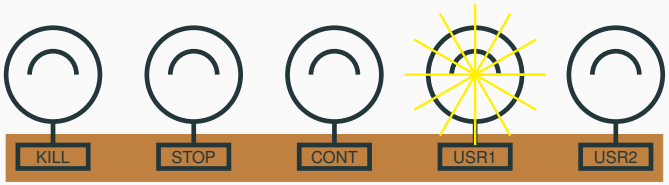


Explications (Installation tardive)

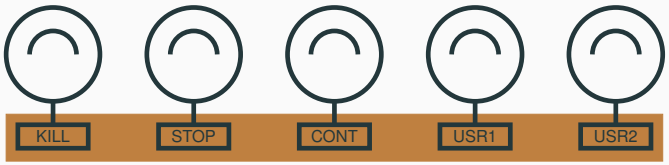
1. Etat initial



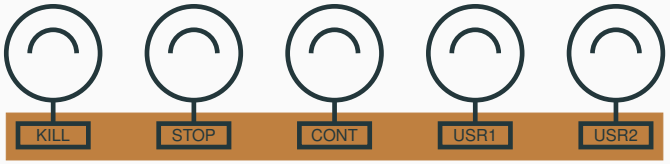
2. Réception de SIGUSR1



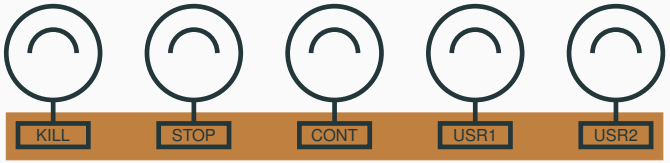
3. Traitement de SIGUSR1



4. Installation du handler ou appel à PAUSE()



5. Attente du signal SIGUSR1



Deadlock

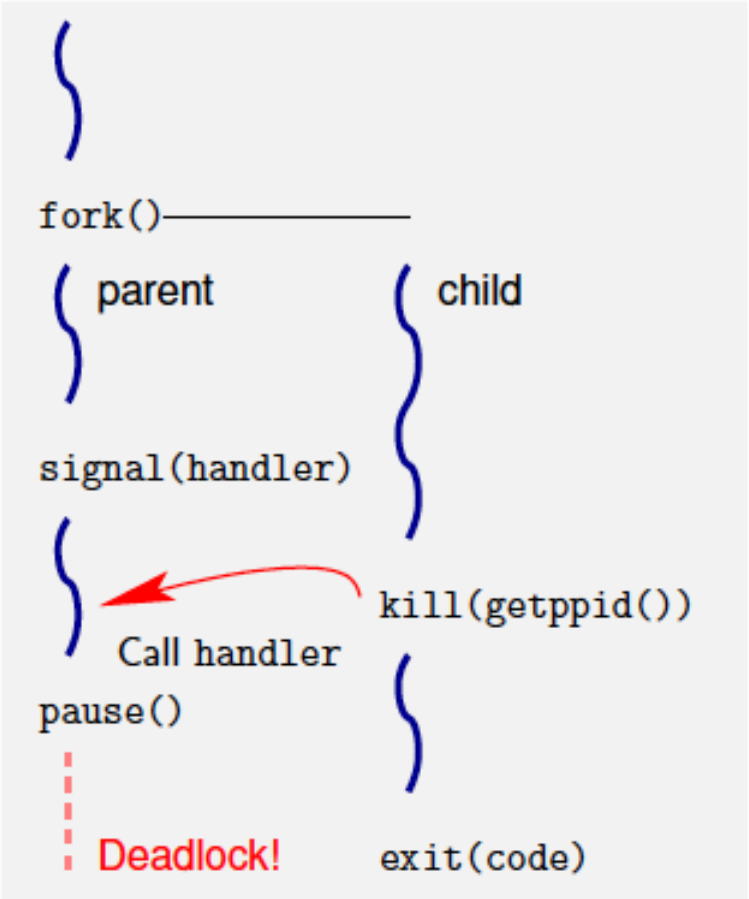
Ne **jamais** utiliser
`signal()/pause()` !!!

Signaux

Synchronisation avancée

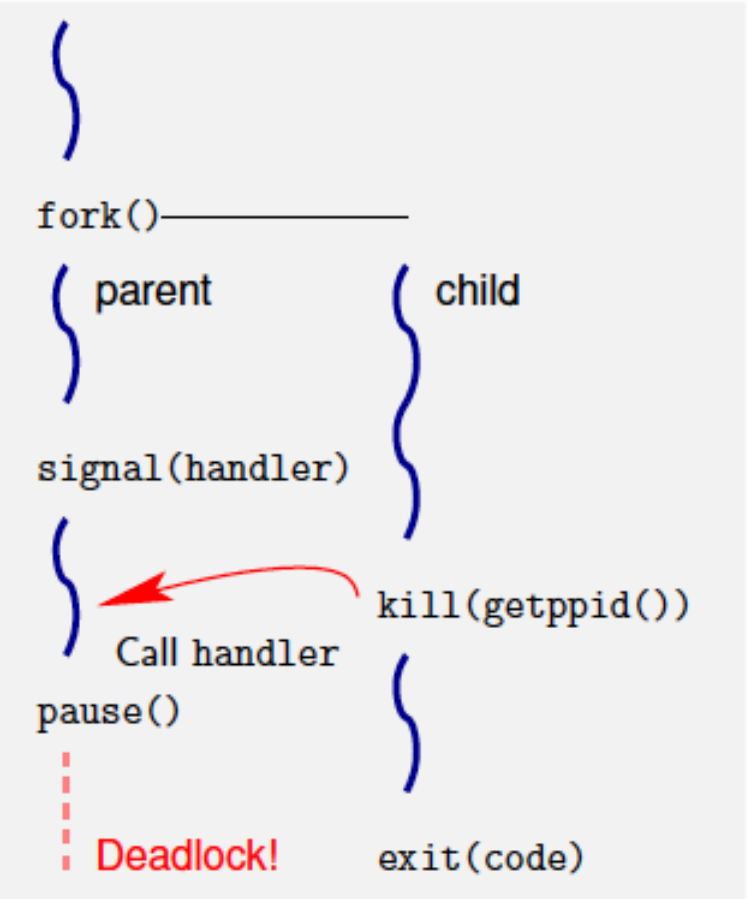
Problème de synchronisation possible

Que se passe-t-il si un processus reçoit un signal avant de l'attendre avec pause?



Problème de synchronisation possible

Que se passe-t-il si un processus reçoit un signal avant de l'attendre avec pause?
$\rightarrow$ Bug difficile à détecter et à reproduire.



Solution

- Masquer (bloquer) le signal qu'on veut gérer
- Mettre en place le gestionnaire
- Débloquer le signal et attendre ce signal
- Le signal ne doit pas arriver entre le moment où on l'a débloqué et le moment où on l'attend.
- Cette opération doit être atomique

L'interface de programmation vue pour l'instant ne suffit pas

Définir un gestionnaire de signaux

```
int sigaction(int signum, struct sigaction *act,
              struct sigaction *old);
```

- Examine et change le gestionnaire de signaux:
- Si act est non nul, installe ce gestionnaire pour le signal signum
- Si old est non nul, récupère l'ancien gestionnaire pour le signal signum
- Retourne:
  - 0 si pas d'erreur, -1 sinon avec errno
  - EINVAL: Signal invalide ou tentative de changer le gestionnaire pour SIGKILL ou SIGSTOP

Structure du gestionnaire de signaux

```
struct sigaction {
  // Meme chose que pour signal
  void (*sa_handler)(int);
  // Gestionnaire de signal, avec
  // information sur son contexte d'execution
  void (*sa_sigaction)(int, siginfo_t *, void *);
  // Ensemble des signaux bloques pendant
  // execution gestionnaire
  sigset_t sa_mask;
  // Options pour le gestionnaire
  int sa_flags;
}
```

Soit `sa_handler` est défini, soit `sa_sigaction`, pas les deux.

Quelques options du gestionnaire

- `SA_NOCLDSTOP`: pour le signal SIGCHLD (signal d'info d'un fils à son père), ne capturera le signal que sur terminaison du fils, pas pour un arrêt (SIGSTOP) ou une reprise (SIGCONT) du fils
- `SA_RESTART`: relance certains appels système interrompus par le signal (open, write, read, ...
- `SA_SIGINFO`: utilise le gestionnaire indiqué par sa_sigaction au lieu de sa_handler
  man 2 sigaction pour plus de (nombreux) détails sur cette interface

Fonctions de manipulation ensemble de signaux `sigset_t`

- `int sigemptyset(sigset_t *set);`
- `int sigfillset(sigset_t *set);`
- `int sigaddset(sigset_t *set, int signum);`
- `int sigdelset(sigset_t *set, int signum);`
- `int sigismember(sigset_t *set, int signum);`

Vide, remplit (avec tous les signaux), ajoute un signal ou en enlève un d'un ensemble de signaux passé par référence.

Test si un signal est dans un ensemble.

```
man 2 sigsetops
```

Exemple type

```
int count_signal = 0;
// Fonction du gestionnaire de signal
void count(int signum) {
  count_signal++;
}
int main() {
  // Masque de signaux
  sigset allsig;
  sigfillset( &allsig );
  // Gestionnaire de signal
  struct sigaction sa;
  // Fonction a appeler a la reception du signal
  sa.sa_handler = count;
  // Signaux a masquer pendant l'execution de la fct
  sigemptyset(&sa.sa_mask);
  sa.sa_flags=0; // Pas d'options
  // Mettre en place le nouveau gestionnaire
  sigaction(SIGUSR1, &sa, NULL);
  while (1) {
    printf("count signal = %d\n",count_signal);
    sigsuspend( &allsig );
  }
}
```

Examine et bloque des signaux

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

Si set est non null, how décrit l'opération:

- `SIG_BLOCK`: les signaux dans l'ensemble set sont bloqués (en plus de ceux déjà bloqués)
- `SIG_UNBLOCK`: les signaux dans l'ensemble set sont débloqués (en plus de ceux déjà bloqués)

Si old n'est pas nul, récupère l'ancien ensemble de signaux bloqués.

Tentatives de bloquer SIGKILL ou SIGSTOP sont ignorées

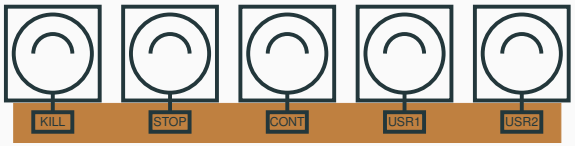
Débloquer des signaux et suspendre l'exécution

```
int sigsuspend(sigset_t *set);
```

- Fait les opérations suivantes de façon atomique
- L'ensemble des signaux bloqués est celui défini par set (utile pour débloquer des signaux)
- Suspend l'exécution du processus, comme `wait()`.
- Usage typique:
  - Appel de sigprocmask pour bloquer des signaux
  - Fait des opérations critiques (installer le gestionnaire par ex)
  - Appel de sigsuspend pour débloquer les signaux et suspendre l'exécution, en attente de signaux

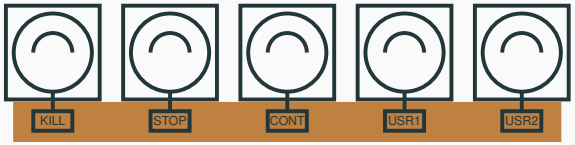
Exemple du ping/pong revu et corrigé

1. Etat Initial:

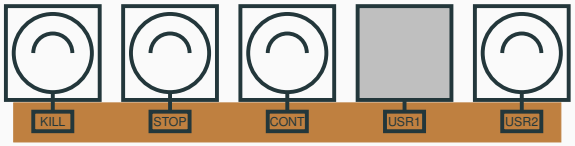


Exemple du ping/pong revu et corrigé

1. Etat Initial:

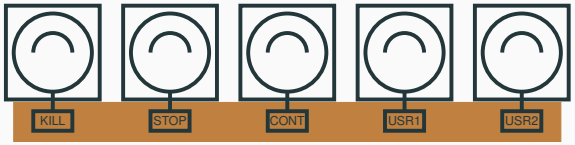


2. Masquer le signal attendu (`sigprocmask`)

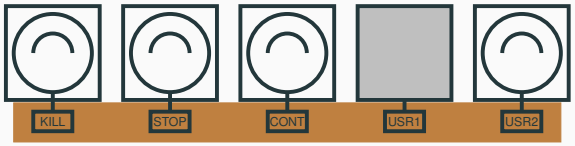


Exemple du ping/pong revu et corrigé

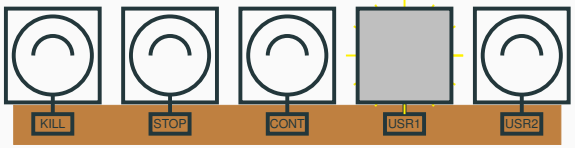
1. Etat Initial:



2. Masquer le signal attendu (`sigprocmask`)

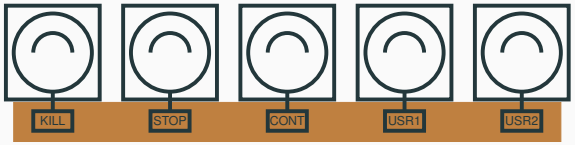


3. Reception de SIGUSR1 (une ou plusieurs fois)

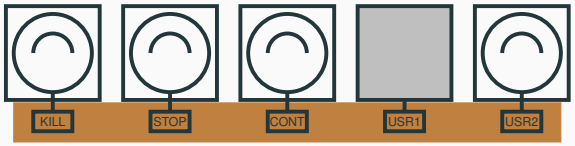


Exemple du ping/pong revu et corrigé

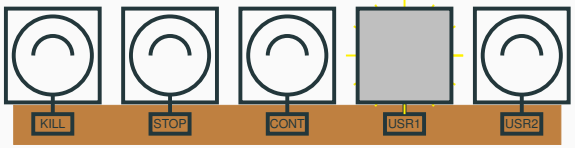
1. Etat Initial:



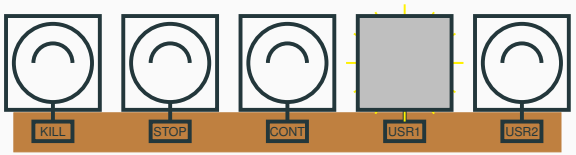
2. Masquer le signal attendu (`sigprocmask`)



3. Reception de SIGUSR1 (une ou plusieurs fois)

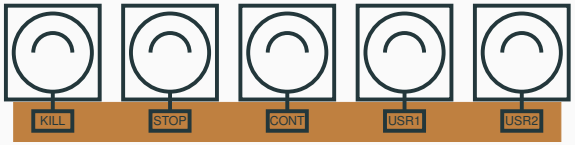


4. Installation du handler (`sigaction`)

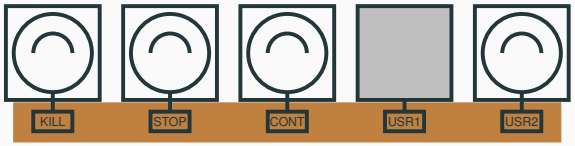


Exemple du ping/pong revu et corrigé

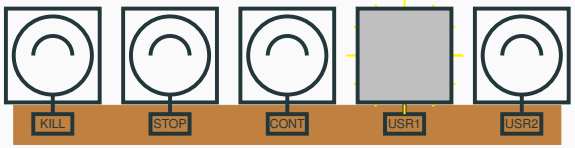
1. Etat Initial:



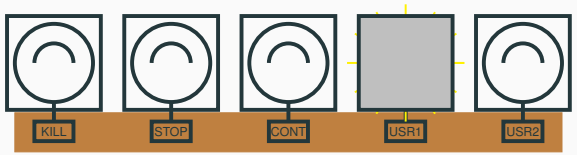
2. Masquer le signal attendu (`sigprocmask`)



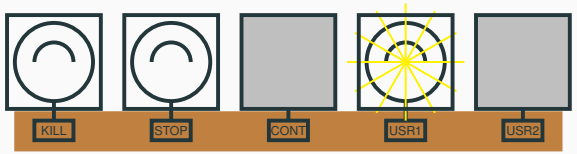
3. Reception de SIGUSR1 (une ou plusieurs fois)



4. Installation du handler (`sigaction`)

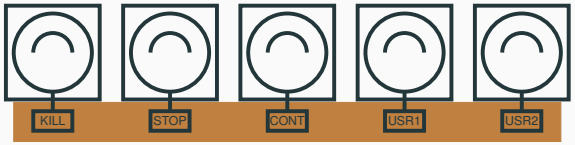


5. Attente et démasquage des signaux (`sigsuspend`)

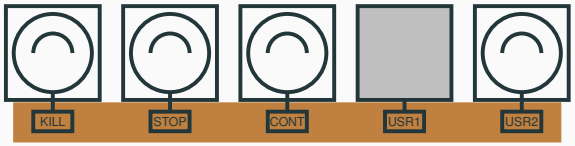


Exemple du ping/pong revu et corrigé

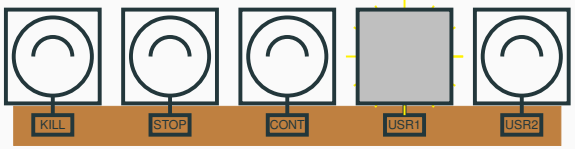
1. Etat Initial:



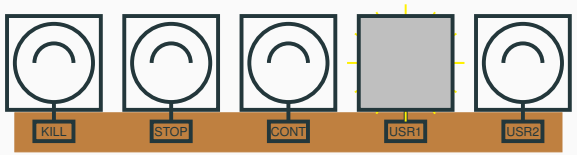
2. Masquer le signal attendu (`sigprocmask`)



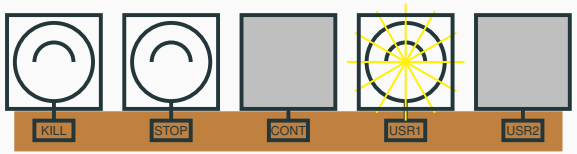
3. Reception de SIGUSR1 (une ou plusieurs fois)



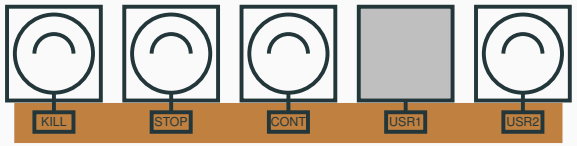
4. Installation du handler (`sigaction`)



5. Attente et démasquage des signaux (`sigsuspend`)

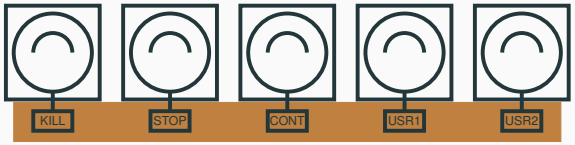


6. Traitement de SIGUSR1 et retour

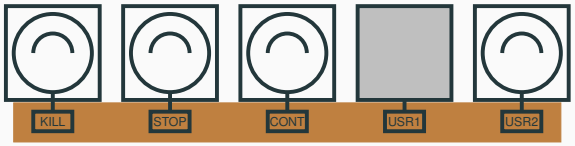


Exemple du ping/pong revu et corrigé

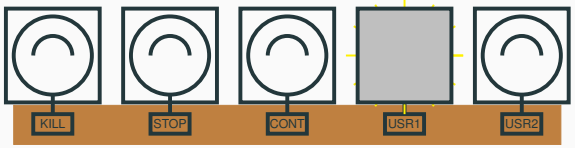
1. Etat Initial:



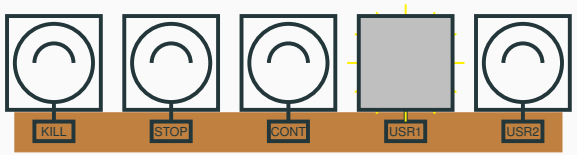
2. Masquer le signal attendu (`sigprocmask`)



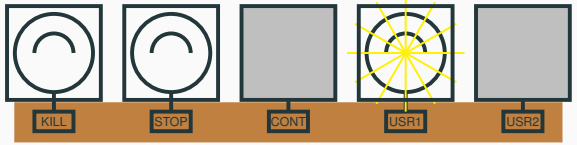
3. Reception de SIGUSR1 (une ou plusieurs fois)



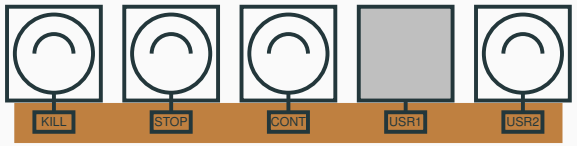
4. Installation du handler (`sigaction`)



5. Attente et démasquage des signaux (`sigsuspend`)



6. Traitement de SIGUSR1 et retour



Tout fonctionne !!!

Mémoire partagée et synchronisation

Mémoire partagée

Mémoire virtuelle

- Segmente la mémoire en pages
- Fait correspondre des pages de mémoire virtuelle à des pages physiques
- Isole chaque processus de l'action des autres

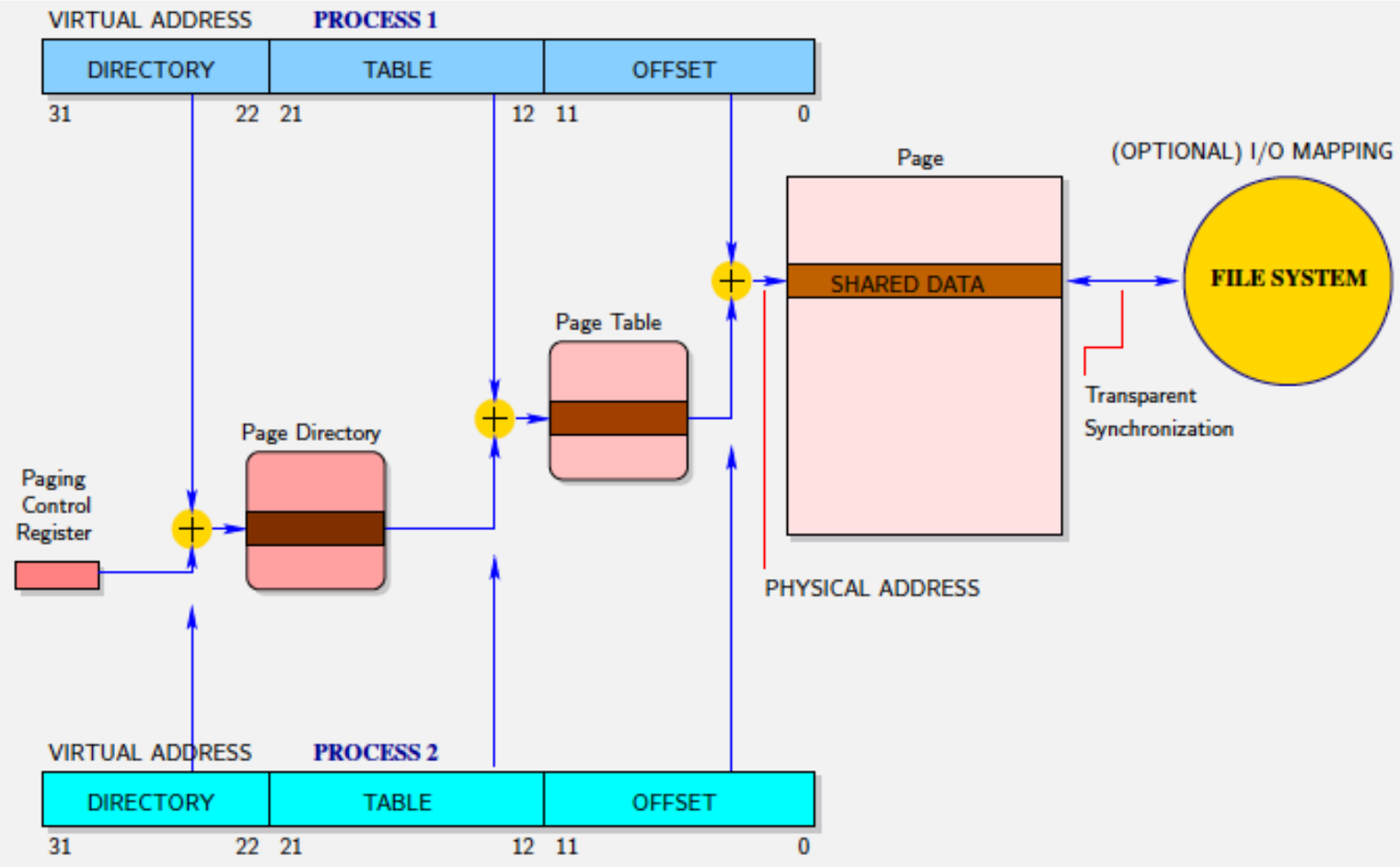
Partage de mémoire

Des pages virtuelles de processus différents sont mappées sur des pages physiques identiques.

Point de rendez-vous: fichiers

Contenu d'un fichier mis dans des pages mémoires, partagées entre processus.

Mémoire partagée



Deux méthodes de partage

mmap(2), munmap(2)

- Contenu d'un fichier est copié dans des pages partagées.
- Le fichier sert de point de rendez-vous
- Le contenu des pages est synchronisé avec contenu du fichier
- Persistant au reboot (c'est un fichier)

shmget(2), shm_open(2), shmat(2), . . .

- Un nom de fichier sert de rendez-vous
- Le contenu du fichier n'est pas utilisé,
- Pas de persistance au reboot

Mémoire partagée et synchronisation

Partage mémoire par un fichier

Synchronise contenu fichier avec pages mémoire

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

Alloue len octets de mémoire virtuelle et les synchronise avec contenu de fichier décrit par fd.

Arguments:

- *addr*: adresse où placer le contenu du fichier. NULL = l'OS choisit (mieux)
- *len*: longueur en octets de la zone mémoire (multiple de taille de page). Le fichier doit faire au moins cette taille ou plus.
- *prot*: autorisation d'accès sur les pages décrite par *ou binaire* entre PROT_NONE (pas d'accès), PROT_READ (lecture), PROT_WRITE (écriture), PROT_EXEC (exécution)

Synchronise contenu fichier avec pages mémoire

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

Alloue len octets de mémoire virtuelle et les synchronise avec contenu de fichier décrit par fd.

Arguments:

- *flags*:
  - MAP_PRIVATE: pages privées, pas de partage
  - MAP_SHARED: pages partagées avec autres processus
  - MAP_ANONYMOUS: pages pas associées à un fichier (fd, offset ignorés)
- *fd* et *offset*: descripteur d'un fichier ouvert et position dans le fichier (multiple d'une taille de page)

Synchronise contenu fichier avec pages mémoire

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

Retourne:

- Adresse mémoire allouée ou `MAP_FAILED`
- Erreurs:
  - `EACCESS`: fd correspond à un fichier non-régulier ou avec des droits d'accès incompatibles (modes O_WRONLY et O_APPEND interdits)
  - `ENOMEM`: pas assez de mémoire

Destruction des pages synchronisées avec un fichier

```
void *munmap(void *start, size_t len);
```

- Désalloue les pages allouées précédemment avec mmap
- Toutes les modifications sur les pages sont synchronisées avec le fichier (`man msync()`)
- Accès ultérieurs aux pages $\rightarrow$ Segfault.

Retourne 0 si succès, -1 si erreur

Partage mémoire par un fichier

Example

```
// Creation
int f = open( "monfichier", O_RDWR );
if ( f == -1 ) {
  perror("open"); exit(1);
}

// Allocation des pages memoires synchronisees avec monfichier
char *A = mmap( NULL, SIZE, PROT_READ|PROT_WRITE,
                MAP_SHARED, f, 0 );
if (A == MAP_FAILED) {
  perror("mmap"); exit(1);
}
A[10] = 'a';

// Destruction
munmap(A, SIZE);
```

Mémoire partagée et synchronisation

Partage mémoire par shm

Création/Modification/Destruction

Création du segment en 3 étapes:

1. Générer une clé à partir d'un nom de fichier: `ftok()`
2. Récupérer l'identifiant de la zone mémoire partagée: `shmget()`
3. Attacher le segment mémoire partagé à l'espace mémoire du processus: `shmat()`

Modification:

- Contrôler le segment mémoire: `shmctl()`

Destruction:

- Détacher le segment mémoire de l'espace mémoire du processus: `shmdt()`
- Détruire le segment mémoire si plus personne ne l'utilise: `shmctl()`

Création: 1-Créer une clé à partir d'un fichier

```
key_t ftok(const char *path, int id);
```

- Crée une nouvelle clé à partir
  - D'un fichier existant *path*
  - D'un entier *id* choisi par utilisateur
- Le fichier sert de point de rendez-vous
- Retourne -1 si erreur (pas de fichier ou pas d'accès)

Création: 2-Créer ou trouver un identificateur de mémoire partagée

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

- *k* est sa clé (cf `ftok()`)
- *size* donne la taille
- *shmflg* donne les flags de création comme les droits d'accès.
- Si la clé est déjà associée à un segment de mémoire
  - Retourne l'identifiant existant
- Sinon, ou si `IPC_CREAT` est dans shmflg, ou si `k=IPC_PRIVATE`
  - Retourne l'identifiant du segment nouvellement créé

Création: 3-Attacher un segment de mémoire partagée

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

Associe le segment de mémoire partagée d'identifiant *shmid* à l'espace d'adressage du processus appelant.

- *shmid*: un identifiant de mémoire (voir `shmget()`)
- *shmaddr*: adresse où devra être placé le segment. Si NULL, l'OS choisit (mieux)
- *shmflg*: 0 ou un *ou binaire* entre plusieurs valeurs possibles dont: SHM_RDONLY: le segment est en lecture seulement

Destruction: Détacher un segment de mémoire partagée

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

- Désassocie le segment de l'espace d'adressage de l'adresse virtuelle `shmaddr`
- Le segment mémoire n'est pas détruit et peut continuer à être accédé par d'autres processus
- `shmaddr` doit avoir été obtenu avec `shmat`

Modification: Contrôle d'un segment mémoire partagé

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

- Effectue une opération sur le segment identifié par *shmid*
- Exemples pour *cmd*
  - `IPC_RMID`: désalloue le segment et détruit les données associées
  - `IPC_STAT`: collecte des informations sur le segment et les place dans buf (`man shmctl` pour la description des champs)
  - `IPC_SET`: change les droits d'accès sur le segment
- Retourne -1 en cas d'erreur.

Résumé

Créer des zones de mémoire partagées entre processus

- Avec synchronisation sur un fichier (mmap)
- Avec un point de rendez vous sur un fichier (shmget)

Pointeurs et mémoire partagée

- Les zones de mémoire partagée commencent à différentes adresses pour chaque processus (mémoire virtuelle)
- Ne pas stocker des pointeurs en mémoire partagée!

Partage mémoire par shm

Example

```
// Create the key
key_t key = ftok("/etc/bash.bashrc", 1);
if ( key == -1 ) { perror("ftok"); exit(1); }

// Get the shmid
id = shmget( key, SIZE, IPC_CREAT|0644 );
if ( id == -1 ) { perror("shmget"); exit(1); }

// Attach
A = shmat( id, NULL, 0 );
if ( A == (void*)-1 ) { perror("shmat"); exit(1); }

// Detach/destroy
shmdt(A);
shctl( id, IPC_RMID, NULL);
```

Threads

Threads

Définition des threads

Threads

Définition: une exécution séquentielle dans un processus (aussi appelé processus léger)

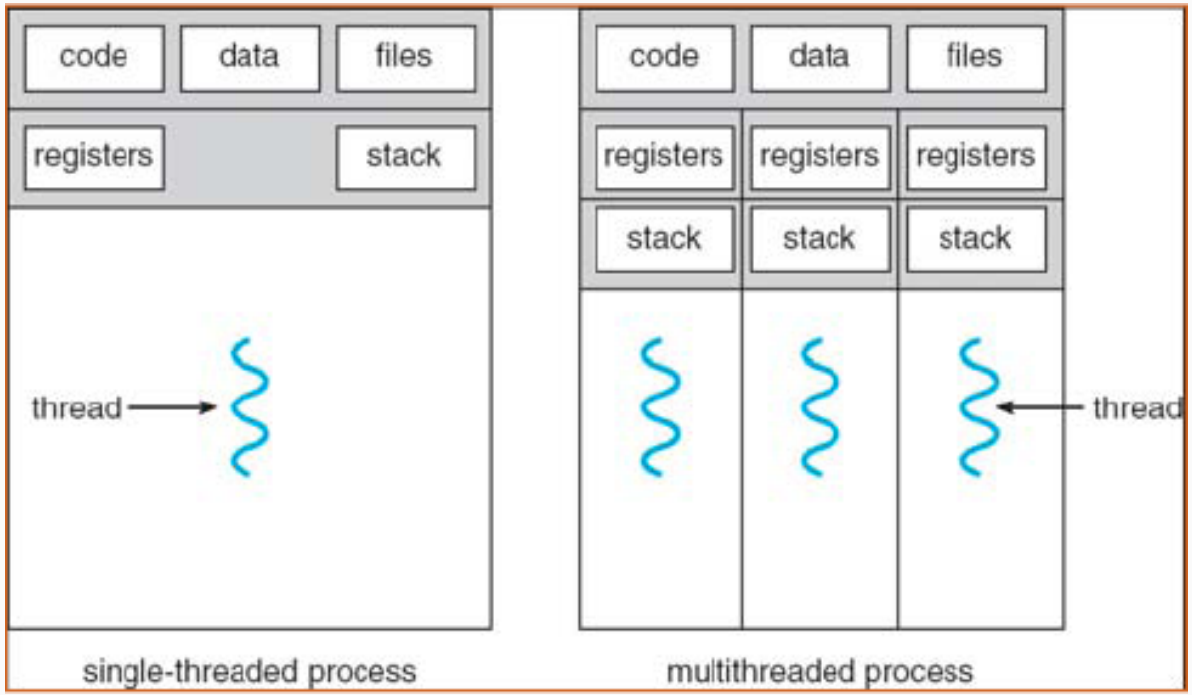
Exécution multithreads:

- Dans un processus unique, multiples exécutions séquentielles
- Tous les threads partagent le même espace d'adressage (virtuel) du processus
- !! Pas de protection ou séparation mémoire entre threads!!

Ce qui est partagé ou non

Partagé entre les threads: Code, données, descripteurs de fichier

Privé à chaque thread: Valeurs des registres, pile (var. locales)



Ce qui est partagé ou non

Données communes entre les threads

- Contenu de la mémoire (programme, tas, état des entrées/sorties)

Données privées

- Données de la pile
- Valeurs des registres
- Informations sur l'ordonnancement du thread
- Gardées dans le Thread Control Block (TCB)

Deux composantes d'un processus:

- Mémoire virtuelle (protection)
- Threads (pour exécution parallèle)

Exemples de programmes multi-threadés

Systèmes embarqués

- Avions, ascenceurs, systèmes médicaux, systèmes audio/vidéo, ...
- Un seul programme, opérations concurrentes

Systèmes d'exploitations modernes

- Opérations parallèles entre les utilisateurs
- Pas de protection nécessaire dans le noyau

Serveurs de base de données

- Accès concurrents (en parallèles) à la base
- Activités en background de maintenance de la base

Serveurs réseau

- Multiples requêtes du réseau
- Serveur de fichier, serveur web, réservation de billets,...

Exemples de programmes multi-threadés

Programmes parallèles (plusieurs processeurs/coeurs)

- Expression du parallélisme pour des coeurs qui partagent la mémoire
- Pour du calcul hautes performances, on a parfois 1 coeur = 1 thread.
- Threads facilitent programmation/communication entre coeurs
- Langages pour programmation multithread: OpenMP par exemple

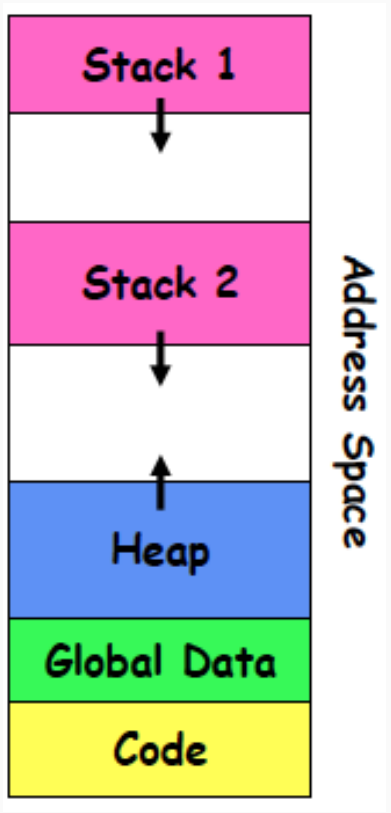
Threads VS processus

Création de threads

- Moins couteuse que celle de processus
- Pas de pages à copier
- Allocation de piles séparées

Exécution:

- Ordonnancement des threads par le système d'exploitation
- Attention: exécution parallèle, ne pas faire d'hypothèse sur l'ordre d'exécution !!



Threads

Interface de programmation POSIX

Création de threads

Fonction posix

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

- Un peu plus compliqué que fork !
- Sous linux, voir fonction système clone.
- Compiler avec -lpthread (ou -lthread)
- Crée un thread après l'appel

Création de threads

Fonction posix

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

- *thread*: identifiant du thread qui est créé
- *attr*: correspond aux attributs du thread (détaché, joignable, ou NULL (défaut)).
  Voir `pthread_detach(3)`, `pthread_attr_init(3)`
- *start_routine*: fonction exécutée par le nouveau thread
- *arg*: paramètre passé à la fonction *start_routine*

Terminaison d'un thread

```
void pthread_exit(void *retval);
```

- Termine l'exécution du thread
- La valeur retval (valeur arbitraire) est la valeur de retour du thread
- Appelée implicitement si le thread termine normalement
- Ne rend jamais la main
- Ne libère pas les ressources du thread ! Voir `pthread_join()` ou `pthread_detach()`
- Attention aux fuites mémoire

Attendre la fin d'exécution d'un thread

```
int pthread_join(pthread_t tid, void **return);
```

- Suspend l'exécution du thread appelant jusqu'à ce que le thread *tid* termine.
- **return* est égal à la valeur de *retval* donnée par le thread *tid* lors de l'appel à `pthread_exit()`
- **return* vaut `PTHREAD_CANCELED` si le thread *tid* a été annulé
- Le thread *tid* ne doit pas avoir été détaché
- Libère les ressources du thread ayant terminé

Signaux globaux au processus

Utiliser fonctions spécifiques pour les threads:

Processus		Threads
`kill()`	→	`pthread_kill()`
`sigprocmask()`	→	`pthread_sigmask()`
`sigsuspend()`	→	`sigwait()`

Exemple

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

/* Function run by the threads.
 * Threads stop when they exit
 * the function.
 */
void *start(void *ptr) {
  int x = *(int *)ptr;
  printf("[pid=%d] New thread=%d\n",
         getpid(), x);
  // Equal to pthread_exit(NULL)
  return NULL;
}
```

```c
int main() {
  pthread_t pid[POOL];
  int i;
  int x=3;
  void *ptr = (void *)&x;
  /* Create the threads */
  for (i=0; i<POOL; i++) {
    pthread_create(&pid[i], NULL,
                   start, ptr);
  }
  printf("Main thread is running\n");
  /* Wait for other threads */
  for (i=0; i<POOL; i++) {
    pthread_join(pid[i], NULL);
  }
  return EXIT_SUCCESS;
}
```

Concurrence

Concurrence

Programmes concurrents et mémoire partagée
Source de maux de têtes et bugs difficiles à reproduire.

Valable pour les processus et mémoire partagée (shmem, mmap) ou threads.

Concurrence

Identification des problèmes

Concurrence et mémoire partagée

La plupart du temps, les threads travaillent sur des données différentes, l'ordonnancement n'est pas important:

Thread 1:
x = 1;

Thread 2:
y = 2;

Concurrence et mémoire partagée

Que se passe-t-il avec le code suivant si initialement y vaut 4?

Thread 1:

```
x = 1 ;
x = y-1;
```

Thread 2:

```
y = 2;
y = y*2;
```

Concurrence et mémoire partagée

Quelles sont les valeurs possibles de x ? (initialement x=1)

Thread 1:

```
x = x+2;
```

Thread 2:

```
x = x *3;
```

Concurrence et mémoire partagée

Quelles sont les valeurs possibles de x ? (initialement x=1)

Thread 1:

```
x = x+2;
```

Thread 2:

```
x = x *3;
```

$\rightarrow$ Non déterministe!

Opérations atomiques

- Séquence d'opérations qui s'exécute complètement ou pas du tout
- Séquence d'opérations indivisible
- Ne sont pas interrompues en plein milieu par l'exécution d'un autre thread
- Opérations atomiques usuelles:
  - un accès mémoire (pour la longueur native des variables)
  - une opération de calcul (correspond à une instruction asm)
- Exemple d'opérations non atomiques:
  - Certains opérateurs de calculs (divisions par exemple, prend plusieurs instructions asm)
  - Accès à une variable 64 bits sur machine 32 bits

Autre exemple de problème

On peut supposer opérations (incrément, comparaison) atomiques.
Programme non déterministe!:

- Les deux peuvent gagner
- Les deux peuvent ne pas terminer ...

Thread 1:

```
while (x<1000) x++;
printf("le thread 1 gagne!");
```

Thread 2:

```
while (x>-1000) x--;
printf("le thread 2 gagne!");
```

Concurrence

Solution logicielles

Analogie vie courante

- Rend les choses plus faciles à comprendre
- Mais ordinateurs moins intelligents que les personnes

Exemple de coordination:

Thread 1	Thread 2
Vérifie frigo	
Plus de lait!	
Va au supermarché	Vérifie frigo
Achète du lait	Plus de lait!
Revient	Va au supermarché
	Achète du lait
	Revient

Analogie vie courante

- Rend les choses plus faciles à comprendre
- Mais ordinateurs moins intelligents que les personnes

Exemple de coordination:

Thread 1	Thread 2
Vérifie frigo	
Plus de lait!	
Va au supermarché	Vérifie frigo
Achète du lait	Plus de lait!
Revient	Va au supermarché
	Achète du lait
	Revient

Résultat: trop de lait !

Définitions

Synchronisation
Utiliser des opérations atomiques pour coordonner les threads.

Exclusion mutuelle

- Un seul thread fait une action particulière à un moment donné
- Un thread exclut les autres de faire cette action en même temps

Section critique
Zone de code qu'un seul thread peut exécuter à la fois.

Verrou
Mécanisme qui empêche un thread de faire une action:

- Attend si verrouillé
- Si pas verrouillé, verrouille en entrant (section critique), déverrouille en partant.

Solution 1

Verrouille le frigo en partant.

Solution 1

Verrouille le frigo en partant.

$\rightarrow$ Très contraignant. Empêche l'utilisation du frigo pour prendre un jus d'orange!

Comment implémenter un verrou en mémoire ?

Solution 2

- Laisser un post-it sur le frigo en partant
- L'enlever en revenant
- Si post-it, ne pas acheter du lait (attendre)

```
if (!lait) {
  if (!postit) {
     postit=true;
     lait++;
     postit=false;
  }
}
```

Solution 2

- Laisser un post-it sur le frigo en partant
- L'enlever en revenant
- Si post-it, ne pas acheter du lait (attendre)

```
if (!lait) {
  if (!postit) {
     postit=true;
     lait++;
     postit=false;
  }
}
```

Résultat: Toujours trop de lait!

Solution 3

Utiliser des post-its différents par thread

- Mettre son post-it
- Si pas de post-it de l'autre, et si plus de lait, en chercher
- Enlever son post-it
- Si post-it de l'autre, attendre.

```
//Thread 1
postit_1=true;
if (!postit_2) {
  // Sec. crit
  if (!lait) lait++;
}
postit_1=false;

//Thread 2:
postit_2=true;
if (!postit_1) {
  // Sec. crit
  if (!lait) lait++;
}
postit_2=false;
```

Solution 3

Utiliser des post-its différents par thread

- Mettre son post-it
- Si pas de post-it de l'autre, et si plus de lait, en chercher
- Enlever son post-it
- Si post-it de l'autre, attendre.

Résultat: Possible que personne n'achète du lait!
Ce type de deadlock = famine

```
//Thread 1
postit_1=true;
if (!postit_2) {
  // Sec. crit
  if (!lait) lait++;
}
postit_1=false;

//Thread 2:
postit_2=true;
if (!postit_1) {
  // Sec. crit
  if (!lait) lait++;
}
postit_2=false;
```

Solution 4

Algorithme de Peterson

- Mettre notre post-it
- Écrire que c'est le tour de l'autre
- Tant qu'il y a l'autre post-it et que c'est le tour de l'autre, attendre
- Si pas de lait, en prendre
- Enlever son post-it

```
//Thread 1
postit_1 = true; tour=2;
while (postit_2 && tour==2);
if (!lait) lait++;// Sec. crit.
postit_1 = false;

//Thread 2
postit_2 = true; tour=1;
while (postit_1 && tour==1) ;
if (!lait) lait++; // Sec. crit.
postit_2 = false;
```

Solution 4

Algorithme de Peterson

- Mettre notre post-it
- Écrire que c'est le tour de l'autre
- Tant qu'il y a l'autre post-it et que c'est le tour de l'autre, attendre
- Si pas de lait, en prendre
- Enlever son post-it

Résultat: correct

```
//Thread 1
postit_1 = true; tour=2;
while (postit_2 && tour==2);
if (!lait) lait++;// Sec. crit.
postit_1 = false;

//Thread 2
postit_2 = true; tour=1;
while (postit_1 && tour==1) ;
if (!lait) lait++; // Sec. crit.
postit_2 = false;
```

Solution 5

Algorithme de Decker

- Mettre son post-it
- Tant que post-it de l'autre:
- si pas mon tour:
  - enlève post-it
  - tant que c'est pas mon tour
  - met mon post-it
- Si plus de lait, en acheter
- C'est le tour de l'autre
- Enlève post-it

```
//Thread 1
postit_1=true;
while (postit_2) {
    if (turn!=1) {
        postit_1 = false;
        while (turn != 1);
        postit_1 = true;
    }
}
if(!lait) lait++; // Sec. crit.
turn = 2;
postit_1=false;
```

Solution 5

Algorithme de Decker

- Mettre son post-it
- Tant que post-it de l'autre:
- si pas mon tour:
  - enlève post-it
  - tant que c'est pas mon tour
  - met mon post-it
- Si plus de lait, en acheter
- C'est le tour de l'autre
- Enlève post-it

Résultat: correct

```
//Thread 1
postit_1=true;
while (postit_2) {
    if (turn!=1) {
        postit_1 = false;
        while (turn != 1);
        postit_1 = true;
    }
}
if(!lait) lait++; // Sec. crit.
turn = 2;
postit_1=false;
```

Solutions logicielles: bilan

- Algorithme de Peterson
  - Plus simple que celui de Dekker
  - Peut être étendu à plus de 2 threads
- Bilan solution logicielle:
  - Solution relativement compliquée juste pour 2 threads
  - Suppose accès mémoire atomiques
  - Suppose accès mémoire faits dans l'ordre séquentiel pour chaque thread
    - Peut être changé par compilateur si dépendances respectées
    - Peut être changé par le matériel (Out of order execution)
  - Fait de l'attente active (while (p);). Très coûteux en temps CPU !

Concurrence

Solution matérielles

Solutions *matérielles* pour la synchronisation

- Verrous (pthread)
- Sémaphores (POSIX)
- Moniteurs et variables conditionnelles (pthread, Java)

Solution 1 au problème du lait

Si on peut mettre un verrou uniquement pour le lait, solution facile:

- *Verrouille()*: attend tant que verrou mis, ensuite met le verrou
- *Deverrouille():* enlève le verrou et réveille ceux qui attendent

Protège accès section critique (1 thread à la fois)

```
//Thread 1 et 2:
verrou_lait.verrouille();
// Section Critique
if (!lait) lait++;
verrou_lait.deverrouille();
```

Verrouille et déverrouille doivent être atomiques

Séquence atomique d'instructions

- Test & set:
  - Met le verrou et vérifie s'il y était déjà
- Sur la plupart des architectures:
  - Une instruction *asm*
- Variantes:
  - Permute deux valeurs dont une est en mémoire (x86)
  - Compare et swap (68xxx)

```
int verrou=0;
test&set() { // Atomique
   resultat = verrou;
   verrou = 1;
   return resultat;
}
```

Utilisation du verrou

- Si verrou pas mis, *test&set* lit 0 et le met. Le while quitte.
- Si verrou mis, *test&set* lit 1 et le met à 1 (ne change rien). Le while boucle.

Attente active: consomme des cycles en attendant

```
int verrou=0;
test&set(p) { // Atomique
   resultat = *p;
   *p = 1;
   return resultat;
}
verrouille() {
  // Attente active
  while (test&set(&verrou));
}
deverouille() {
  verrou = 0;
}
```

Attente active

- Consomme des cycles en attendant
- Ralentit autres threads/processus qui ne sont pas en attente
- Inversion des priorités: si thread en attente active a priorité plus haute que le thread ayant le verrou $\rightarrow$ rien n'avance!

- Possible de faire un verrou sans attente ? Non
- Possible de faire un verrou sans attente active ? En partie

Attente active limitée

```
verrouille() {
  // Courte attente active
  while (test&set(&porte));
  if (verrou == OCCUPE) {
    PutThrdInWaitingList;
    porte = 0 & sleep();
  } else {
    verrou = OCCUPE;
    porte = 0;
  }
}
```

```
deverrouille() {
  if (un thread dors) {
    RemoveThrdFromWaitingList;
    PutThrdInActiveList;
  }
  else {
    verrou = LIBRE;
  }
}
```

- Idée: Faire de l'attente active uniquement pour vérifier atomiquement la valeur de lock

Programmation des verrous

- Verrouiller: `pthread_mutex_lock()`
- Déverrouiller: `pthread_mutex_unlock()`

Nécessite la création d'un mutex (verrou pour exclusion mutuelle):

```
int pthread_mutex_init(pthread_mutex_t *m,
pthread_mutexattr_t *attr);
```

Création d'un nouveau mutex. Si attr est NULL, attributs par défaut pour le mutex.

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

Libère les ressources allouées pour le mutex

C'est de l'attente passive, voir `int pthread_spin_lock()` pour l'attente active.

Programmation des verrous

Verrouiller un mutex:

```
int pthread_mutex_lock(pthread_mutex_t *m)
```

Si le mutex est déjà verrouillé, bloque le thread (l'endort).

Renvoie -1 si erreur:

- EDEADLK: un deadlock va arriver si le mutex est bloqué à cause de ce mutex

Déverrouiller:

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

Sémaphores

- Sémaphores généralisent les verrous
- Inventés par Dijkstra à la fin des années 1960
- Principale primitive de synchronisation à l'origine dans UNIX
- Définition: un sémaphore est un entier non négatif qui a deux opérations
  - `P()` : opération atomique qui attend que le sémaphore soit positif et le décrémente de 1.
    Proberen (tester en néerlandais) = puis-je ?
  - `V()` : opération atomique qui incrémente le sémaphore de 1, et réveille les threads/processus bloquant sur P(). *Verhogen* (incrémente en néerlandais) = vas-y

Sémaphores

- Comme des entiers $>= 0$
- Seules opérations:
  - initialisation lors de la création
  - P(), atomique
  - V(), atomique

Valeur initiale = 1

$\rightarrow$ Sémaphore = verrou

```
semaphore.P();
// section critique
semaphore.V();
```

Sémaphores

- Comme des entiers $>= 0$
- Seules opérations:
  - initialisation lors de la création
  - P(), atomique
  - V(), atomique

Valeur initiale = 0

$\rightarrow$ Permet d'attendre un autre thread.

Exemple d'attente d'un thread devant lors de la terminaison d'un autre thread:

```
Threadjoin() {
  semaphore.P();
}
Threadexit() {
  semaphore.V();
}
```

Sémaphores

Valeur initiale $k > 1$

$\rightarrow$ Utilisé pour limiter accès à une ressource par k threads au plus.

```
semaphore.P();
// k threads ici au plus en meme temps
semaphore.V();
```

Utile si ressource en k exemplaires seulement (processeur, périphériques, ...).

Programmation des sémaphores (Nommées)

Création d'un sémaphore

```
sem_t *sem_open(char *name, int flags);
```

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

Arguments:

- Seulement quelques valeurs possibles de flags parmi celles du open: O_CREAT et O_EXCL pour flags
- name: nom utilisé pour que d'autres processus utilisent le même sémaphore
- *value*: valeur d'initialisation du sémaphore. Par défaut, 1.

Retourne -1 si une erreur.

Désalloue les ressources du sémaphore:

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

Programmation des sémaphores (Anonymes)

Création d'un sémaphore `int sem_init(sem_t *sem, int pshared, unsigned int value);`

Arguments:

- *pshared*: Spécifie le niveau de partage de la sémaphore (threads ou processus)
- *value*: valeur d'initialisation du sémaphore.

Désalloue les ressources du sémaphore:

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

Programmation des sémaphores

Incrémente un sémaphore, V()

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

Débloque le sémaphore et l'incrémente de 1.

Décrémente un sémaphore, P() et bloque processus si nécessaire

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

Bloque si la valeur est $<= 0$. Le processus est débloqué si la valeur > 0. Retourne -1 si erreur.

Exemple producteur/consommateurs

Définition du problème:

- Un producteur écrit des données dans un buffer partagé (taille limitée)
- Un consommateur les enlève du buffer
- Besoin d'une synchronisation entre consommateur et producteur
  - Le producteur doit attendre si buffer plein
  - Le consommateur doit attendre si buffer vide
- Buffer = ressource partagée modifiée.
  $\rightarrow$ Besoin section critique

Example

- séquence de pipe, gcc (cpp | cc1 | cc2 | as | ld)
- filtres sur des images

Un mot sur les variables de condition

Comment (atomiquement) tester une condition et dormir si elle est fausse?

- Si la condition est un entier qui doit être positif, un sémaphore suffit.
- Les variables de condition peuvent servir aux autres cas:
  - Prendre un verrou.
  - Tester si la condition est vraie.
  - Si oui, continuer en tenant le verrou.
  - Sinon dormir en relachant atomiquement le verrou et être reveillé plus tard en reprenant atomiquement le verrou.

Avec les pthreads

Pour attendre qu'une condition devienne vraie:

```
pthread_mutex_lock(&mutex);
/* attente passive que <condition> soit vraie */
while (!<condition>)
  pthread_cond_wait(&cond, &mutex);
/* <condition> est vraie */
...
pthread_mutex_unlock(&mutex);
```

Le thread qui influe sur `<condition>` doit réveiller ceux qui attendent ci-dessus avec avec `pthread_cond_signal()` ou `pthread_cond_broadcast()`.

Concurrence

Deadlocks

Deadlocks

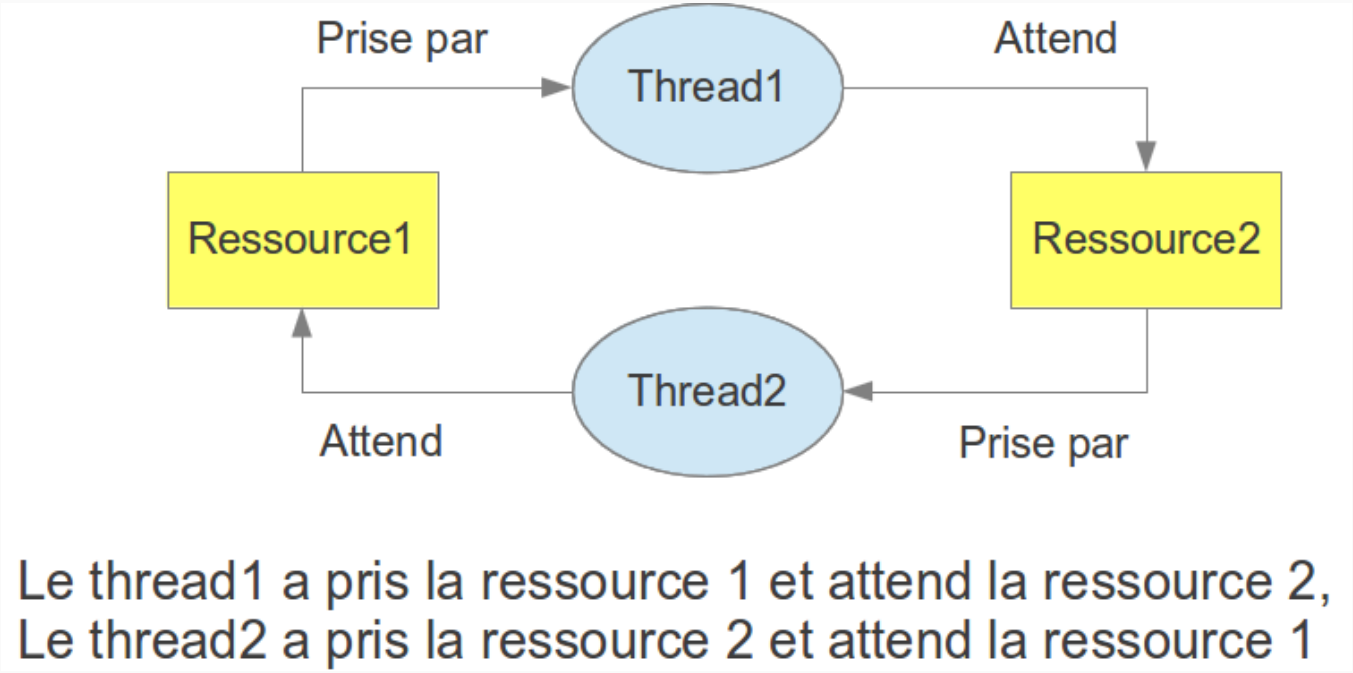
Ressources: entités nécessaires aux threads pour leur exécution

- Processeur, temps, espace disque, mémoire, périphérique
- Certaines ressources sont partageables entre threads, d'autres non (nécessite section critique):
  - Imprimante non partageable pendant l'impression
  - Fichiers en lecture seule sont partageables
- Ressources préemptables: on peut les retirer aux threads

Un but essentiel des systèmes d'exploitation est de gérer les ressources

Deadlocks

Deadlocks: situations d'inter-blocage dû au partage de ressources entre threads/processus.



Conditions pour un deadlock

- Dépend de l'exécution, pas déterministe

Thread 1	Thread 2
x.P();	y.P();
y.P();	x.P();
// deadlock free	// deadlock free
y.V();	x.V();
x.V();	y.V();

  Un deadlock ne se produit pas toujours pour ce code.
- Doit impliquer de multiples ressources
  - Impossible de débugger pour chaque ressource indépendamment
- Pas de débloquage sans intervention extérieure

Quelques exemples

- Carrefour avec priorité à droite
  - Chaque voiture possède un segment de route et veut acquérir le segment suivant
  - Pour traverser, doit avoir deux segments
- Résolution du deadlock
  - Une voiture fait marche arrière (rollback). Plusieurs voitures devront reculer éventuellement.
  - Une voiture est enlevée (préemption). Problème si la voiture enlevée a une autre ressource.
- Famine possible: impossibilité de traverser si beaucoup de trafic dans une direction.

Dîner des philosophes (Djikstra, Hoare)

- 5 fourchettes pour 5 personnes
- Besoin de 2 fourchettes pour manger
- Philosophes prennent celle de droite puis de gauche

$\rightarrow$ Deadlock possible

Prévention:

- Un des philosophes repose une fourchette
- Ne jamais prendre la dernière fourchette si aucun philosophe affamé n'a 2 fourchettes après

Famine possible.

4 Conditions pour un deadlock

1. Exclusion mutuelle
2. Les threads/processus attendent en possédant des ressources
3. Pas de préemption
   - Les ressources ne sont relâchées que volontairement par un thread/processus
4. Attente circulaire
   - Il y a un ensemble T1..Tn de threads en attente, avec
     T1 attend une ressource prise par T2
     T2 attend une ressource prise par T3
     ...
     Tn attend une ressource prise par T1

Empêcher les deadlocks

- Éliminer les symétries ou attentes circulaires
  - Pas toujours possible ni souhaitable
  - Force un ordre pour prendre les ressources
- Utiliser préemption
  - Risque d'état incohérent (tue un processus par ex.)
- Détecte situation potentielle de deadlock
  - Construit graphe d'utilisation des ressources
  - Détecte les cycles (algorithme d'allocation du banquier)
- Imposer des timeouts
  - Pas de garantie en soit que deadlock ne se reproduira pas
- Détection (par autre thread) de deadlock
  - Rollback possible.

Deadlocks: conclusion

- Problèmes de concurrence
  - Non déterminisme des calculs
  - Inter-blocages, famines
- Solutions
  - Tout accès en écriture d'une ressource partagée
    $\rightarrow$ dans section critique
  - Exclusion mutuelle logicielle possible, si accès mémoire atomique
  - Solution matérielle préférable (verrous, sémaphores)
- Deadlocks
  - Bug difficile à reproduire
  - Nécessite 4 conditions pour se produire

Réseau - sockets

Modes de connexion

Il existe 2 modes de connexion:

- Connecté: la connexion est établie une seule fois entre les deux hôtes, puis les données sont échangées
  - Similaire téléphone
  - Type `STREAM` de socket. Fonctionne comme une FIFO, fiable (détection erreurs, ré-envoi)
- Non connecté: la socket sert à l'envoi/la réception de messages
  - Similaire télégrammes
  - Type `DGRAM` de socket. Non fiable (duplication, perte)

Création d'une socket

- Création d'une socket

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

- Domaine:
  - AF_UNIX: local (127.0.0.1), AF_INET: internet Ipv4, ...
- Type:
  - SOCK_STREAM: mode connecté
  - SOCK_DGRAM: mode non connecté
  - SOCK_RAW: accès direct aux données du réseau
- Protocole:
  - 0: le système choisit le protocole
  - IPPROTO_UDP: protocole UDP (pour SOCK_DGRAM)
  - IPPROTO_TCP: protocole TCP (pour SOCK_STREAM)

Domaine d'une socket

- Local: une entrée est créée pour la socket dans système de fichier
- Internet: besoin de définir une adresse + port
  - Sous unix, $port < 1024 \Rightarrow$ pour l'OS
  - Exemple:
    - ftp $\rightarrow$ 21/tcp
    - http $\rightarrow$ 80/tcp
    - telnet $\rightarrow$ 23/tcp
  - Liste des services de l'OS: dans /etc/services

Lier addresse+port à une socket

Attacher adresse+port à une socket

```
int bind(int sock, struct sockaddr *addr, socklen_t
size);
```

Associe l'adresse définie dans `addr` à la socket `sock`

Format générique de `struct sockaddr`:

```
/* /usr/include/.../bits/socket.h */
/* Structure describing a generic socket address.  */
struct sockaddr
  {
    /* Common data: address family and length.  */
    __SOCKADDR_COMMON (sa_);
    /* Address data.  */
    char sa_data[14];
  };
```

struct sockaddr (Unix)

Format Unix:

```
/* /usr/include/.../sys/un.h */
/* Structure describing the address of an AF_LOCAL
   (aka AF_UNIX) socket. */
struct sockaddr_un
  {
    __SOCKADDR_COMMON (sun_);
    /* Path name. */
    char sun_path[108];
  };
```

struct sockaddr (Internet)

Format Internet:

```
/* /usr/include/netinet/in.h */
/* Structure describing an Internet socket address.  */
struct sockaddr_in
  {
    __SOCKADDR_COMMON (sin_);
    /* Port number.  */
    in_port_t sin_port;
    /* Internet address.  */
    struct in_addr sin_addr;
    /* Pad to size of struct sockaddr.  */
    unsigned char sin_zero[sizeof (struct sockaddr) -
                           __SOCKADDR_COMMON_SIZE -
                           sizeof (in_port_t) -
                           sizeof (struct in_addr)];
  };
```

Adresse Internet

Fabriquer une socket avec l'adresse de l'hôte local

```
struct sockaddr_in s;
s.sin_family = AF_INET;
s.sin_port = 7000; // un numero de port > 1024
// une adresse ip de la machine courante
s.sin_addr.s_addr = INADDR_ANY
```

Fabriquer une socket avec une adresse distante

```
struct sockaddr_in s; struct hostent *h;
s.sin_family = AF_INET;
s.sin_port= 7000;
// obtient une IP a partir d'un DNS
h = gethostbyname("www.google.com");
memcpy(s.sin_addr , h.h_addr_list[0], h.h_length);
```

Réseau - sockets

Mode Connecté (Serveur)

Mode Connecté (Serveur)

Attendre connexions sur une socket:

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

Déclare un service à l'OS:

- `sock`: descripteur de socket créé par socket
- `backlog`: nombre max de demandes de connexions mises en attente (longueur max file d'attente)
- Socket devient passive, utilisée pour attendre les connexions.

Mode Connecté (Serveur)

Accepter une connexion sur une socket

```
int accept(int sock, struct sockaddr *a, socklen_t
*len);
```

- `sock`: descripteur de la socket
- `a`: rempli par l'appel. Adresse de l'hôte qui se connecte
- `len`: initialement, longueur de la structure a. Puis rempli par l'appel à la réelle longueur de a.

L'appel est bloquant:

- Exécution stoppée tant que pas de connexion
- Retourne le descripteur $>$ 0 de la socket crée, -1 si erreur.

Mode Connecté (Client)

Demander une connexion à une socket distante

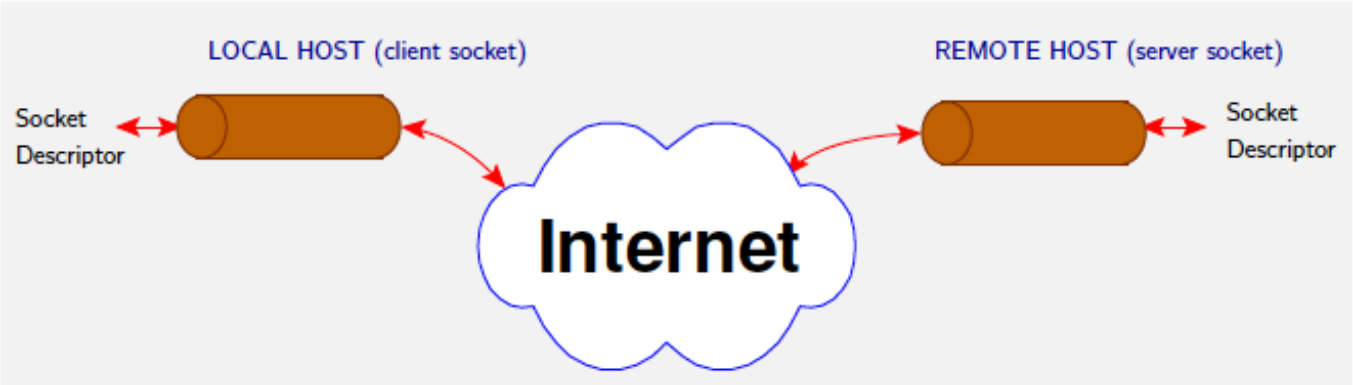
```
int connect(int sock, struct sockaddr *server,
socklen_t len);
```

- `sock`: descripteur de socket
- `server`: adresse du serveur auquel on se connecte
- `len`: longueur de la structure server

L'appel est bloquant.

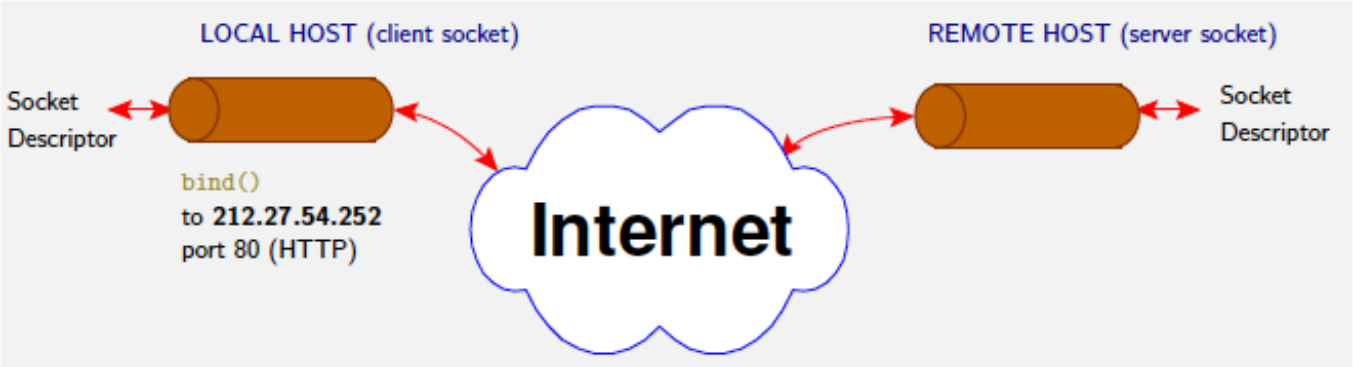
Mode Connecté

ATTENTION: sur la figure, client et serveur sont inversés



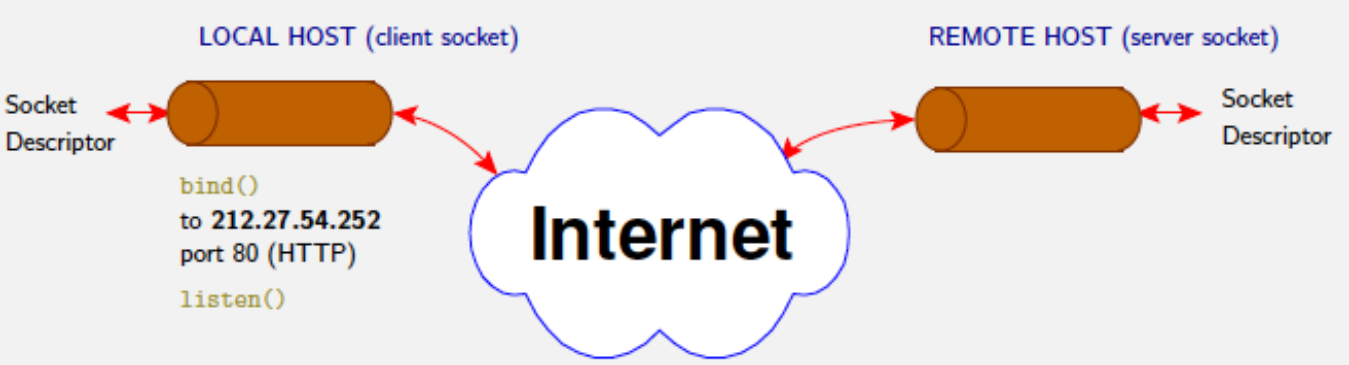
Mode Connecté

ATTENTION: sur la figure, client et serveur sont inversés



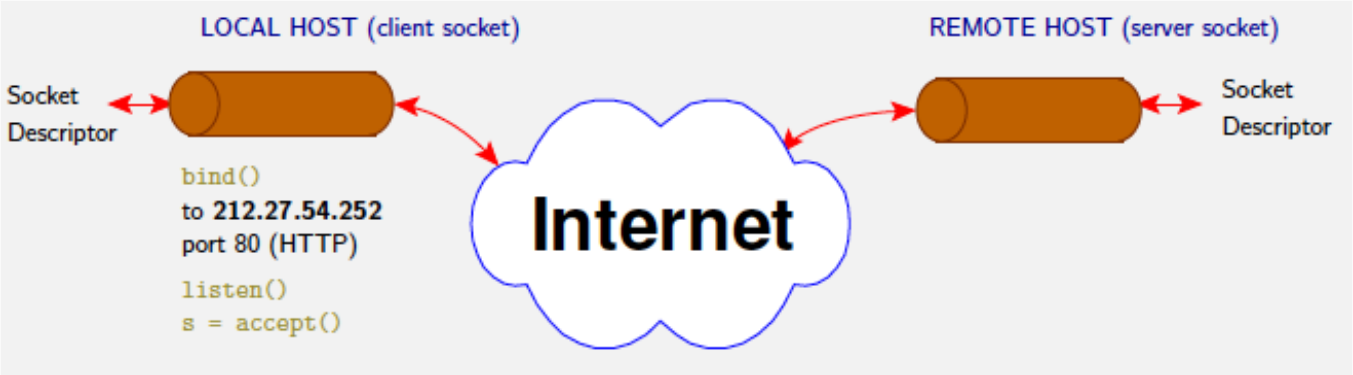
Mode Connecté

ATTENTION: sur la figure, client et serveur sont inversés



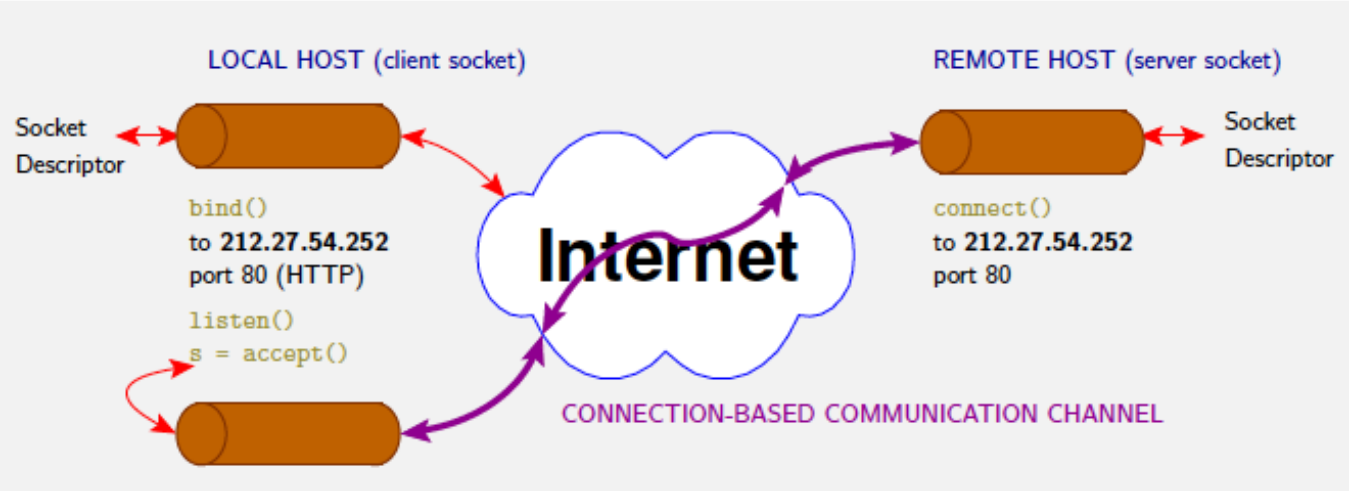
Mode Connecté

ATTENTION: sur la figure, client et serveur sont inversés



Mode Connecté

ATTENTION: sur la figure, client et serveur sont inversés

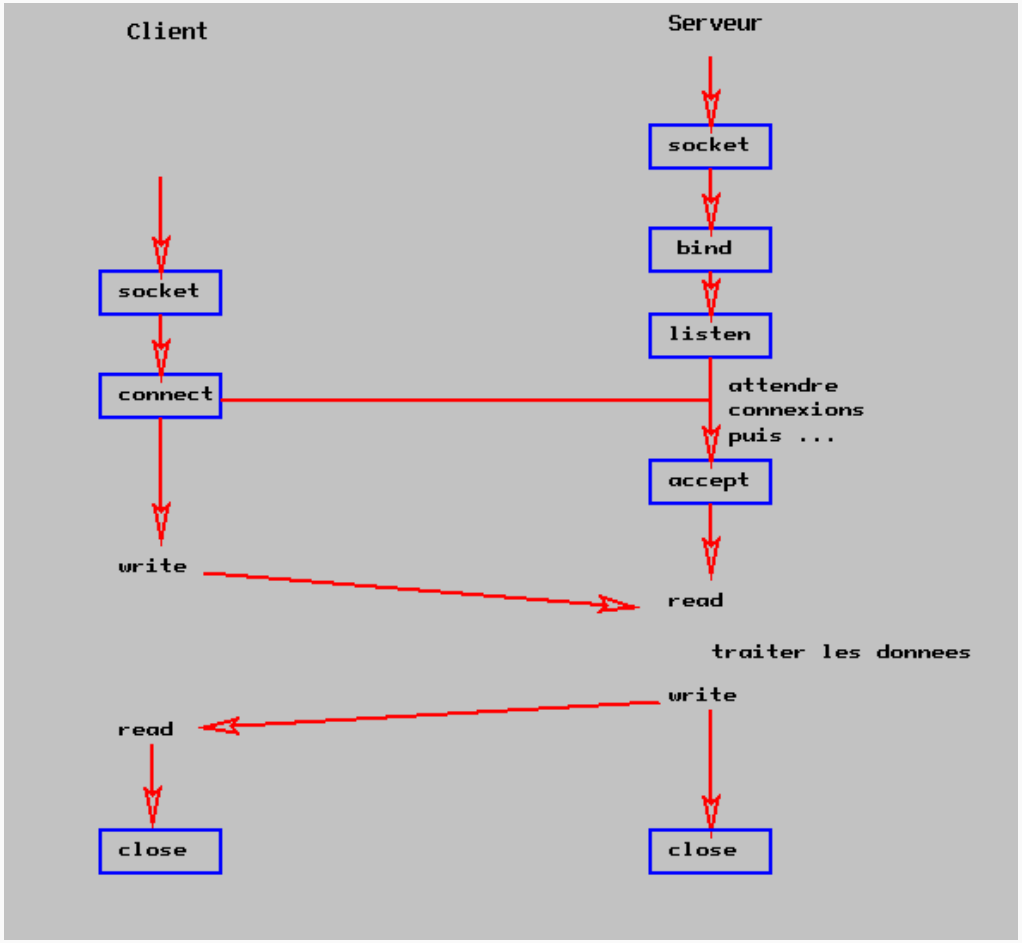


Mode Connecté

Avec la socket retournée par accept:

- Écritures:
  - Avec `write(int sock, void *buffer, size_t size);`
  - Avec `send(int sock, void *msg, size_t len, int flags);`
- Lectures:
  - Avec `read(int sock, void *buffer, size_t size);`
  - Avec `recv(int sock, void *msg, size_t len, int flags);`
- Fermeture:
  - `int close(int sock)`

Schéma Mode Connecté



Réseau - sockets

Mode non connecté

Mode Non Connecté

Envoi de message

```
ssize_t sendto(int socket, void *buffer, size_t
length, int flags, struct sockaddr *dest_addr,
socklen_t len);
```

Combine les effets d'un connect et d'un send du mode connecté.

Retourne le nombre d'octets envoyés (ou -1 si erreur)

Mode Non Connecté

Réception de message

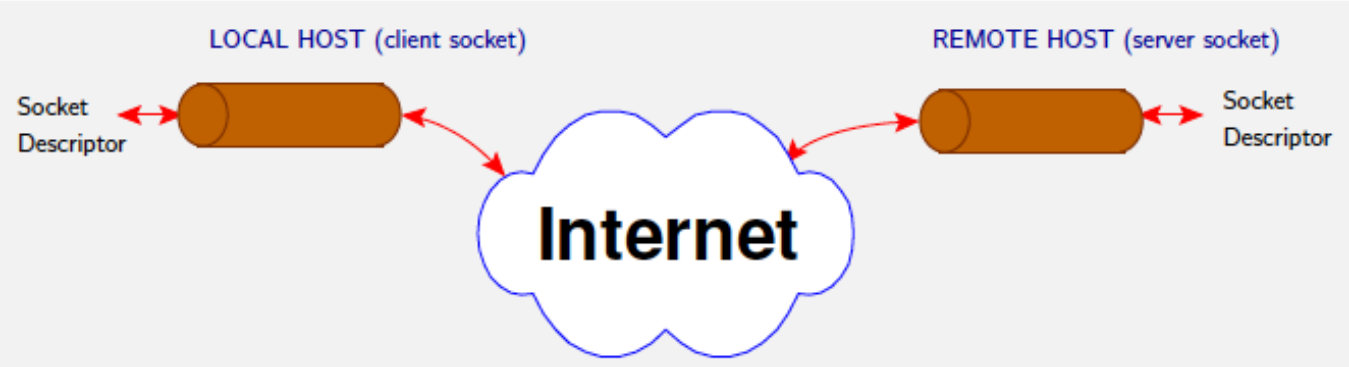
```
ssize_t recvfrom(int socket, void *buffer, size_t
length, int flags, struct sockaddr *dest_addr,
socklen_t *len);
```

Combine l'effet d'un accept et d'un recv du mode connecté.

Retourne le nombre d'octets reçus (ou -1 si erreur)

Mode Non Connecté

ATTENTION: sur la figure, client et serveur sont inversés



Mode Non Connecté

ATTENTION: sur la figure, client et serveur sont inversés

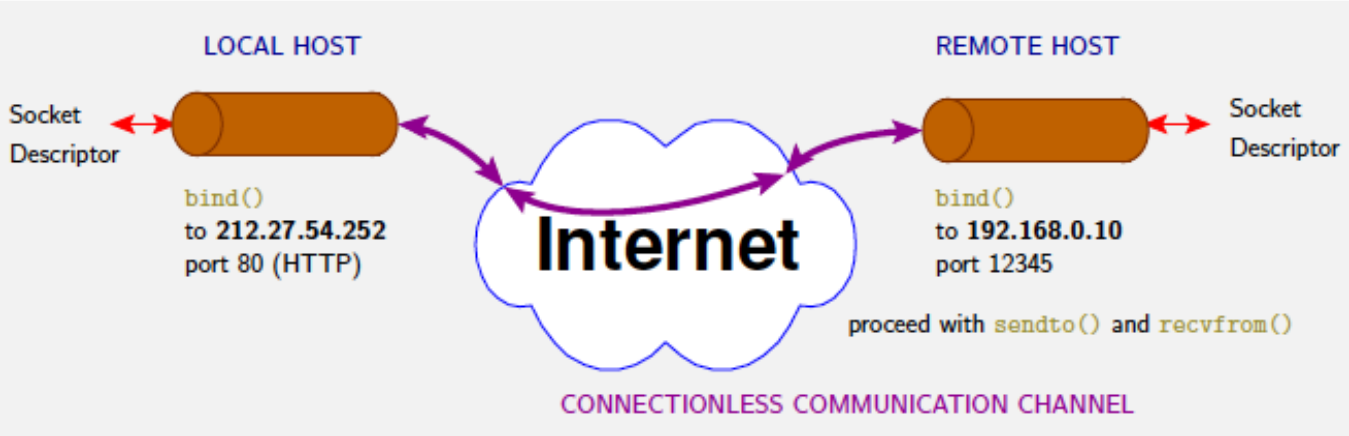
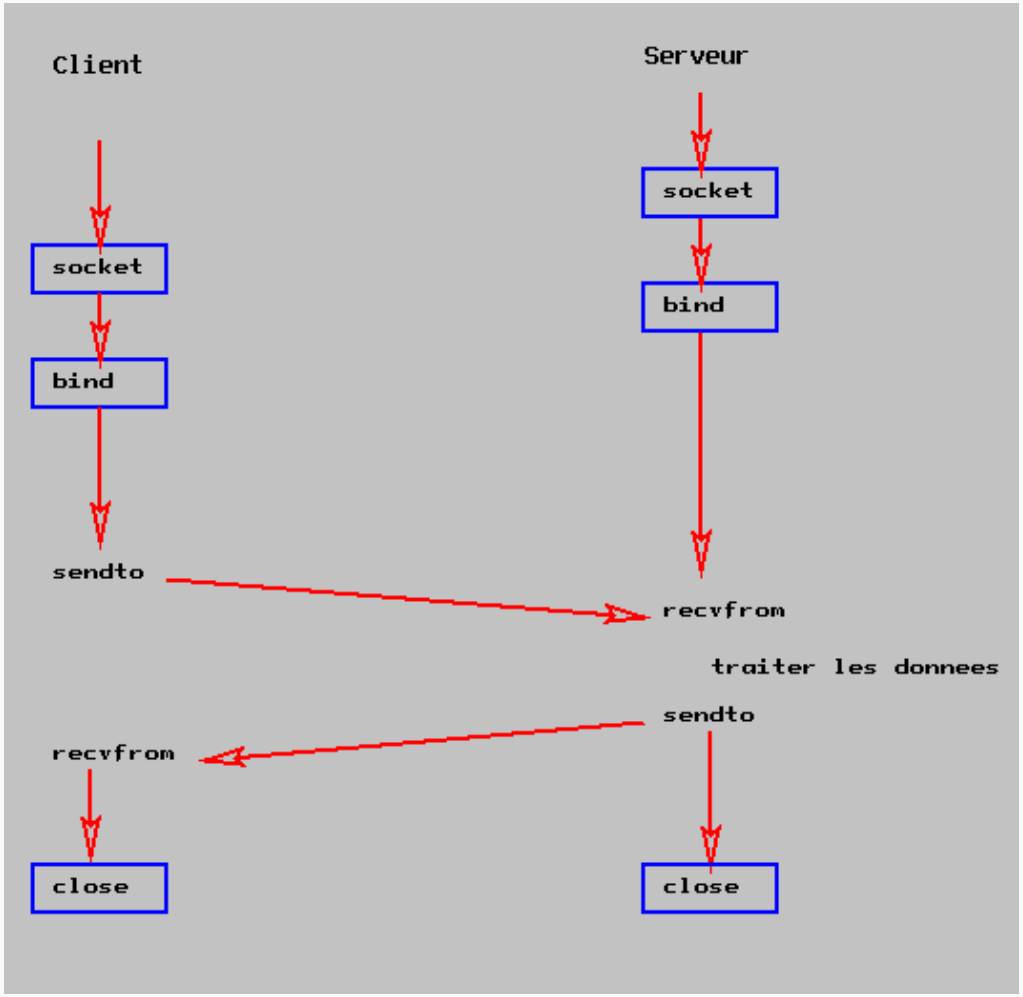


Schéma Mode Non Connecté



Exemples de serveurs

- Serveurs simples en mode connecté
- Serveurs multithreadés (ou multiprocessus) en mode connecté:
  - Le thread principal initialise la socket et attend les connexions (accept())
  - Crée un thread pour faire le service.
  - Le thread principal revient sur le accept.
- Serveur multithreadé avec pool de threads:
  - Idem mais le nombre de threads est limité
  - Principe du producteur/consommateur
  - Producteur: le thread principal, *produit* des connexions.
  - Consommateur: les threads de service.