

# Programmation Système

Philippe LALEVÉE  
ENSM-SE / CMP

Contributions de L. Mugwaneza

# Objectifs

- Principes généraux des systèmes d'exploitation
  - Vocabulaire
  - Architectures
- Présenter les mécanismes de programmation système
  - Algorithmique système
  - Plus forte maîtrise du langage C
- Réaliser un projet de développement
  - Le cadre est fixé
  - Le sujet est libre

# Fiche programme

- Séance 1 : Introduction
  - Présentation des S.E. et de l'API système
  - Environnement de développement et langage C
- Séances 2 et 3 : Les entrées-sorties
  - Appels système pour les E/S
  - Les drivers
  - Les FIFOS
  - Les sockets TCP/IP
- Séances 4 à 6 : Processus et *Threads*
  - Multiprogrammation « lourde »
  - Multiprogrammation « légère »
  - Serveurs multiprogrammés
- Séances 7 et 8 : Gestion de la concurrence
  - Exclusion mutuelle
  - Algorithmes classiques

# Organisation

- Projet « fil rouge »
  - Apprentissage par projet
  - Pratique de la programmation système
  - Réalisation d'un logiciel complet
- Évaluation
  - Projet
  - Questionnaire écrit

# Bibliographie

- Sources principales du cours

- *The Linux Programming Interface* (cf. **[LPI]** dans le support)  
M. Kerrisk, 2011, No Starch Press (**1556 pages**)
- *Linux System Programming*. R Love, O'Reilly 2013.

- Sources des transparents

- *Operating System Concepts. Slides*, Silberschatz et al.
- *Modern Operating Systems. Slides*, A. Tanenbaum.
- *Polycopié IAAI*, L. Mugwaneza et H. Quiroz, 2002.
- *Polycopié ISMEA*, H Ettaleb et al, 2005.

- Lectures conseillées

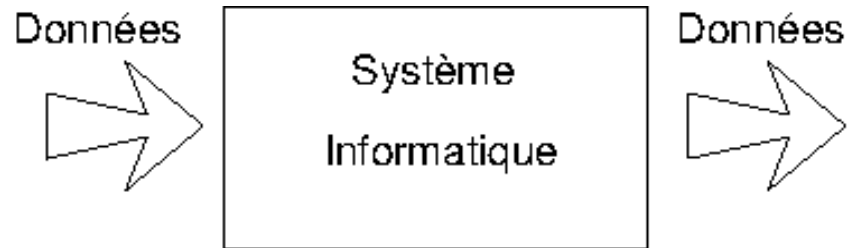
- *Operating System Concepts*.  
Silberschatz et al, Wiley, 2008
- *Systèmes d'exploitation*.  
A. Tanenbaum, Pearson Education, 2008.

Séance 1 : Systèmes d'exploitation et Programmation système

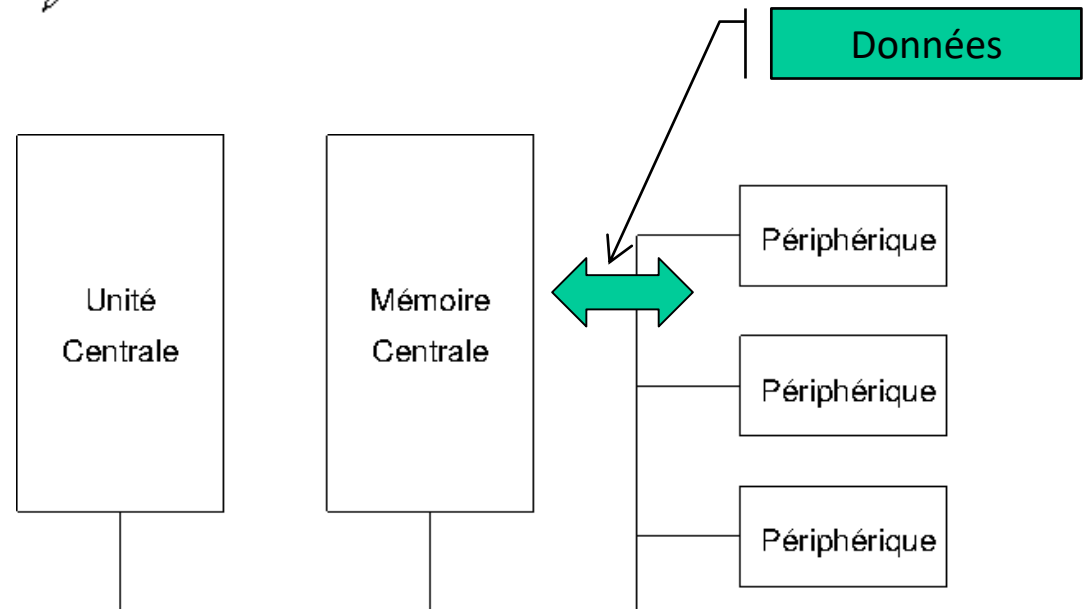
# INTRODUCTION GÉNÉRALE

# Objectifs d'un système informatique

- Traitement de l'information ...



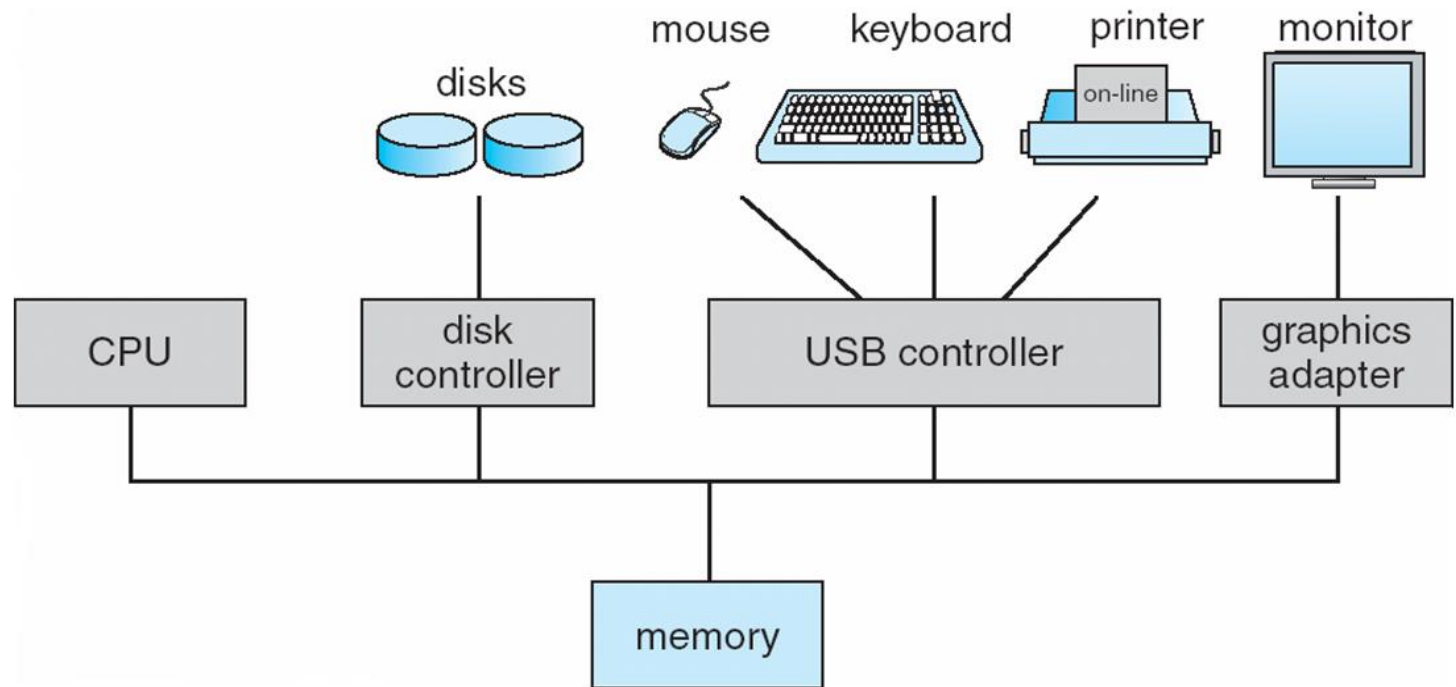
- ... par un ordinateur



# Organisation d'un ordinateur

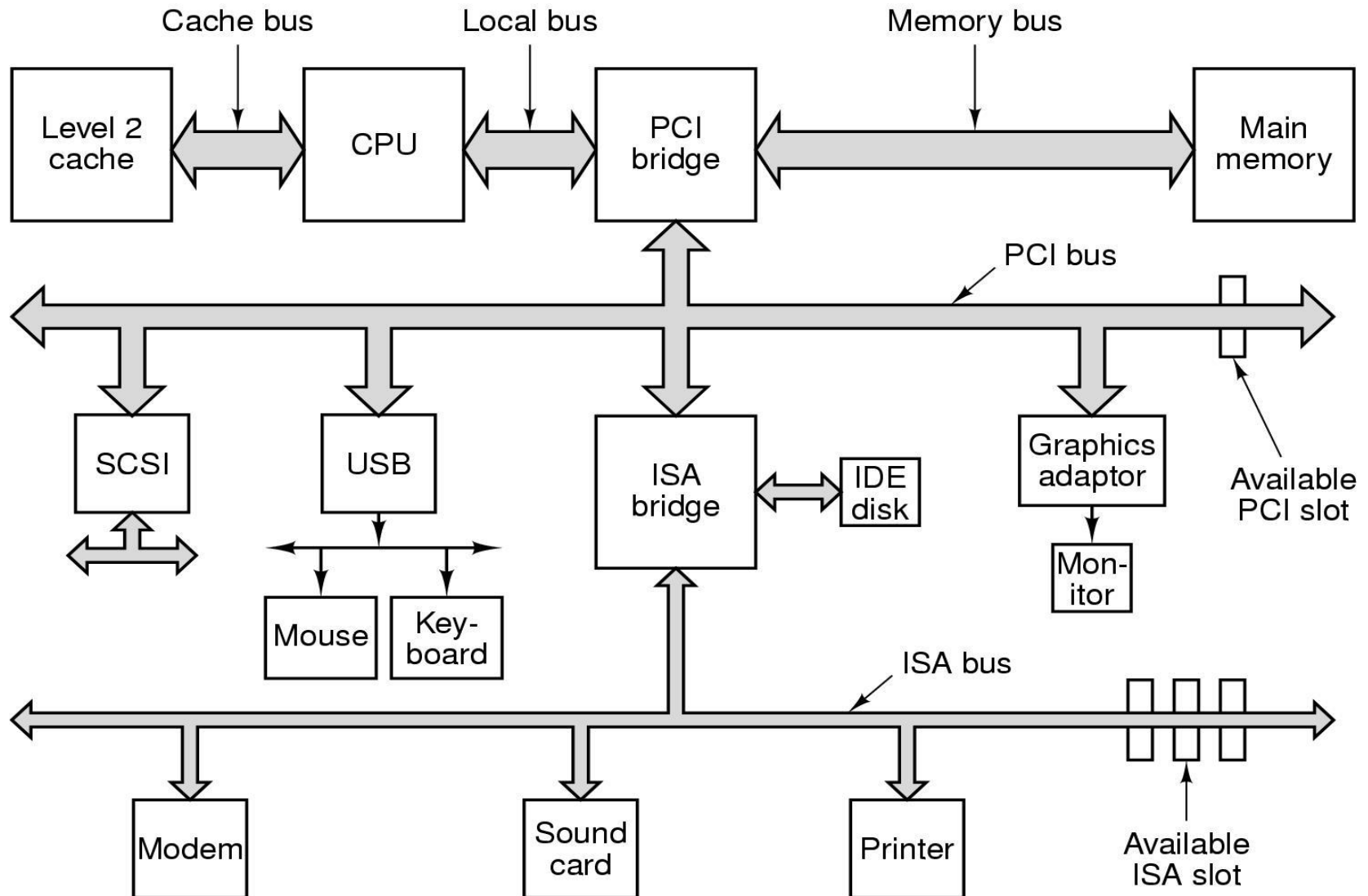
- Schéma général

- Un bus commun interconnecte un ou plusieurs CPU, des contrôleurs de périphériques et une mémoire partagée

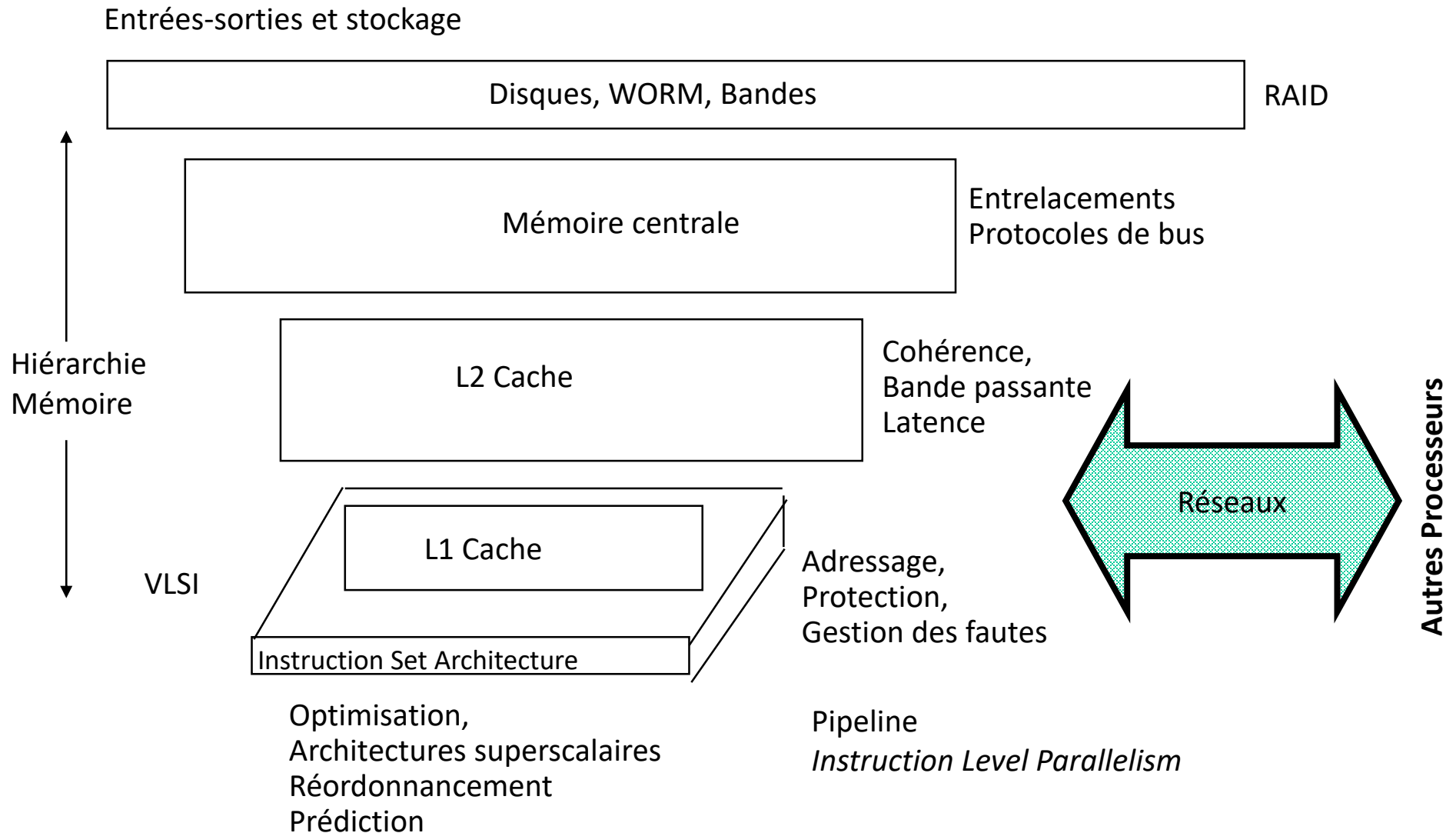




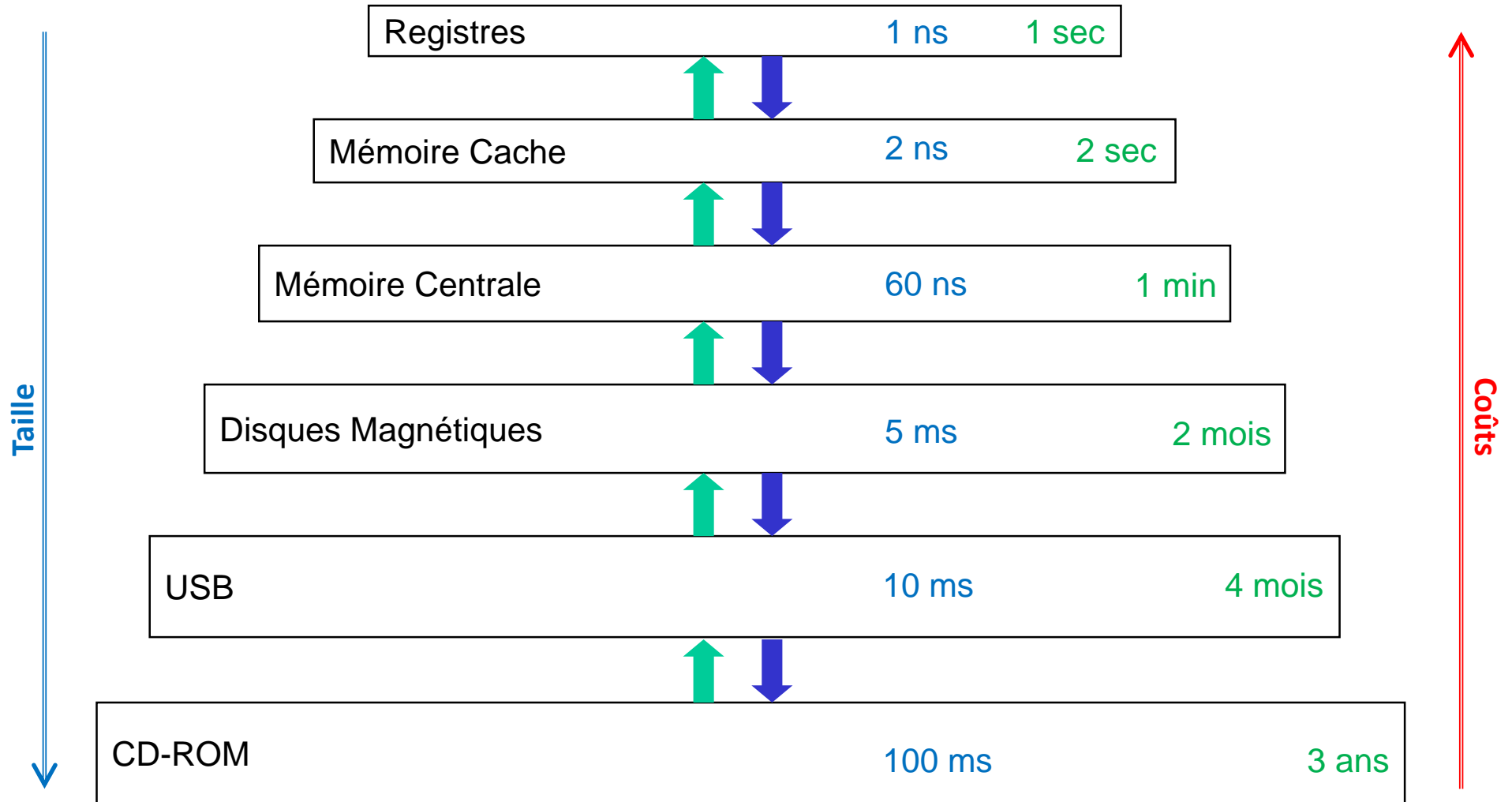
# Schéma général d'un PC



# Domaines fonctionnels d'un ordinateur

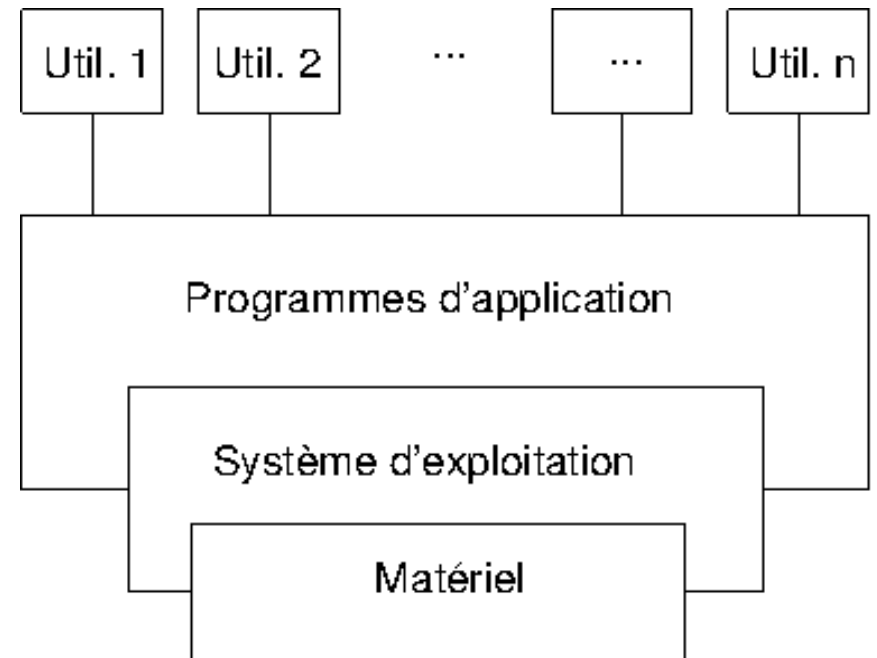


# Hiérarchie mémoire



# Nécessité d'un système d'exploitation

- Gestion des ressources matérielles
  - Partage entre « utilisateurs »
  - Performance d'accès
  - Sécurité
- Interface simplifiée pour l'utilisation
  - Information  $\longleftrightarrow$  Donnée
  - Machine idéale (virtuelle)



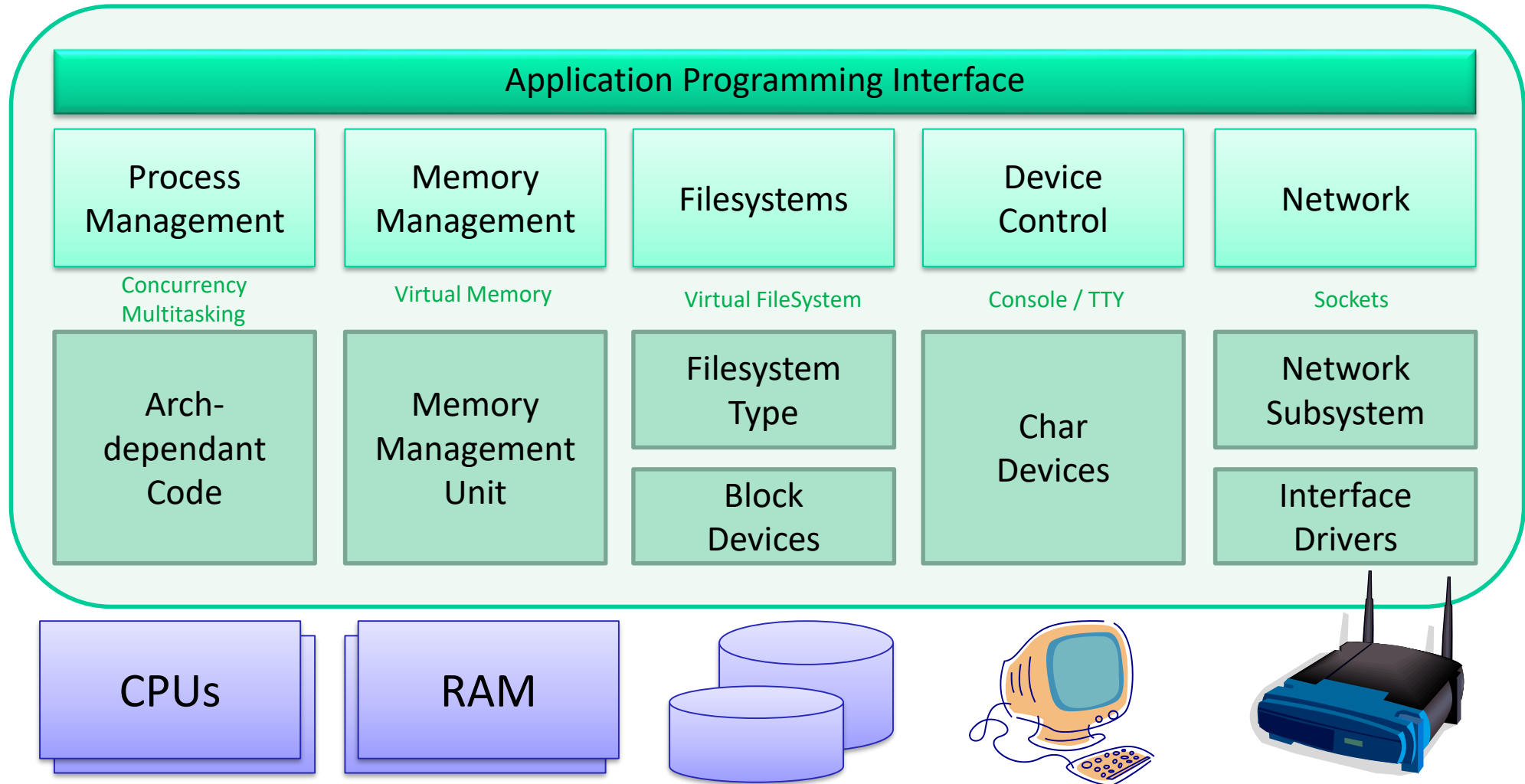
# Machine idéale (virtuelle)

- Émulation logicielle d'une machine abstraite
  - Le matériel « émulé » dispose des fonctionnalités dont les utilisateurs ont besoin
  - Programmation de la machine abstraite indépendante du matériel
- Simplicité de programmation
  - Chaque activité « croit » disposer de toutes les ressources CPU/mémoire
  - Chaque activité « croit posséder » tous les périphériques
  - Des périphériques différents ont la même interface
  - Les interfaces des périphériques sont plus « riches » que les périphériques bruts (*raw*)
- Isolation des fautes
  - Les activités ne s'impactent pas entre elles
  - Les erreurs ne « plantent » pas la machine (ne devraient pas)
  - Protection de l'exécution contre les dysfonctionnements

# Système d'exploitation ?

- Deux significations différentes :
  1. **Tout ce qui est « vendu » avec la machine !**  
Ensemble des logiciels qui gèrent les ressources de l'ordinateur, ainsi que les logiciels fournis (distribués) pour les utiliser : interface graphique, utilitaires, éditeurs...
  2. **LE programme qui est toujours en exécution !**  
Plus strictement, logiciel de base qui gère et alloue les ressources de l'ordinateur (CPU, RAM et périphériques)
- Le **noyau** (ou « *kernel* ») est un synonyme de la deuxième définition ➔ objet de ce cours

# Organisation du noyau



# Tâches du noyau : une interface plus simple

- API programmation système
  - Demandes de service au noyau
    - Point d'entrée dans le noyau = **appel système**
  - Nouveaux objets associés à des types de données abstraits (plus riches)
    - Notion de fichier et de système de fichiers
    - Notion de processus
      - Processus = exécution d'un programme en mémoire
      - Opérations sur les processus : création / terminaison des processus, communication
    - « Objets » servant à la communication/synchronisation entre processus
      - ex: sockets, FIFO, sémaphores...
- Gestion des utilisateurs
  - Les utilisateurs disposent de droits
  - Protection de l'ordinateur et des autres utilisateurs



# Tâches du noyau : gestion du matériel

## C'est un gestionnaire de ressources

- Gestion des ressources de l'ordinateur
  - Allocation du CPU (ordonnancement des processus)
  - Gestion du réseau
    - Accès et routage (couches 2 et 3)
  - Accès aux autres périphériques
    - Gestion des communications (entrées-sorties)
    - Partage des accès
    - Synchronisation / adaptation
- Partage et protection des ressources
  - Allocation de la mémoire
    - Partage et protection de la mémoire
    - Mémoire virtuelle (segmentée / paginée)
  - Gestion des disques
    - Partage et protection de l'espace disque

# Modes d'exécution

- Comment assurer le partage sécurisé des ressources ?
  - Les processeurs offrent deux modes d'exécution : utilisateur et noyau
  - Des instructions spéciales du processeur permettent de changer le mode
  - L'espace mémoire est séparé en espaces utilisateur et un espace noyau
- Mode utilisateur
  - Le processeur peut uniquement accéder à l'espace utilisateur courant
  - L'accès à l'espace noyau est protégé : déclenchement d'une exception
- Mode noyau
  - Le processeur peut accéder à tous les espaces utilisateur et à l'espace noyau
  - Certaines opérations ne peuvent être exécutées qu'en mode noyau : arrêter le système, accès à la MMU (unité de gestion mémoire), accès aux périphériques
- En utilisant ce mécanisme, puis en plaçant le système d'exploitation dans l'espace noyau, cela assure que :
  - Les processus utilisateur ne peuvent pas accéder aux instructions et aux données du noyau ni à celles des autres utilisateurs
  - Et ainsi, ils ne peuvent pas corrompre le système

# Programme utilisateur

- **Les processus utilisateurs sont l'exécution de programmes utilisateur**
- **Transparence de localisation en mémoire et de stockage de ses données**
- **Isolation**
  - Pas de communication avec les autres processus
  - Pas d'accès aux périphériques
  - Pas de connaissance des autres processus
- **Comportement asynchrone vis-à-vis du noyau**
  - Pas de connaissance de l'allocation du processeur
  - Pas d'accès direct aux événements

# Programmation noyau

- **Le noyau sait tout et contrôle tout**
- **Gestion des processus**
  - Allocation du(es) processeur(s)
  - Maintien des informations de gestion
  - Gestion des espaces mémoire et de stockage
- **Gestion des communications**
  - Inter médiation de tous les échanges
  - Gestion des entrées-sorties et des périphériques
- **Gestion de tous les événements**
  - Gestion de toutes les requêtes
  - Création et terminaison des activités

# Programmation système

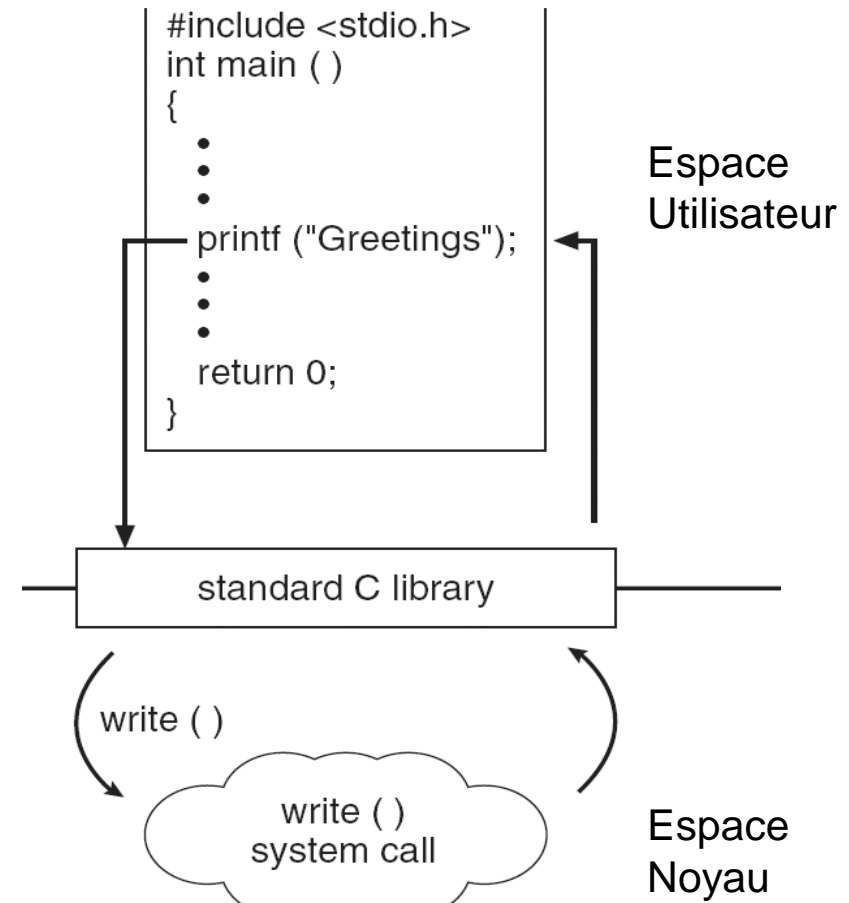
- Programme système
  - Programme utilisateur utilisant directement les services du système (API système, section 2 du man, appels système)
  - Pourquoi ? nouveau service du système, service permanent, optimisation, droits spéciaux...
- La programmation système
  - C'est écrire des programmes système
  - C'est utiliser le langage C !
  - Nécessite une bonne connaissance du fonctionnement du noyau
- Exemples
  - Serveur : Web, bases de données, mail, fichiers...
  - Outils : sauvegardes, lecteurs, machines virtuelles...
  - Systèmes embarqués

# Appels système

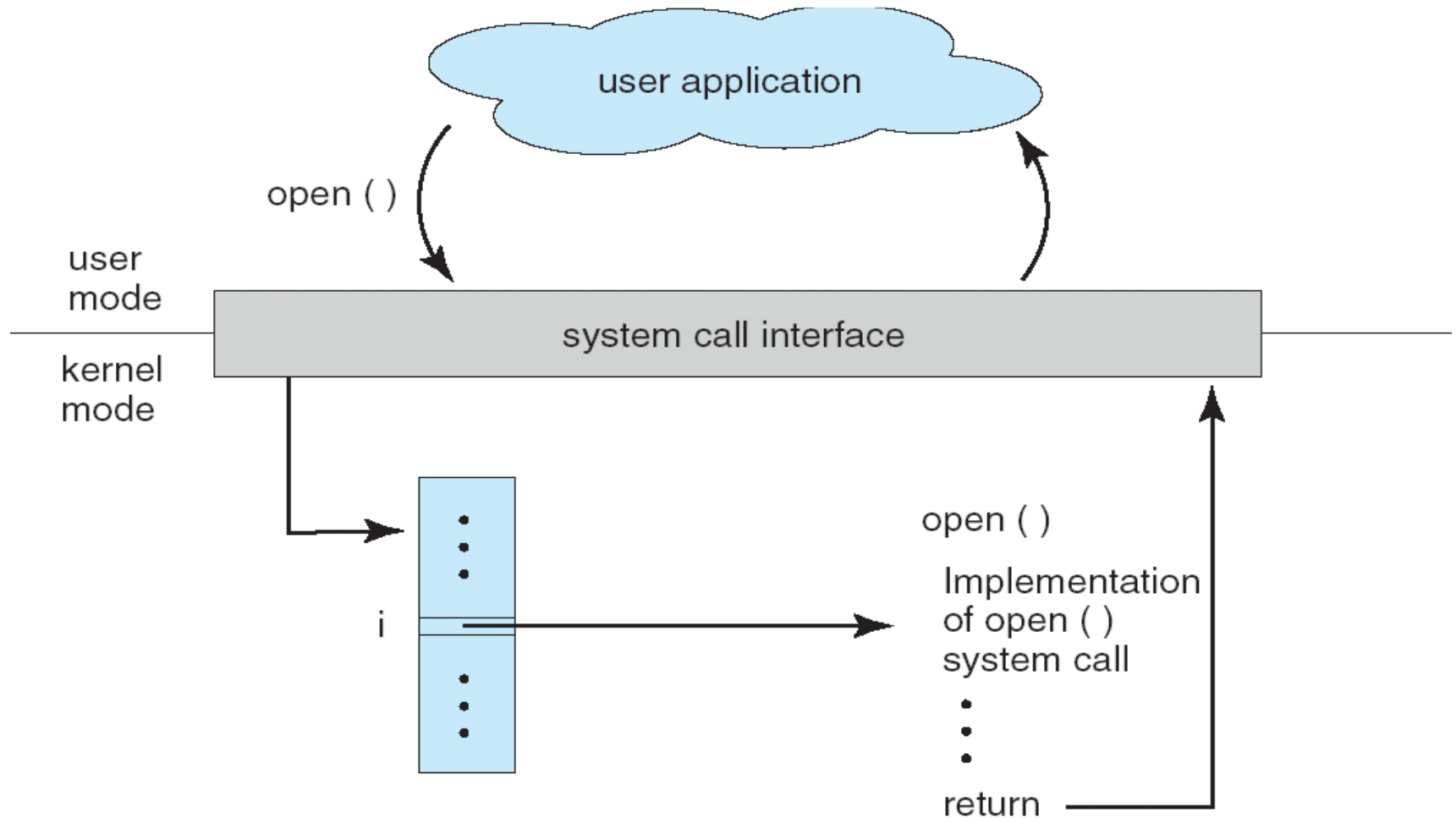
- **Appel système** = point d'entrée dans le noyau
  - Le paramétrage du service est à effectuer
  - Principe : interface C par appel de fonction (cf. section 2 du **man**)
  - Le nombre d'appels système (points d'entrée) est fixe
    - Chaque appel système est identifié par un numéro
    - Identification transparente pour le programmeur : table de nommage
- **Changement du mode d'exécution**
  - Passage en mode noyau lors de l'appel
  - Passage en mode utilisateur lors du retour
- **Paramètres et valeur de retour**
  - Transférés de l'espace utilisateur dans l'espace noyau lors de l'appel
  - En sens contraire, lors du retour
  - Idem pour les objets référencés (copie profonde)

# Programme C / appel système

1. Un programme appelle la fonction **printf()** de la bibliothèque standard du C (prototype dans **<stdio.h>**)
2. **printf** appelle la fonction **write()** qui est l'interface C de l'appel système du même nom (passage en mode noyau)
3. Cette fonction du noyau transmet les paramètres et les buffers de l'espace utilisateur vers l'espace noyau et invoque l'appel système
4. Elle récupère la valeur de retour et la renvoie au programme utilisateur
  - **write()** : nombre d'octets
  - **printf()** : nombre de caractères
  - Par convention, une valeur négative (-1) indique une erreur et la variable globale **errno** est positionnée



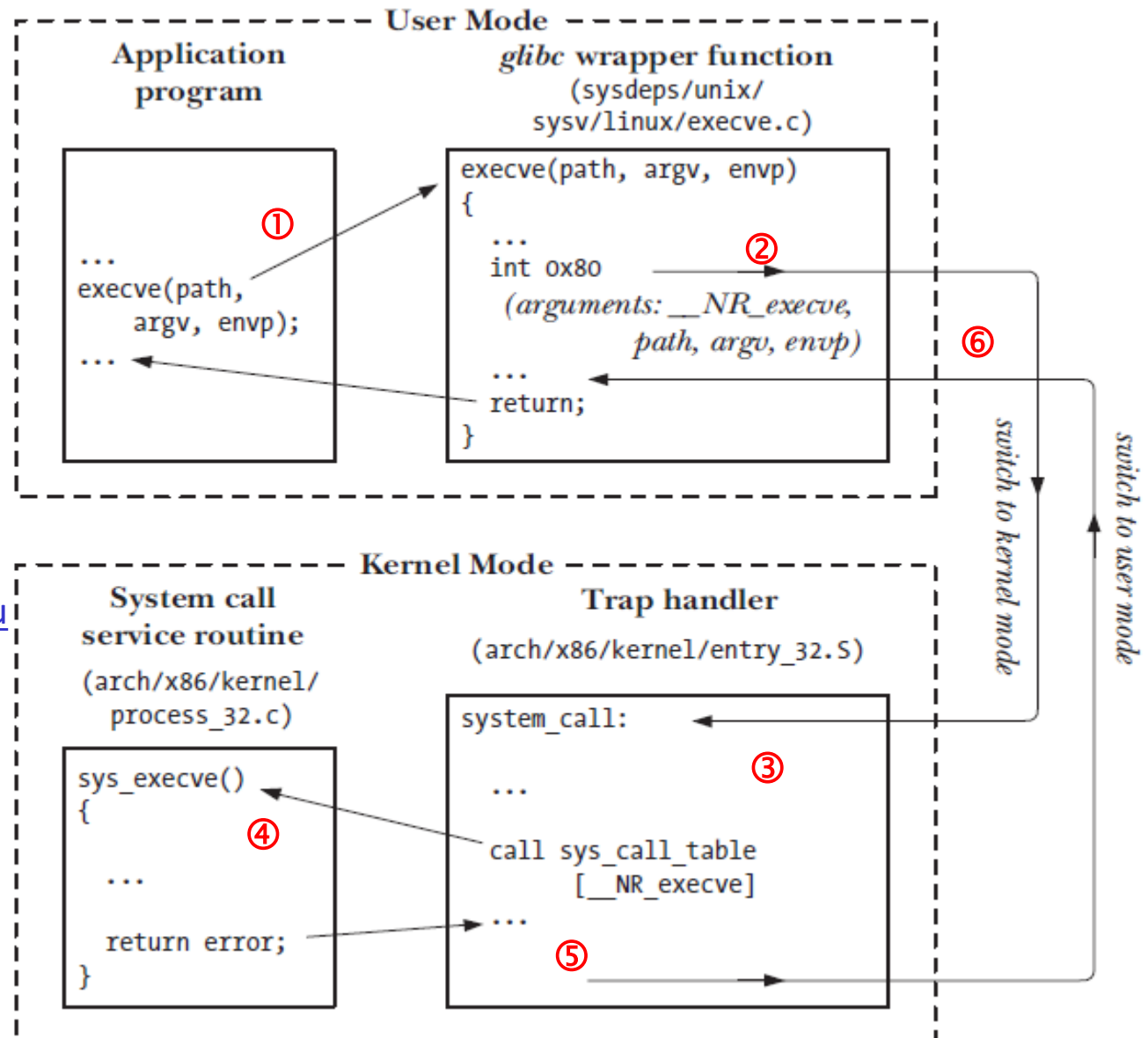
# Appel système / noyau





# Détails sous Linux

1. Interface C / appel système
2. Instruction d'appel système (*trap instruction*) de passage en mode noyau (ici int)
  - Paramètres passés dans les registres ou pile
  - dont le numéro de l'appel (`__NR_execve`)
3. Invocation dans le noyau du gestionnaire d'exception (*trap handler*)
  - Sauvegarde du contexte
  - Transfert des arguments et des données espace utilisateur vers espace noyau
4. Invocation de l'appel système (table indicé par numéro)
5. Restauration du contexte et transmission de la valeur de retour
6. Passage en mode utilisateur et transmission de la valeur de retour et des données (instruction spéciale du CPU pour changer de mode)



# Espaces mémoire

- Séparation des espaces d'adressage
  - Un espace pour le noyau
  - Un espace utilisateur pour chaque processus
- Espace noyau
  - Mémoire réelle (physique)
  - Adressage direct autorisé si mode noyau
- Mémoire virtuelle
  - Translation entre une adresse virtuelle et une adresse physique
  - Effectué par la MMU « *Memory Management Unit* »
  - Isolation des espaces, espace processus logique
- Types de mémoire virtuelle :
  - Segmentée : la mémoire des processus est découpée en segments repérés par une adresse de début et une taille
    - Adresse virtuelle = numéro de segment et déplacement à l'intérieur de celui-ci
    - Les segments sont associés au programme et ont des attributs : lecture seule, partageable...
  - Paginée : la mémoire des processus est découpée en pages de longueur fixe
    - Adresse virtuelle = numéro de page et déplacement à l'intérieur de celle-ci
    - Les pages ont des attributs : lecture seule, modifié, date...
  - Hybride : pagination d'un segment (translation spécifique à chaque segment)

# Gestion des erreurs

- Obligatoire au retour de tous les appels système
  - Valeur non négative → succès
  - Valeur négative → erreur
    - En général -1 → consulter le manuel en ligne (**man**)
    - La variable globale **errno** contient le code d'erreur
    - Codes définis dans **<errno.h>**
    - Ils débutent par **E**, par exemple **EAGAIN**, **EINTR**...
    - Consulter la section **ERRORS** du **man**
- Affichage « en clair » : **perror** (prototype dans **stdio.h**)
  - **void perror(const char \*msg);**
  - Le message **msg** est affiché avant le texte d'erreur
  - **perror("ouverture")** avec **errno** valant **ENOENT** affichera  
ouverture: No such file or directory

**CONSULTER LE MANUEL EN LIGNE !!**

# Comment programmer ?

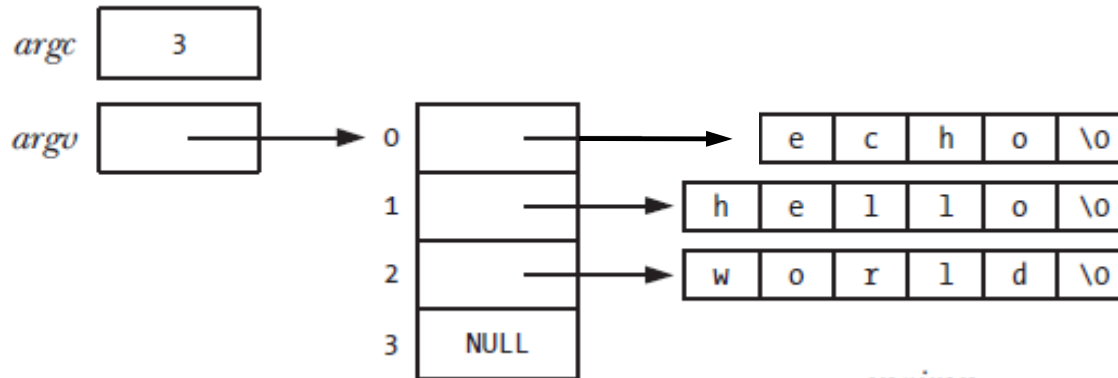
- Programmation C classique
  - Programmation modulaire
  - Indentation / commentaires / nommage ...
  - Utilisation de **gcc** avec les options usuelles
    - **gcc -Wall -ansi -o fichier\_exec fichiers.c**
- Fichiers d'inclusion
  - **<unistd.h>** types, constantes et appels systèmes Unix standards
  - **<sys/types.h>** constantes et types de données standards
  - **<sys/stat.h>** caractéristiques des fichiers
  - **<fcntl.h>** opérations de contrôle sur les fichiers
  - **<errno.h>** codes d'erreur
  - Et **<stdio.h>**, **<stdlib.h>**, **<string.h>** ...

# Écrire des programmes portables

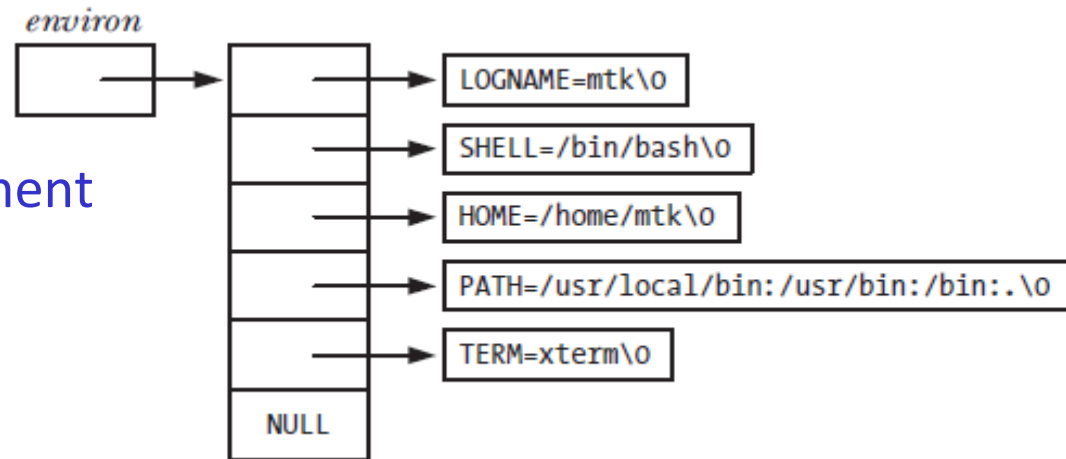
- Nombreuses variantes Unix : Linux, FreeBSD, NetBSD, OpenBSD, Solaris...
- Standards et normes
  - POSIX (*Portable Operating System Interface*) : définition de services système
  - POSIX.1 : Standard IEEE et ISO (ISO/IEC 9945-1:1990)
  - Nombreuses variantes : POSIX.1[a-g], POSIX-2, FIPS 151, Open Group
  - Fusion IEEE / ISO / Open Group
    - SUSv3 : document de spécifications de 3700 pages !
    - Actuellement : SUSv4 (*Single Unix Specifications, version 4*)
  - LSB (*Linux Standard Base*) : compatibilité binaire entre distributions
- Taille et type des données variables → utilisez les types standards (sys/types.h)
  - **size\_t** : taille en octets                      **off\_t** : position en octets dans un fichier
  - **id\_t** : identifiant                              **time\_t** : temps en secondes depuis le 1/1/1970
  - ...
- Voir le manuel des fonctions (**man**) pour connaître
  - Les fichiers d'inclusion standards à ajouter à vos fichiers C
  - Les types standards utilisés (et à utiliser)
  - Les erreurs retournées par les appels système (à gérer)
  - Les standards en consultant si elle est présente, la section **CONFORMING TO**

# Pratique du C

- Afficher les arguments de la fonction **main()**
  - int main ( int argc, char \*argv[] ) ;**
  - Exemple : **\$ echo hello world**

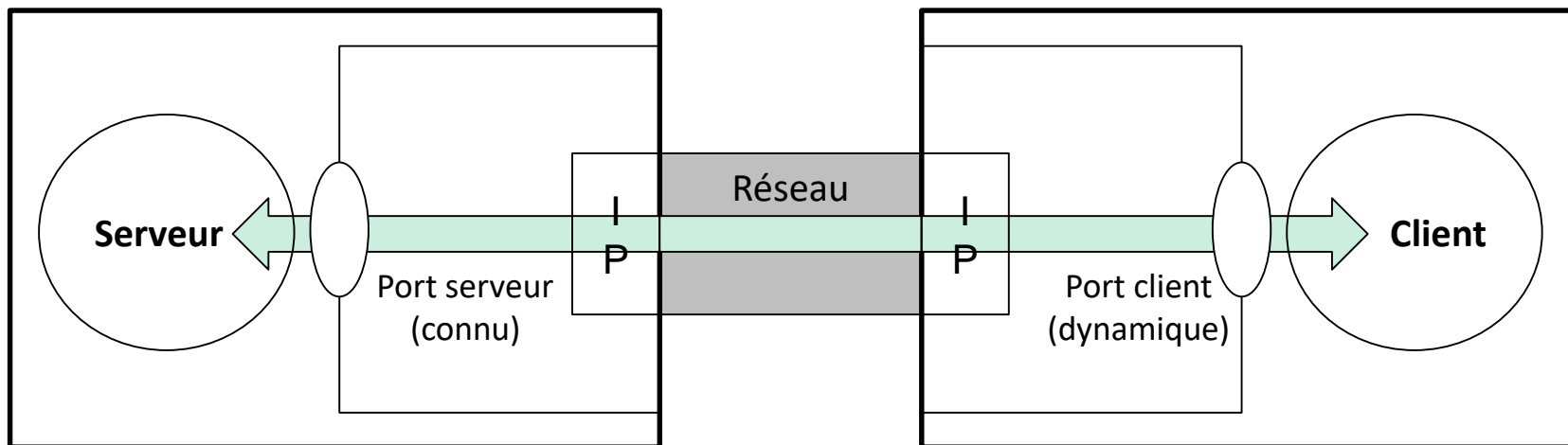


- Afficher les variables d'environnement du programme
  - extern char \*\* environ ;**
  - Recherche avec **getenv()**



# Rappels sur les adresses IPv4

- Adresses sur 4 octets (32 bits)
  - Exemple : 192.168.10.10 (0xC0A80A0A, ou 3232238090)
  - Classe, masque, *subnetting* : transparents pour les applications
- Numéro de port sur deux octets (16 bits)
  - De 1 à 1023 : ports « bien connus » (réservés)
  - De 1024 à 49151 (0xC000 - 1) : ports enregistrés
  - Au-delà : ports dynamiques (utilisés pour le client)



# Représentation des données

- Représentation des entiers

- Network Order : Internet est *Big Endian* (dans les mots de 16 ou 32 bits les octets sont stockés avec les poids forts dans les adresses basses)

3	2	1	0
---	---	---	---

- Host Order : les PC (x86) sont *Little Endian* (dans les mots de 16 ou 32 bits les octets sont stockés avec les poids faibles dans les adresses basses)

0	1	2	3
---	---	---	---

- Fonctions de conversion pour les adresses IP et numéros de port à transmettre (prototypes dans **<arpa/inet.h>**)

- Avant l'émission : **htons()** ( **htonl()** ) → conversion d'un **short** ( **long** ) de « *host order* » vers « *network order* »
- Après la réception : **ntohs()** ( **ntohl()** ) → conversion d'un **short** ( **long** ) de « *network order* » vers « *host order* »



# Résolution DNS avec **getaddrinfo()**

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

```
int getaddrinfo (const char *host, const char *service,
                  const struct addrinfo *hints, struct addrinfo **result) ;
```

Retourne 0 en cas de succès, ou un code d'erreur

- **host** : nom de la machine dont on cherche l'adresse IP
- **service** : nom du service dont on cherche le numéro
- **hints** : critères de recherche
- **result** : liste chaînée de structures résultat, allouées par la fonction (nécessite l'appel de **freeaddrinfo()** pour la libérer)
- Les adresses IP et numéro de port sont dans le *Network Order*
- La gestion du code d'erreur n'est pas gérée par **perror()** mais par **gai\_strerror()** qui retourne le message en clair

```
struct addrinfo *resultat;
if ( (code = getaddrinfo ( "www.emse.fr", "http", &hints, &resultat ) ) != 0 ) {
    fprintf ( stderr, "getaddrinfo: %s\n", gai_strerror ( code ) );
    exit ( EXIT_FAILURE );
}
/* voir transparent suivant pour hints et resultat */
freeaddrinfo ( resultat );
```

# Utilisation de struct addrinfo

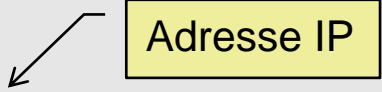
```
/* extrait de <netdb.h>
struct addrinfo {
    int ai_flags;
    int ai_family; /* Address family */
    int ai_socktype; /* Type: SOCK_STREAM, SOCK_DGRAM */
    int ai_protocol; /* Socket protocol */
    size_t ai_addrlen; /* Size of structure pointed to by ai_addr */
    char *ai_canonname; /* Canonical name of host */
    struct sockaddr *ai_addr; /* Pointer to socket address structure */
    struct addrinfo *ai_next; /* Next structure in linked list */
};
```

- En **vert**, les champs à positionner dans **hints**
  - **ai\_family** : y mettre **AF\_INET**
  - **ai\_socktype** : y mettre **SOCK\_STREAM**
- En **rouge**, les champs utiles de la structure pointée par **result**
  - **ai\_addr** : paramètre de type **sockaddr\_in** après *cast*
  - **ai\_next** : si plusieurs adresses sont associées au domaine, permet d'en parcourir la liste
- **ai\_canonname** : nom officiel (par exemple, le nom canonique de **www.emse.fr** est **wawawa.emse.fr**, cf. commande **host** pour confirmer)

# Type struct sockaddr\_in

- **sockaddr** est un type générique pour tout type de réseau
- Utilisation de **struct sockaddr\_in** pour les réseaux IP (nécessité de *cast* pour accéder aux champs)

```
/* extrait de <netinet/in.h> */
struct in_addr {
    in_addr_t s_addr; /* entier non signé de 32 bits */
};
struct sockaddr_in {
    sa_family_t    sin_family; /* AF_INET, cf. socket() */
    in_port_t      sin_port;   /* numero de port sur 2 octets (network order) */
    struct in_addr sin_addr;    /* contient l'adresse IP (network order) */
    unsigned char  _pad...;    /* padding de correspondance avec sockaddr */
};
```



```
/* exemple d'affichage */
struct sockaddr_in adresse;
...
printf( "port %hd", ntohs ( adresse.sin_port )); /* port (network order) */
printf( "adr %x", ntohl ( adresse.sin_addr.s_addr ) /* IP (network order) */
```

# Exemple complet de requête

```
/* requete DNS */

struct addrinfo *infos, hints;
int code;

memset ( &hints, 0, sizeof (struct addrinfo)); /* remise à zéro des champs */

hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;

code = getaddrinfo ( "www.emse.fr", "http", &hints, &infos ) ;
if ( code != 0 ) {
    fprintf ( stderr, "Erreur: %s\n", gai_strerror(code) );
    exit ( EXIT_FAILURE ) ; }

...
/* struct sockaddr_in *adresse = (struct sockaddr_in *) infos->ai_addr ;

    adresse->sin_addr.s_addr doit contenir 0xC131AEC2,
                                   ie 193.49.174.194 */
    adresse->sin_port doit contenir 80 */
...
freeaddrinfo ( infos );
```

Séance 2 : Premiers appels système

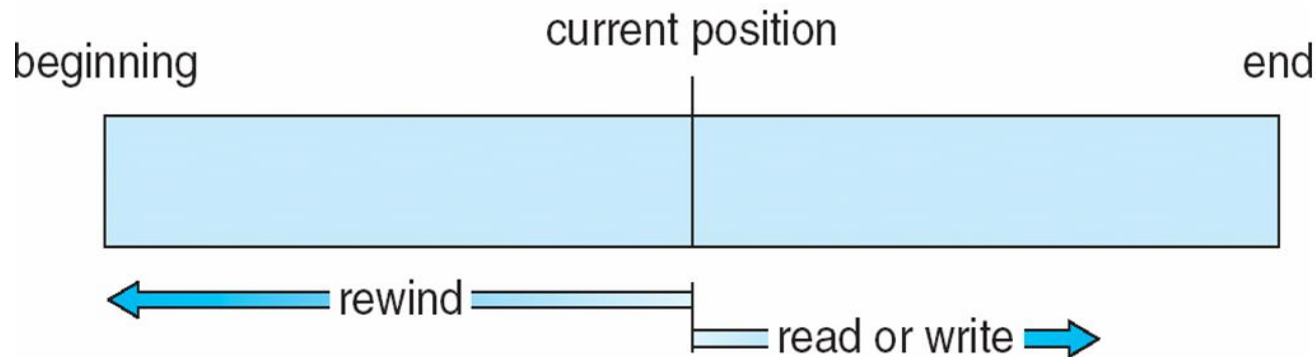
# LES ENTRÉES-SORTIES

# Entrée-sortie : accès à un fichier

- Fichier : abstraction fondamentale Unix
  - Principe « **tout est fichier** »
  - Toute entrée-sortie est réalisée sur un fichier quelle que soit sa nature
- Pour accéder à un fichier, il doit être **ouvert**
  - Correspondance entre le processus et le fichier lui-même
  - Le noyau maintient une table des fichiers ouverts indexée par des **descripteurs** (entier)
- Différents types de fichier
  - Fichiers normaux : nos fichiers de données (utilisateur)
  - Répertoires : gestion des liens (*hardlinks*)
  - Liens symboliques : *raccourcis*
  - Fichiers spéciaux : périphériques de type *block*, périphériques de type *char*, tubes nommés et sockets Unix

# Fichiers normaux

- Abstraction du S.E.
  - Vision uniforme et logique indépendante du support
  - Correspondance entre une vision utilisateur et des caractéristiques physiques
  - Plus petite unité d'allocation ayant un **nom logique**, visible par les applications
- Espace d'adressage logique contigu non typé (flot d'octets)



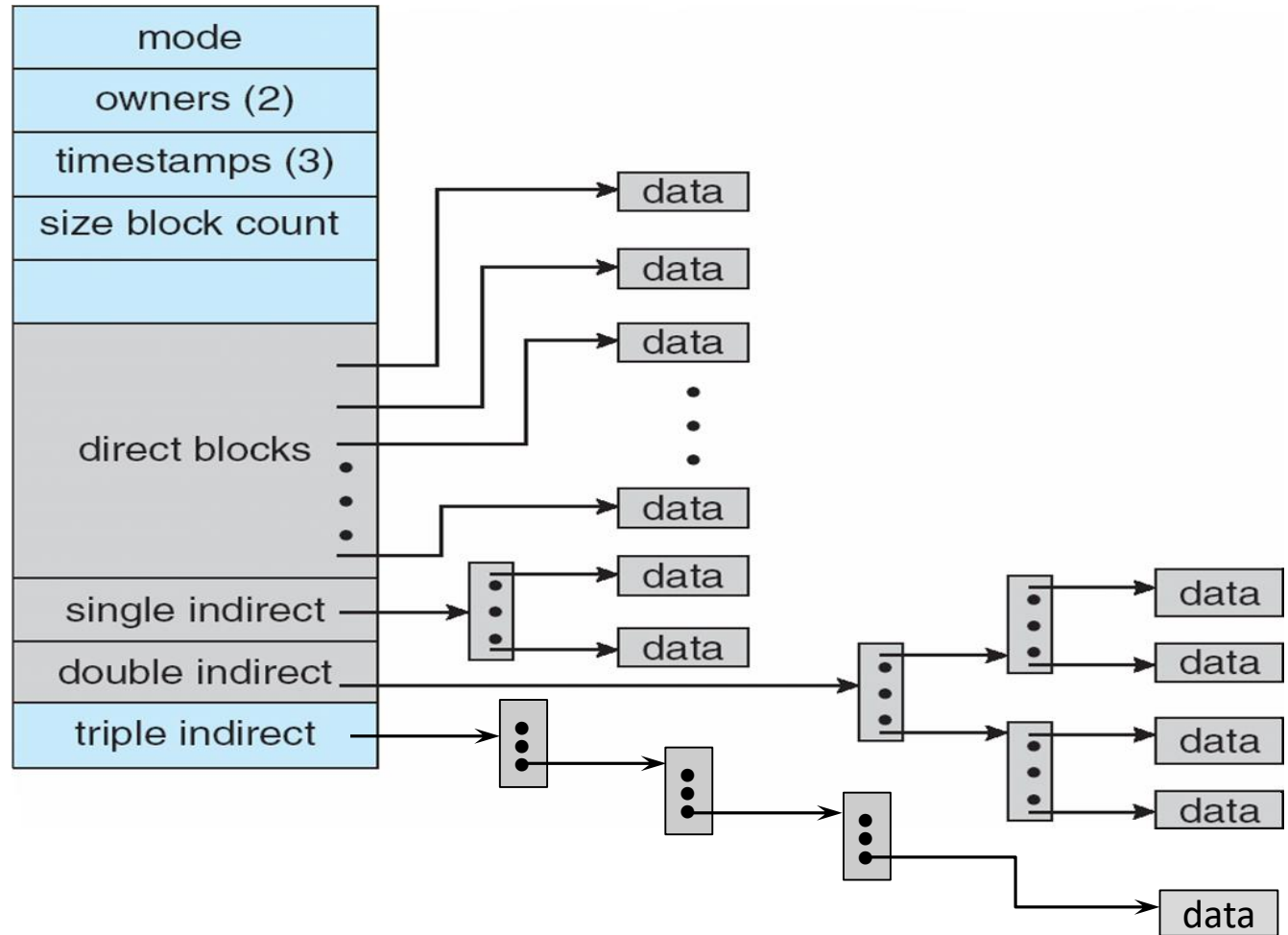
# Les répertoires et liens

- Les fichiers sont référencés par des *inodes* dans le système de fichiers
  - Structure de données de gestion du fichier
  - Repérée par un entier **unique** dans le système de fichiers
  - ➔ Système de fichiers = espace d'adressage
- Nécessité d'une fonction annuaire : les répertoires
  - Répertoire = fichier contenant des couples (nom fichier, numéro inode)
  - Ces couples sont appelés des liens (**hard links**)
  - Deux répertoires « point d'entrée » :
    - La racine du SGF : le nom commence par '/' ➔ désignation absolue
    - Le répertoire courant ('.'): désignation relative sinon
- Liens symboliques
  - Fichier spécial contenant une désignation (nom de fichier)

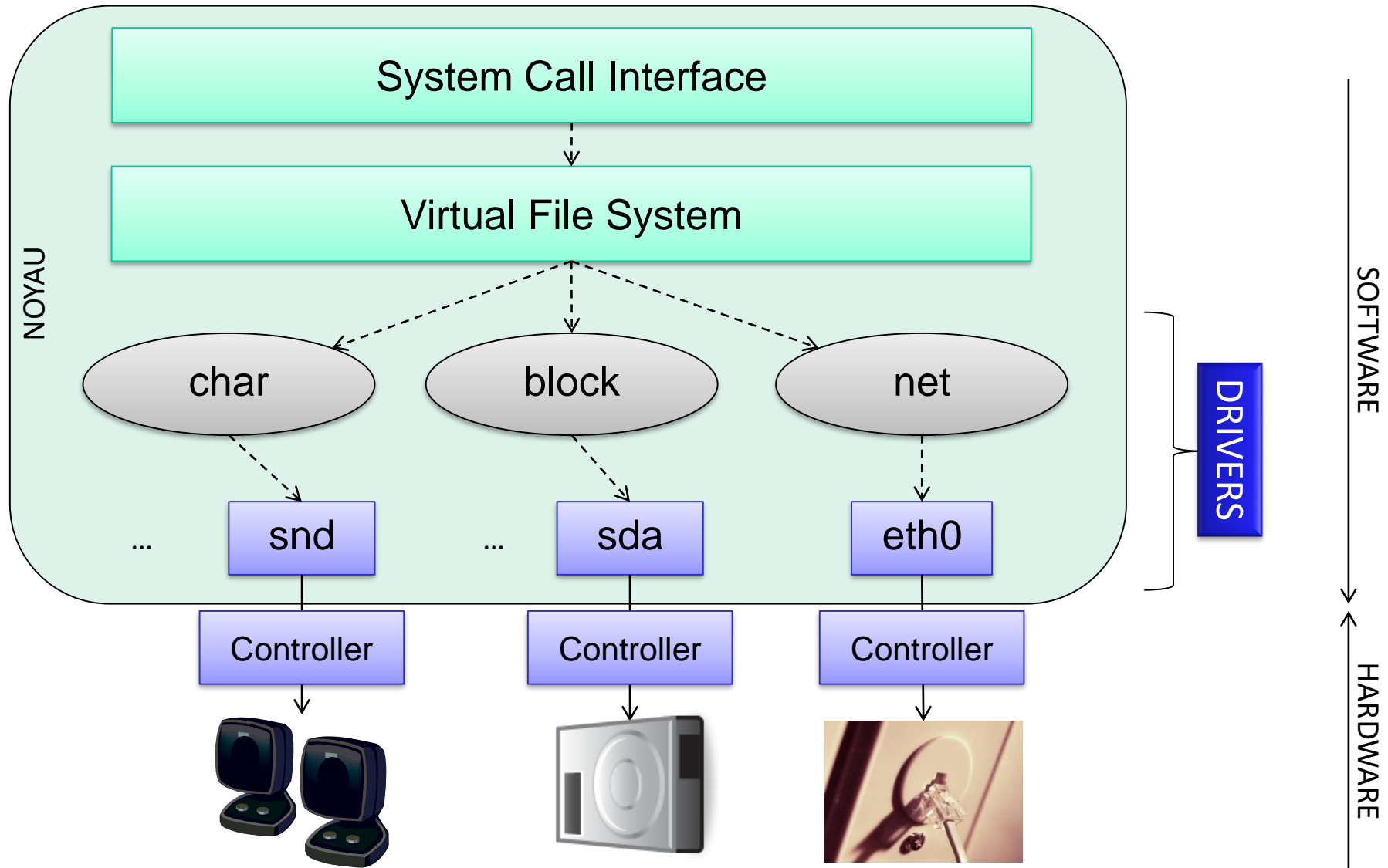


# Structure de gestion : *inode*

- Chaque fichier est repéré par un *inode*
  - Type, mode
  - Droits, accès
  - Taille, etc...
- Données contenues dans des blocs logiques
- L'inode contient la liste indexée des blocs alloués
- Numéro d'inode unique par SGF
- Stockés sur disque et en mémoire du noyau



# Les fichiers spéciaux (1/2)



# Les fichiers spéciaux (2/2)

- Les périphériques sont accédés par des fichiers spéciaux présents dans **/dev**
  - Repérés par deux entiers : **majeur** et **mineur**
  - Le majeur identifie le périphérique et le mineur identifie l'unité
    - **/dev/sda1** : **majeur 8** et **mineur 1** (disque SATA (**sd**), première unité (**a**) et première partition (**1**) )
    - **/dev/tty1** : **majeur 4** et **mineur 1** (télétype (**tty**) et premier terminal (**1**) )
  - Exemples particuliers:
    - **/dev/null** : les écritures sont ignorées
    - **/dev/full** : périphérique toujours plein
    - **/dev/zero** : les lectures ne retournent que des zéros binaires
    - **/dev/random** : les lectures retournent des nombres (pseudo-) aléatoires
- Créés avec **mknod nom type majeur mineur**
  - ou dynamiquement depuis les versions récentes de Linux
- Associés avec un driver (**/lib/modules/...**)
  - Interfaces standardisées
  - Commandes associées : **lsmod**, **modinfo**, **insmod**, **modprobe**, **rmmod**
  - Appel système pour accéder au driver : **ioctl** (faire **man**)

# Modèle universel des entrées-sorties

- Mêmes appels système pour tous les types de fichiers, y compris les périphériques

- **open()** : ouverture d'un « fichier »
- **read()** : lecture d'octets à partir d'un « fichier »
- **write()** : écriture d'octets vers un « fichier »
- **close()** : fermeture d'un « fichier »

- Vision SUSv4 (Unix) d'un fichier

- Flot (« *stream* ») séquentiel d'octets
- Accès direct (« *random* ») pour les fichiers disque avec **lseek()**
- Pas de marque de fin de fichier !

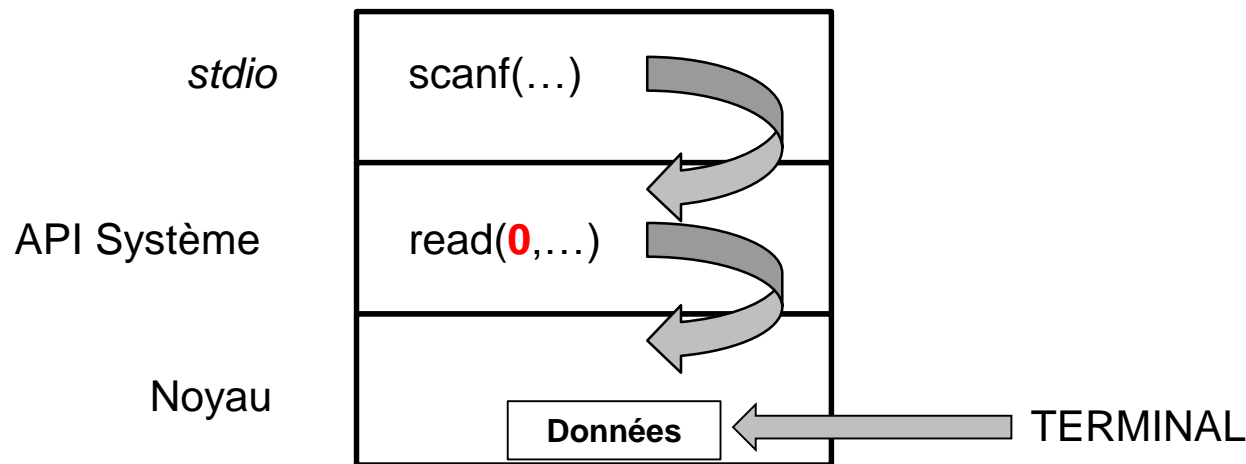
- Fichiers d'inclusion

- **<sys/types.h>**
- **<sys/stat.h>**
- **<fcntl.h>**
- **<unistd.h>**

**Fin de fichier = lecture qui ne retourne pas de donnée**

# Descripteurs de fichier

- Référence à un fichier ouvert : *file descriptor*
  - Entier (**short**) non négatif
  - Obtenu par un appel à **open()** avec un chemin d'accès
- Chaque processus « hérite » de 3 descripteurs de fichier ouvert
  - 0 : entrée standard (POSIX : **STDIN\_FILENO**) (stdio : *stdin*)
  - 1 : sortie standard (POSIX : **STDOUT\_FILENO**) (stdio : *stdout*)
  - 2 : erreur standard (POSIX : **STDERR\_FILENO**) (stdio : *stderr*)
- Lien avec les entrées-sorties standards (prototypes dans **<stdio.h>**)
  - **fopen()**, **fclose()**, **scanf()**, **printf()**, **fread()**, **fwrite()**...



# open() : ouverture d'un fichier

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

```
int open ( const char *chemin, int acces [, mode_t droits]);
```

Retourne un (le plus petit) descripteur de fichier, ou -1 en cas d'erreur

- **chemin** : nom du fichier (chemin d'accès)
- **acces** : mode d'accès au fichier (une seule valeur parmi les 3)
  - **O\_RDONLY** : lecture seule
  - **O\_WRONLY** : écriture seule
  - **O\_RDWR** : lecture et écriture
  - Autres modes d'accès usuels, à ajouter avec | (« ou »)
    - **O\_CREAT** : création du fichier
    - **O\_APPEND** : écritures en fin de fichier
    - **O\_EXCL** : création exclusive (échec si le fichier existe lors de sa création avec **O\_CREAT**)
    - **O\_TRUNC** : écrasement du fichier (lecture ou écriture)
- **droits** (optionnels) : permissions si **O\_CREAT**
  - Valeur octale (équival. à **chmod**) : rwxrwxrwx (3 bits pour User, Group et Other)

```
int lec = open("/etc/passwd", O_RDONLY);
if ( lec == -1 ) { perror ( "open" ); exit ( EXIT_FAILURE ); }
int cre = open("donnee.txt", O_WRONLY | O_CREAT | O_TRUNC, 0600);
int fle = open("rapport.log", O_RDWR | O_CREAT | O_APPEND, 0644);
```

# read() : lecture d'un fichier

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

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

Retourne le nombre d'octets lus, 0 en cas d'EOF, ou -1 en cas d'erreur

- **fd** : descripteur du fichier (ouvert) à lire
- **buffer** : adresse d'une zone mémoire dans laquelle stocker les données lues
- **count** : nombre maximum d'octets à lire

```
char text [20];
ssize_t nb = read ( lec, text, 20 ); /* lec: cf. open() */
if ( nb == -1 ) { perror ( "read" ); exit ( EXIT_FAILURE ); }
else if ( nb == 0 ) {
    printf ( "fin de fichier\n" );
}
else if ( nb < 20 ) {
    printf ( "autre lecture de %d octets a faire...", 20 - nb);
}
else
    printf ( "lecture de 20 octets\n" );
```

# write() : écriture dans un fichier

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

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

Retourne le nombre d'octets écrits ou -1 en cas d'erreur

- **fd** : descripteur du fichier (ouvert) à lire
- **buffer** : adresse d'une zone mémoire contenant les données à écrire
- **count** : nombre d'octets à écrire

```
int val [20];  
size_t cpt = 20 * sizeof ( int );  
ssize_t nb = write ( cre, val, cpt ); /* cre: cf. open() */  
if ( nb == -1 ) { perror ( "write" ); exit ( EXIT_FAILURE ); }  
else if ( nb < cpt ) {  
    printf ( "autre ecriture de %d octets a faire...", cpt - nb );  
}  
else  
    printf ( "ecriture de 20 entiers\n" );
```



# **lseek()** : positionnement dans un fichier

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

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

Retourne la nouvelle position ou -1 en cas d'erreur

- **fd** : descripteur du fichier (ouvert) à positionner
- **depl** : nombre d'octets à se déplacer (éventuellement négatif)
- **whence** : à partir d'où se positionner
  - **SEEK\_SET** : à partir du début du fichier (**depl** positif)
  - **SEEK\_CUR** : à partir de la position courante (**depl** négatif ou positif)
  - **SEEK\_END** : à partir de la fin du fichier (**depl** négatif)

```
off_t depl = -10;
off_t pos = lseek ( lec, depl, SEEK_CUR ); /* lec: cf. open() */
if ( pos == -1 ) {
    perror ( "lseek" );
    exit ( EXIT_FAILURE );
}
/* la prochaine lecture relira les dix derniers caractères du tableau text */
```

# close() : fermeture d'un fichier

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

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

Retourne 0 en cas de succès ou -1 en cas d'erreur

- **fd** : descripteur du fichier (ouvert) à fermer
- Écrit les données encore présentes en mémoire cache
- Libère les ressources associées au fichier
- Le retour de la fonction **main()** ou l'appel explicite à **exit()** lance la fermeture automatique de tous les fichiers ouverts

```
if ( close ( fle ) == -1 ) { /* fle: cf. open() */  
    perror ( "close" );  
    exit ( EXIT_FAILURE );  
}
```

# Application : tubes nommés

- Moyen de communication interprocessus
  - Zone mémoire d'échange vue comme un fichier
  - Tube avec **pipe()**
    - Communication entre des processus de même filiation (héritage des descripteurs)
    - Utilisation pour les filtres du shell (**ls | wc**)
  - Tube nommé (FIFO)
    - Nom dans le système de fichier (commande ou fonction **mkfifo**)
    - Communication par tout processus disposant des droits d'accès
    - Ouverture comme tout fichier
    - Point d'entrée pour agir sur des processus
- Caractéristiques
  - Flot d'octets unidirectionnel (pas de notion de message)
  - Les octets sont lus dans l'ordre dans lequel ils ont été écrits
  - Impossibilité d'accès direct : **lseek()** impossible

# Sémantique des FIFOs

- Synchronisation de l'ouverture d'un FIFO
  - Un processus qui veut ouvrir en lecture seule un FIFO qui n'est pas déjà ouvert en écriture est suspendu
  - Un processus qui veut ouvrir en écriture seule un FIFO qui n'est pas déjà ouvert en lecture est suspendu
- Lecture depuis un FIFO vide
  - Si le FIFO est encore ouvert en écriture, le processus lecteur est suspendu jusqu'à la prochaine écriture
  - Sinon (le FIFO a été fermé) le processus lecteur reçoit **EOF** (fin de fichier)
- Écriture
  - Si le FIFO n'est plus ouvert en lecture, une tentative d'écriture « tue » le processus écrivain (sauf s'il intercepte le signal **SIGPIPE**)
  - Une écriture dans un FIFO « plein » suspend le processus écrivain jusqu'à la prochaine lecture (les FIFO ont une taille limitée, 64 Ko sous Linux)
- Écritures « simultanées » dans un FIFO
  - Les écritures sont atomiques
  - Les données écrites sont garanties de ne pas être « mélangées » si leur taille ne dépasse pas **PIPE\_BUF** (4096 octets sous Linux, cf **<linux/limits.h>**)

# mkfifo() : création d'un FIFO

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

```
int mkfifo (const char *chemin, mode_t droits) ;
```

Retourne 0 en cas de succès, ou -1 en cas d'erreur

- **chemin** : chemin d'accès du FIFO à créer
- **droits** : permissions (voir **droits** de **open()** avec **O\_CREAT** : valeur octale)
- **ls -l** du FIFO affichera **p** en première colonne
- On peut aussi utiliser la commande **mkfifo** (**man 1 mkfifo**) pour créer le FIFO (**rm** permet de le supprimer)
- Les appels système **open()**, **read()**, **write()** et **close()** sont utilisés normalement (avec le respect de la sémantique vue précédemment)

```
if ( mkfifo ( "FIFO", 0600) == -1 ) {
    perror ( "mkfifo" );
    exit (EXIT_FAILURE);
}
fd = open ( "FIFO", O_RDONLY);
...
```

```
$ mkfifo -m 0600 FIFO
$ ls -l FIFO
prw----- 1 lalevee lalevee 0 2011-06-10 15:23 FIFO
$ rm FIFO
rm : supprimer FIFO «FIFO» ? y
```

Séance 3 : Entrées-Sorties avancées

# LES SOCKETS TCP/IP

# Les Sockets

- Communications entre deux applications situées sur deux nœuds réseau
  - Si les nœuds sont identiques → communication locale
  - Applicables à tout type de réseau
    - **IPv4** : objet de ce cours, restreint à TCP
    - UNIX : communication locale
    - IPv6, IPX, Appletalk, X25...
  - **socket** = point de communication dans un domaine réseau (fichier spécial pour le noyau)
- La communication est dissymétrique
  - Processus en attente de requêtes : rôle de serveur (passif)
  - Processus à l'initiative de la communication : rôle de client (actif)
- Scénario type
  - Chaque processus crée un socket
  - Le serveur associe son socket à une adresse IP et un numéro de port
  - Le client communique avec le serveur en utilisant cette adresse et ce numéro de port

# Types de socket IP

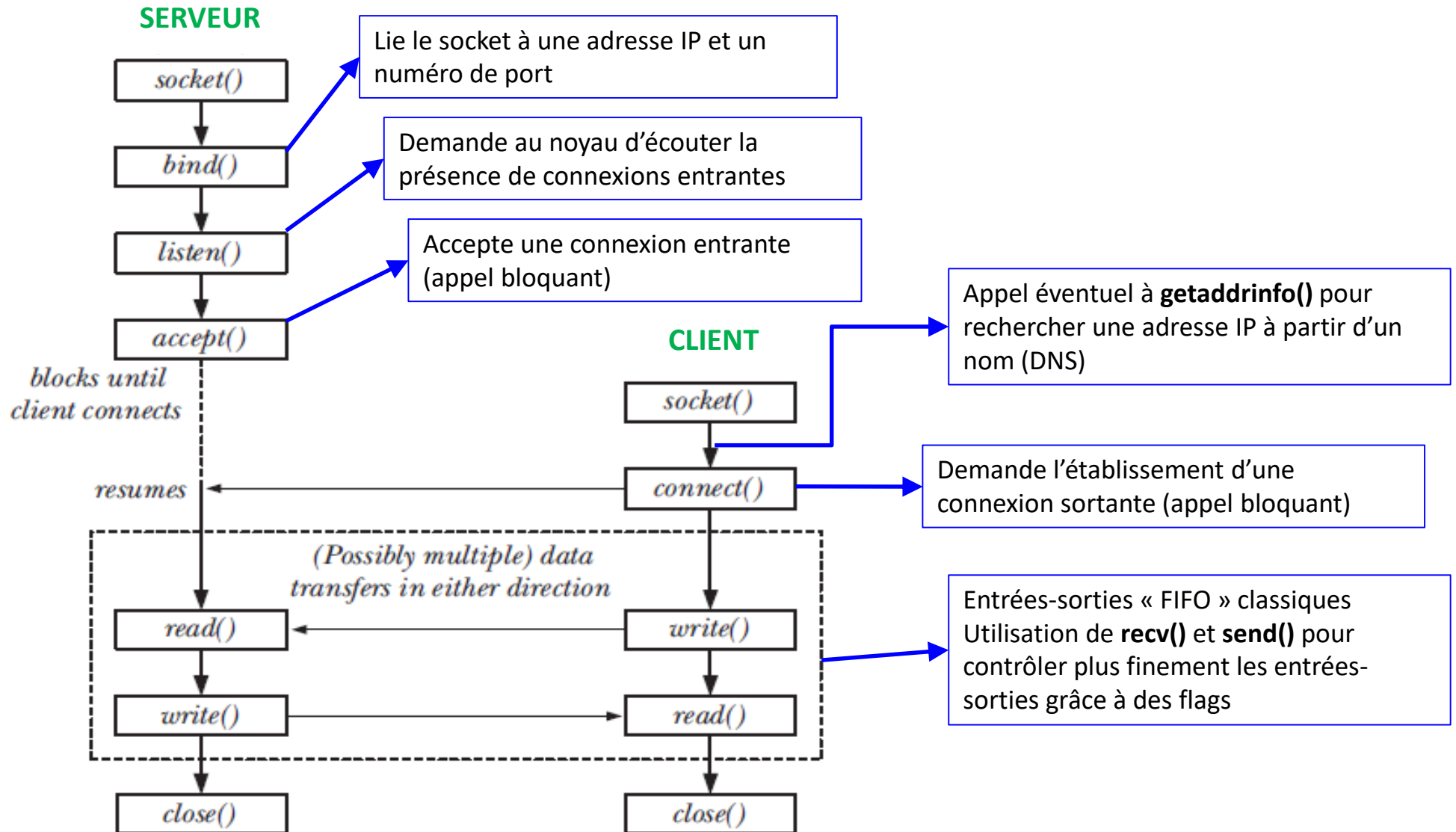
- *Stream socket (SOCK\_STREAM)*
  - Fiable : transmission sûre des données (ordre, contenu...) et gestion des erreurs
  - Flot d'octets bidirectionnels (double FIFO)
  - Mode connecté : transmission après établissement de la connexion (réservation de bout en bout des ressources réseau)
  - Protocole TCP
- *Datagram socket (SOCK\_DGRAM)*
  - Non fiable : ordre non préservé, perte possible, pas de gestion d'erreurs
  - Données échangées sous forme de messages (datagrammes)
  - Mode non connecté
  - Protocole UDP



# Scénario en mode connecté

- |  |   |
|--|---|
| <ul style="list-style-type: none"><li>• Acheter des téléphones</li><li>• Pour pouvoir recevoir des appels<ul style="list-style-type: none"><li>– Disposer d'un numéro de téléphone (numéro de poste)</li><li>– Brancher le téléphone</li><li>– Activation de la ligne par l'opérateur</li></ul></li><li>• Appel téléphonique<ul style="list-style-type: none"><li>– L'appelant interroge l'annuaire pour connaître le numéro de téléphone de l'appelé</li><li>– L'appelant compose le numéro de l'appelé et attend la connexion</li><li>– L'appelé décroche</li></ul></li><li>• Les deux personnes peuvent se parler</li><li>• La communication est rompue dès que l'un des deux raccroche</li></ul> | <ul style="list-style-type: none"><li>• Création de sockets</li><li>• Pour pouvoir recevoir une communication<ul style="list-style-type: none"><li>– Disposer d'une adresse IP et d'un numéro de port</li><li>– Demander au noyau d'associer l'adresse IP et le port au socket ( <b>bind()</b> ) et d'écouter les connexions entrantes ( <b>listen()</b> )</li></ul></li><li>• Demande de connexion<ul style="list-style-type: none"><li>– Le client interroge le DNS pour connaître l'adresse du serveur ( <b>getaddrinfo()</b> )</li><li>– Le client demande l'établissement d'une connexion vers l'adresse IP et le numéro de port de l'application ( <b>connect()</b> )</li><li>– Le serveur accepte la connexion ( <b>accept()</b> )</li></ul></li><li>• Les deux applications peuvent communiquer ( <b>read()</b> / <b>write()</b> )</li><li>• La communication est interrompue dès qu'un des processus ferme son socket ( <b>close()</b> )</li></ul> |
|--|---|

# Scénario complet (mode connecté)



# socket() : création d'un socket

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

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

Retourne un descripteur de fichier, ou -1 en cas d'erreur

- **domain** : indique la famille de protocoles, **AF\_INET** pour TCP/IPv4, **AF\_INET6** pour TCP/IPv6
- **type** : sémantique de communication
  - **SOCK\_DGRAM** pour UDP (non fiable) ← non traité dans le cours
  - **SOCK\_STREAM** pour TCP (équivalent aux FIFOs mais bidirectionnels)
  - **SOCK\_RAW** pour IP (droits administrateur)
- **protocol** : toujours à zéro pour **AF\_INET**
- Retourne un descripteur de fichier à fermer avec **close()**

```
int sock = socket (AF_INET, SOCK_STREAM, 0 ) ;
if ( sock == -1 ) {
    perror ("socket" );
    exit ( EXIT_FAILURE );
}
...
close ( sock );
```

# bind() : association d'une IP à un socket

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

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

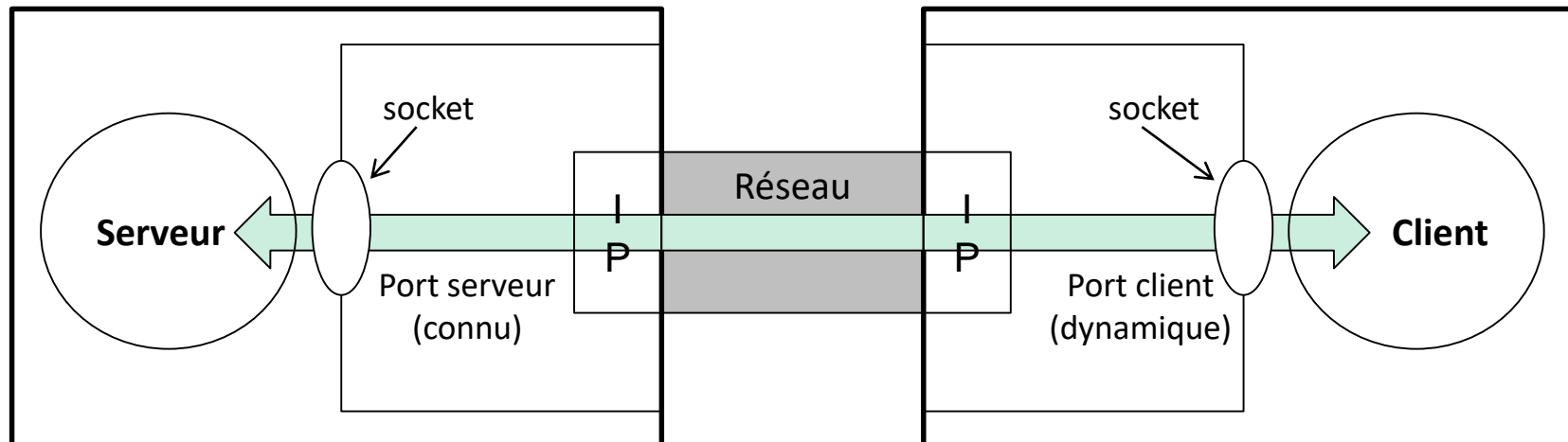
Retourne 0 en cas de succès, ou -1 en cas d'erreur

- **sockfd** : descripteur de fichier retourné par **socket()**
- **addr** : structure contenant l'adresse IP et le numéro de port sur lesquels être à l'écoute
- **addrlen** : longueur en octets de la structure pointée par **addr**

```
sockaddr_in adresse;
... /* cf. transparents suivants */
if ( bind ( sock, (struct sockaddr *) &adresse, sizeof ( adresse ) ) != 0 ) {
    perror ( "bind" );
    exit ( EXIT_FAILURE );
}
```

# Rappels sur les adresses IPv4

- Adresses sur 4 octets (32 bits)
  - Exemple : 192.168.10.10
  - Classe, masque, *subnetting* : transparents pour les applications
  - **INADDR\_ANY** : désigne toutes les adresses IP locales
  - **INADDR\_LOOPBACK** : 127.0.0.1, désigne **localhost**
- Numéro de port sur deux octets (16 bits)
  - De 1 à 1023 : ports « bien connus » (réservés)
  - De 1024 à 49151 : ports enregistrés
  - Au-delà : ports dynamiques (utilisés pour le client)



# Représentation des données

- Représentation des entiers

- Network Order : Internet est *Big Endian* (dans les mots de 16 ou 32 bits les octets sont stockés avec les poids forts dans les adresses basses)

3	2	1	0
---	---	---	---

- Host Order : les PC (x86) sont *Little Endian* (dans les mots de 16 ou 32 bits les octets sont stockés avec les poids faibles dans les adresses basses)

0	1	2	3
---	---	---	---

- Fonctions de conversion pour les adresses IP et numéros de port à transmettre (prototypes dans **<arpa/inet.h>**)

- Avant l'émission : **htons()** ( **htonl()** ) → conversion d'un **short** ( **long** ) de « *host order* » vers « *network order* »
- Après la réception : **ntohs()** ( **ntohl()** ) → conversion d'un **short** ( **long** ) de « *network order* » vers « *host order* »

# Type struct sockaddr\_in

- **sockaddr** est un type générique pour tout type de réseau
- Utilisation de **struct sockaddr\_in** pour les sockets IP (nécessité de *cast* pour **bind()**, **accept()** et **connect()** )

```
/* extrait de <netinet/in.h> */
struct in_addr {
    in_addr_t s_addr; /* entier non signé de 32 bits */
};
struct sockaddr_in {
    sa_family_t    sin_family; /* AF_INET, cf. socket() */
    in_port_t      sin_port;   /* numero de port sur 2 octets (network order) */
    struct in_addr sin_addr;   /* adresse IP (network order) */
    unsigned char  _pad...;    /* padding de correspondance avec sockaddr */
};
```

```
/* exemple d'initialisation */
struct sockaddr_in adresse;
in_addr_t adrIP    = 0XC0A80A02; /* adresse 192.168.10.2 */
adresse.sin_family = AF_INET;
adresse.sin_port   = htons ( 2345 ); /* port (network order) */
adresse.sin_addr.s_addr = htonl ( adrIP ); /* adresse IP (network order) */
adresse.sin_addr.s_addr = INADDR_ANY; /* toutes les interfaces locales */
```

# listen() : écouter des connexions

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

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

Retourne 0 en cas de succès, ou -1 en cas d'erreur

- Cette fonction permet de demander au noyau d'écouter les demandes de connexion (le socket est mis en mode passif, i.e. en mode serveur)
- **sockfd** : descripteur de fichier retourné par **socket()**
- **backlog** : nombre maximal de connexions en attente (taille de la file d'attente)
- La file d'attente contient les informations à conserver pour un prochain **accept()**
- Au-delà, les demandes sont rejetées (*connection refused*)

```
if ( listen ( sock, 20 ) != 0 ) {
    perror ( "listen" );
    exit ( EXIT_FAILURE );
}
```



# accept() : accepter une connexion

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

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

Retourne un descripteur de fichier, ou -1 en cas d'erreur

- Cette fonction extrait la première demande de connexion en attente (ou se bloque sinon) puis crée un socket de connexion entre le client et le serveur
- Analogie téléphone
  - Le socket d'écoute est le standard
  - Le socket de connexion retourné est la liaison vers le correspondant
- **sockfd** : socket d'écoute (après **listen()** )
- **addr** : adresse IP et numéro de port du client connecté (paramètre en sortie)
- **addrlen** : taille de la structure **addr** (paramètre en sortie)

```
sockaddr_in adresse;
socklen_t len;
int sockconn = accept ( sock, (struct sockaddr *) &adresse, &len );
if ( sockconn == -1 ) { perror ( "accept" ); exit ( EXIT_FAILURE ); }
/* adresse.sin_addr.s_addr: IP du client et adresse.sin_port: port de réponse */
/* par la suite : read(sockconn, ...) permettra de lire les octets du client et
write(sockconn, ...) permettra de lui en envoyer */
```

# connect() : se connecter à un serveur

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

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

Retourne 0 en cas de succès, ou -1 en cas d'erreur

- Cette fonction demande la connexion vers le serveur indiqué par son adresse IP → utilisation de **getaddrinfo()** ( fonction DNS )
- **sockfd** : socket de communication
- **addr** : adresse IP et numéro de port du serveur vers lequel se connecter
- **addrlen** : taille de la structure **addr**

```
sockaddr_in adrServ; /* a valoriser ... */
if ( connect ( sock, (struct sockaddr *) &adrServ, sizeof ( adrServ ) ) == -1) {
    perror ( "accept" );
    exit ( EXIT_FAILURE );
}
/* read() et write() sont maintenant possibles */
```

# getaddrinfo() : résolution DNS

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

```
int getaddrinfo (const char *host, const char *service,
                 const struct addrinfo *hints, struct addrinfo **result) ;
```

Retourne 0 en cas de succès, ou un code d'erreur

- **host** : nom de la machine dont on cherche l'adresse IP
- **service** : nom du service dont on cherche le numéro
- **hints** : critères de recherche
- **result** : liste chaînée de structures résultat, allouées par la fonction (nécessite l'appel de **freeaddrinfo()** pour la libérer)
- Les adresses IP et numéro de port dans le *Network Order*
- La gestion du code d'erreur n'est pas gérée par **perror()** mais par **gai\_strerror()** qui retourne le message en clair

```
struct addrinfo *resultat;
if ( (code = getaddrinfo ( "www.emse.fr", "www", &hints, &resultat ) ) != 0 ) {
    fprintf ( stderr, "getaddrinfo: %s\n", gai_strerror ( code ) );
    exit ( EXIT_FAILURE );
}
/* voir transparent suivant pour hints et resultat */
freeaddrinfo ( resultat );
```

# Utilisation de struct addrinfo

```
/* extrait de <netdb.h>
struct addrinfo {
    int ai_flags;
    int ai_family; /* Address family */
    int ai_socktype; /* Type: SOCK_STREAM, SOCK_DGRAM */
    int ai_protocol; /* Socket protocol */
    size_t ai_addrlen; /* Size of structure pointed to by ai_addr */
    char *ai_canonname; /* Canonical name of host */
    struct sockaddr *ai_addr; /* Pointer to socket address structure */
    struct addrinfo *ai_next; /* Next structure in linked list */
};
```

- En **vert**, les champs de la structure pointée par le paramètre **hints** à positionner
  - **ai\_family** : y mettre **AF\_INET**
  - **ai\_socktype** : y mettre **SOCK\_STREAM**
- En **rouge**, les champs utiles de la structure pointée dont l'adresse est stockée dans **result** pour utiliser **bind()**, **accept()** et **connect()**
  - **ai\_addrlen** : paramètre **socklen**
  - **ai\_addr** : paramètre **addr** sans **cast**
- Si plusieurs adresses sont associées au nom, **ai\_next** permet d'en parcourir la liste
- **ai\_canonname** : nom officiel (par exemple, le nom canonique de **www.emse.fr** est **www.wawawa.fr**, cf. commande **host**)

# Exemple complet de requête

```
/* cote serveur : requete DNS pour generer un sockaddr local pour bind() */

struct addrinfo *infos, hints;
memset ( &hints, 0, sizeof (struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if ( getaddrinfo ( "localhost", "www", &hints, &infos ) != 0 ) {
    exit ( EXIT_FAILURE ); }
/* infos->ai_addr.sin_addr.s_addr soit contenir htonl (0X7F000001), ie 127.0.0.1 */
...
if ( bind (sock, infos->ai_addr, infos->ai_addrlen ) == -1 ) {
    perror ( "bind" );
    exit(EXIT_FAILURE);
}
freeaddrinfo ( infos );
```

```
/* cote client : requete DNS pour connect() */

struct addrinfo *infos, hints;
memset ( &hints, 0, sizeof (struct addrinfo));
hints.ai_family = AF_INET;
hints.ai_socktype = SOCK_STREAM;
if ( getaddrinfo ( "www.emse.fr", "www", &hints, &infos ) != 0 ) {
    exit ( EXIT_FAILURE ); }
...
/* infos->ai_addr.sin_addr.s_addr soit contenir htonl (0XC131AEC2),
                                     ie 193.49.174.194 */
/* infos->ai_addr.sin_port doit contenir htons ( 80 ) */
...
if ( connect (sock, infos->ai_addr, infos->ai_addrlen ) == -1 ) {
    perror ( "connect" );
    exit (EXIT_FAILURE );
}
freeaddrinfo ( infos );
```

# Les fonctions **recv()** et **send()**

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

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

Retournent le nombre d'octets reçus ou envoyés, ou -1 en cas d'erreur

- Si **flags** mis à zéro, **recv()** est équivalente à **read()** et **send()** à **write()** (sémantique FIFO)
- **flags** : booléens à positionner avec des « ou logiques » ( | )
  - Flag usuel : **MSG\_DONTWAIT** : opération non bloquante (les fonctions retournent -1 et **errno** vaut **EAGAIN** ou **EWOULDBLOCK**) (cf. manuel)

```
char texte[20];
int lus = recv ( sock, texte, 20, MSG_DONTWAIT );
if ( lus == -1 ) {
    if ( errno == EAGAIN ) { printf ( "pas de donnees a recevoir\n" ); ... }
    else { perror ( "accept" ); exit ( EXIT_FAILURE ); }
}
else { int ecrit = send ( sock, texte, 20, 0 ); ... }
```

Séance 4 : multiprogrammation « lourde »

# PROCESSUS

# Programmes et Processus

- Programme

- Source : séquence d'opérations et des données exprimée en « langage source » pour effectuer un travail informatique
- Exécutable : séquence d'instructions machine, de données et d'informations traduite du programme source par un compilateur, stockées dans un fichier

- Processus

- Instance de programme exécutable chargé en mémoire et géré par le système (attributs)
- Point de vue utilisateur : réalisation du travail informatique souhaité (double clic ou commande)
- Point de vue noyau : entités se partageant les ressources de l'ordinateur (mémoire, CPU, E/S...), que le noyau alloue et retire



# Applications multiprocessus

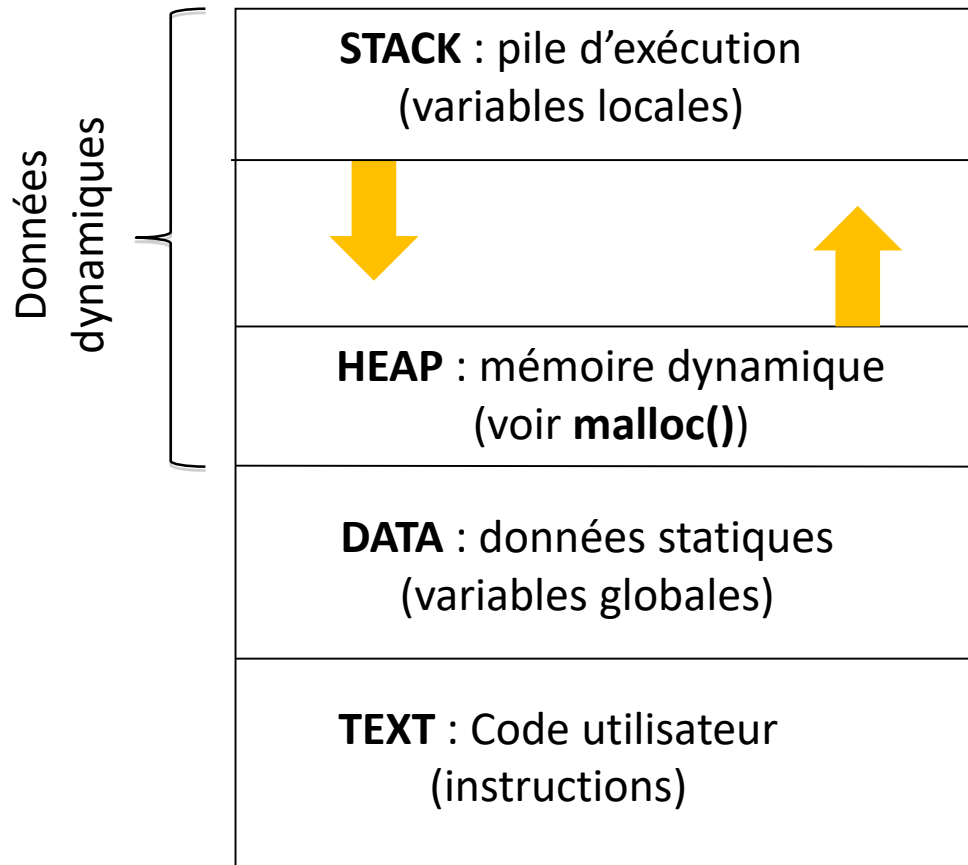
- Pourquoi ?

- Exécution asynchrone de requêtes bloquantes
- Faire plusieurs choses « en même temps »
- Ne pas attendre la fin d'une longue commande
- Ne pas attendre la fin d'une entrée-sortie
- Permettre l'accès à plusieurs utilisateurs

- Comment ?

- Sous le shell : ajouter **&** à la commande (ou la séquence **CTRL-Z** suivi de **bg**)
- Sous GNOME : double-cliquer sur les icones
- Par programme : appels système !

# Processus en mémoire

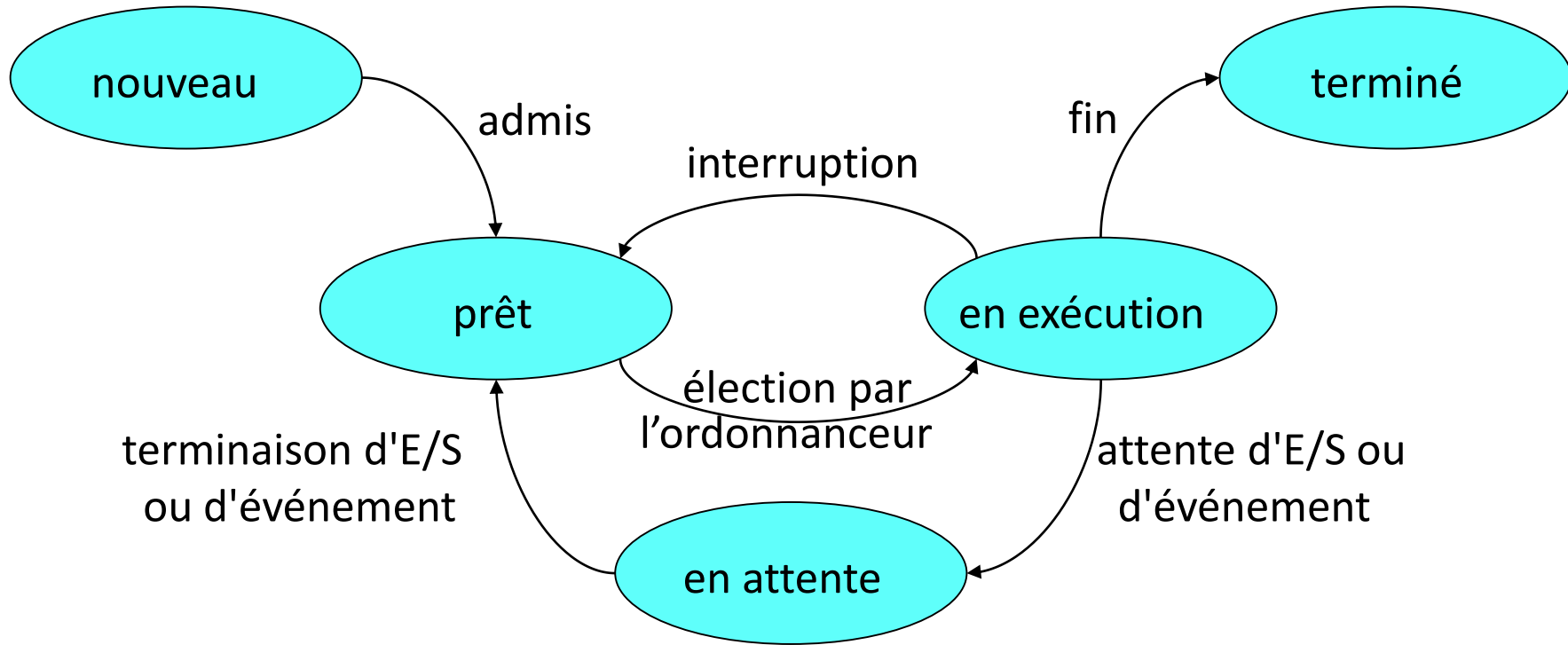


Contexte (*layout*) mémoire  
d'un processus  
(espace utilisateur)

Sauvegarde registres (compteur ordinal..)
Limites mémoire État des fichiers ouverts
<b>PID</b> , PID du père, PID des fils
État du processus (prêt, attente...)
.....

Données de gestion  
d'un processus stockées  
dans la table des processus  
(espace noyau)

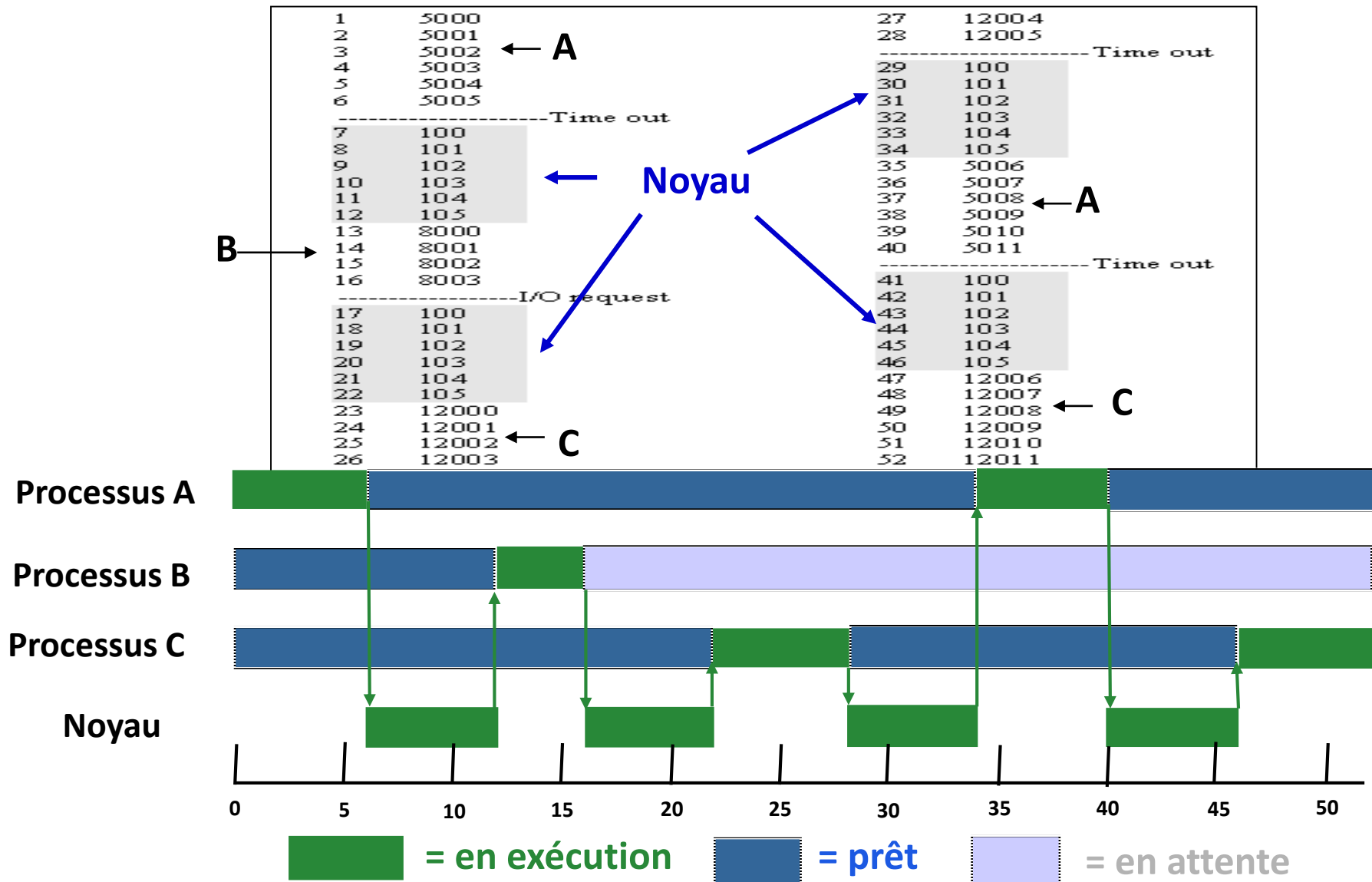
# États d'un processus



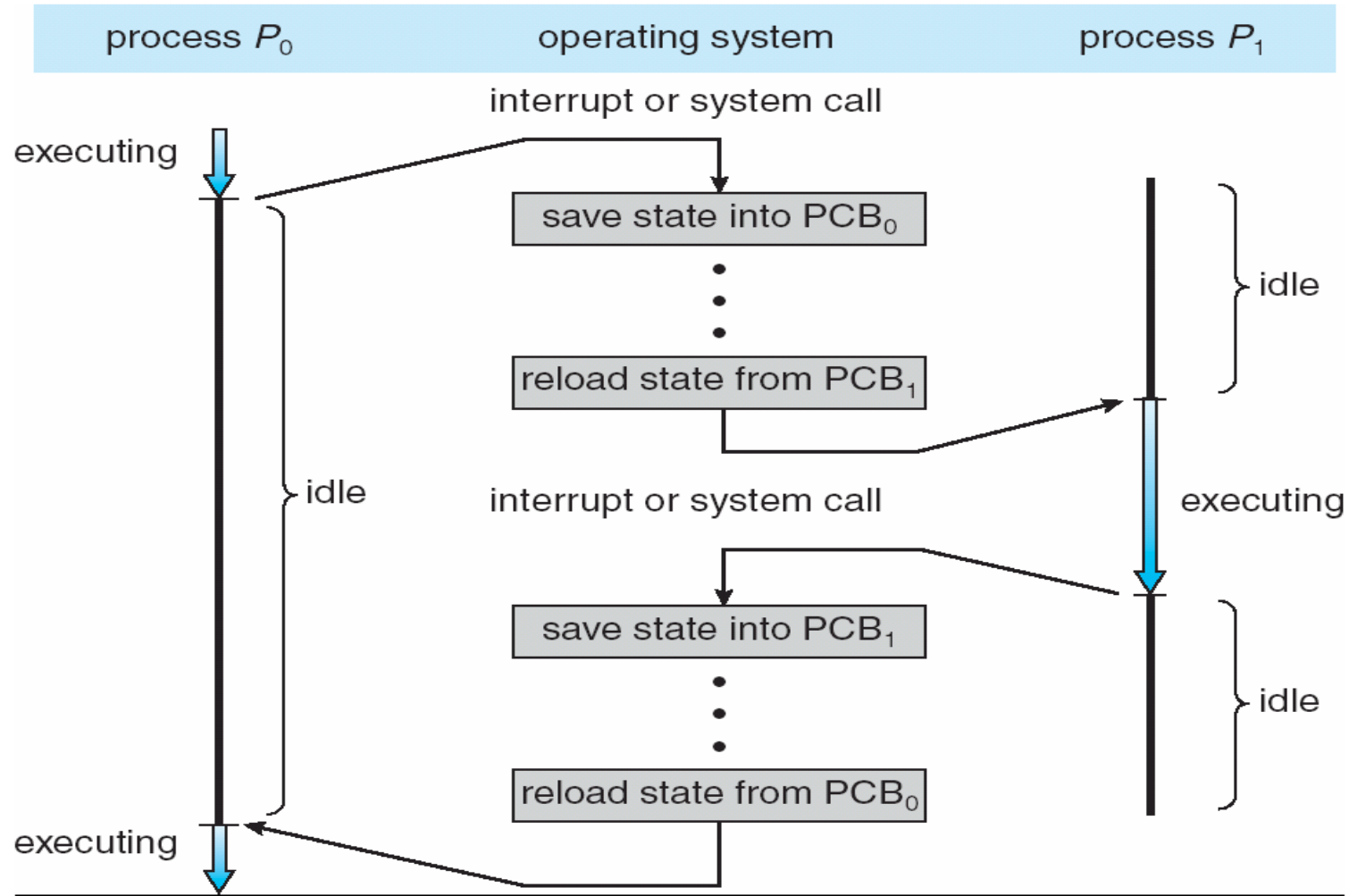
## À un instant donné :

- un processus au plus est en exécution par le processeur (si 1 seul processeur)
- plusieurs processus peuvent être nouveaux, prêts, en attente ou terminés

# Multiprogrammation

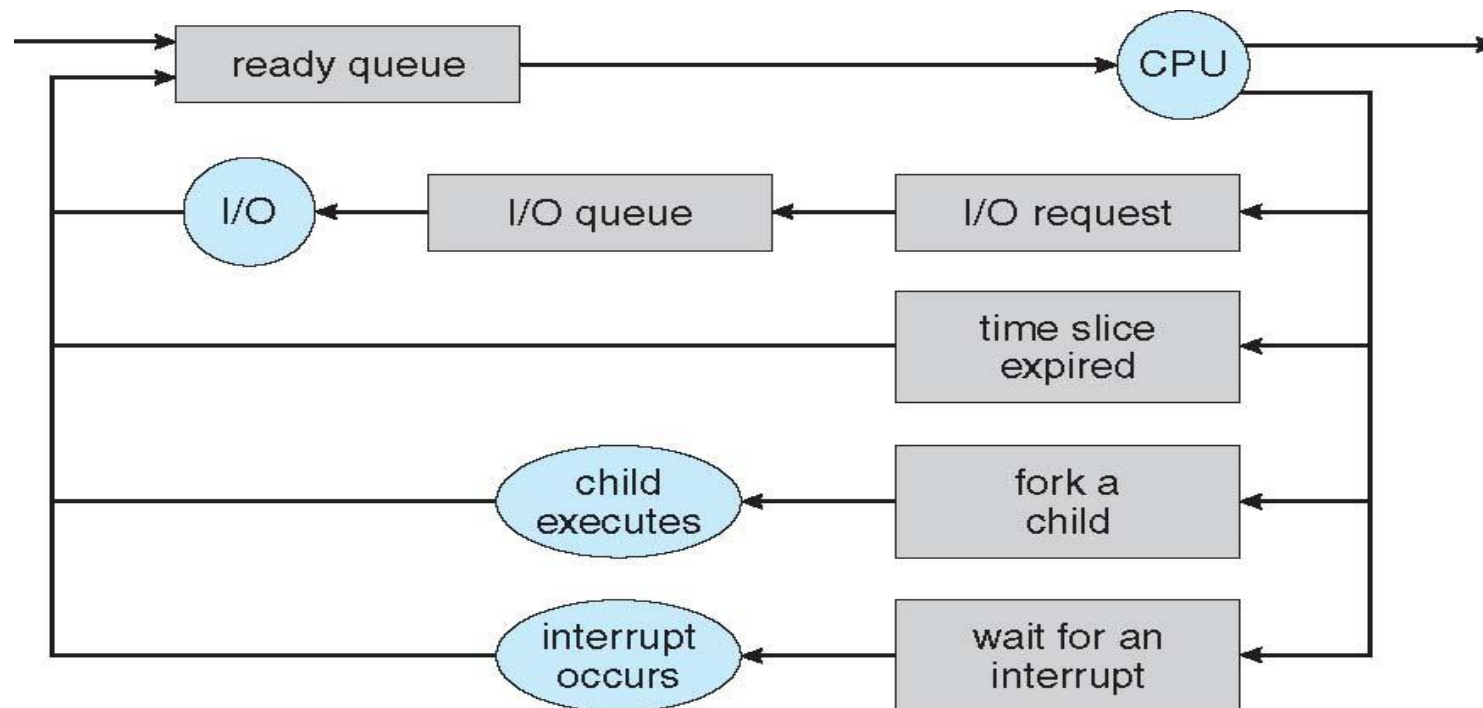


# Changement de contexte



# Ordonnancement (*scheduling*)

- Les processus sont placés dans des files d'attente
  - Prêts : en attente du CPU
  - Attente entrées-sorties (disques, réseaux, résolution défaut de page, etc.)
- Politiques d'ordonnancement
  - Gestion des priorités
  - Type d'utilisation du système : calculs, interactivité, multimédia...



# À savoir sur les processus Unix

- Tout processus Unix est unique → identifiant = **PID**
- Processus privilégiés
  - Ils appartiennent au « super-utilisateur » (UID = 0)
  - Permissions d'accès « court-circuitées »
- *Daemons*
  - Processus de longue durée (la vie du système)
  - Pas de terminal associé (exécution en *background*)
- Processus **init**
  - Processus privilégié créé au démarrage du système (**/sbin/init**)
  - « Ancêtre » de tous les processus : soit créés par lui ou par un de ses descendants
  - « Père » adoptif des processus orphelins
  - Son **PID** vaut **1**
  - Sa terminaison = arrêt (**shutdown**) du système
- Pour mémoire :
  - **ps** : commande d'affichage des processus
  - **pstree** : commande d'affichage de l'arborescence des processus
  - **Moniteur système** : affichage graphique sous GNOME
  - Système de fichiers **/proc** à parcourir

# Informations sur un processus

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

```
pid_t getpid ( void );
```

```
pid_t getppid ( void );
```

Retourne le numéro de **PID** du processus ou de son père

- Appels toujours réussis
- Les **PID** sont alloués séquentiellement (circulaire)
- Seul le **PID 1** est réservé pour **init**
- Exemple

```
pid_t monpid = getpid ();  
pid_t papa = getppid ();  
printf ("Mon numero est %d et celui de papa est %d\n", monpid, papa);
```



# Création de processus

- Un processus sous Unix est une « enveloppe » qui héberge un programme → 2 appels système
  - Création de l'« enveloppe » : **fork()**
  - « Hébergement » d'un programme : **execve()**
- Création par duplication (clonage)
  - Le processus « père » demande la création avec **fork()**
  - Le processus « fils » est le clone du processus « père » par recopie ou partage (le segment **TEXT** est partagé)
  - Identiques, hormis quelques données de gestion, comme le **PID**
  - Chaque processus est ensuite indépendant (exécution asynchrone)
- Exécution d'un programme par écrasement
  - Le nouveau programme exécutable est chargé en mémoire avec **execve()**
  - L'espace utilisateur du processus est écrasé
  - Les données de gestion du processus sont mises à jour (initialisées, adaptées ou préservées)

# Terminaison de processus

- Terminaison normale

1. Exécution de **return** dans la fonction **main()**  
→ appel implicite à **exit()** ajouté par le compilateur
2. Fin (accolade fermante « } ») de la fonction **main()**  
→ appel implicite à **exit()** ajouté par le compilateur
3. Appel explicite à la fonction **exit()** ou à l'appel système **\_exit()**

- Terminaison anormale

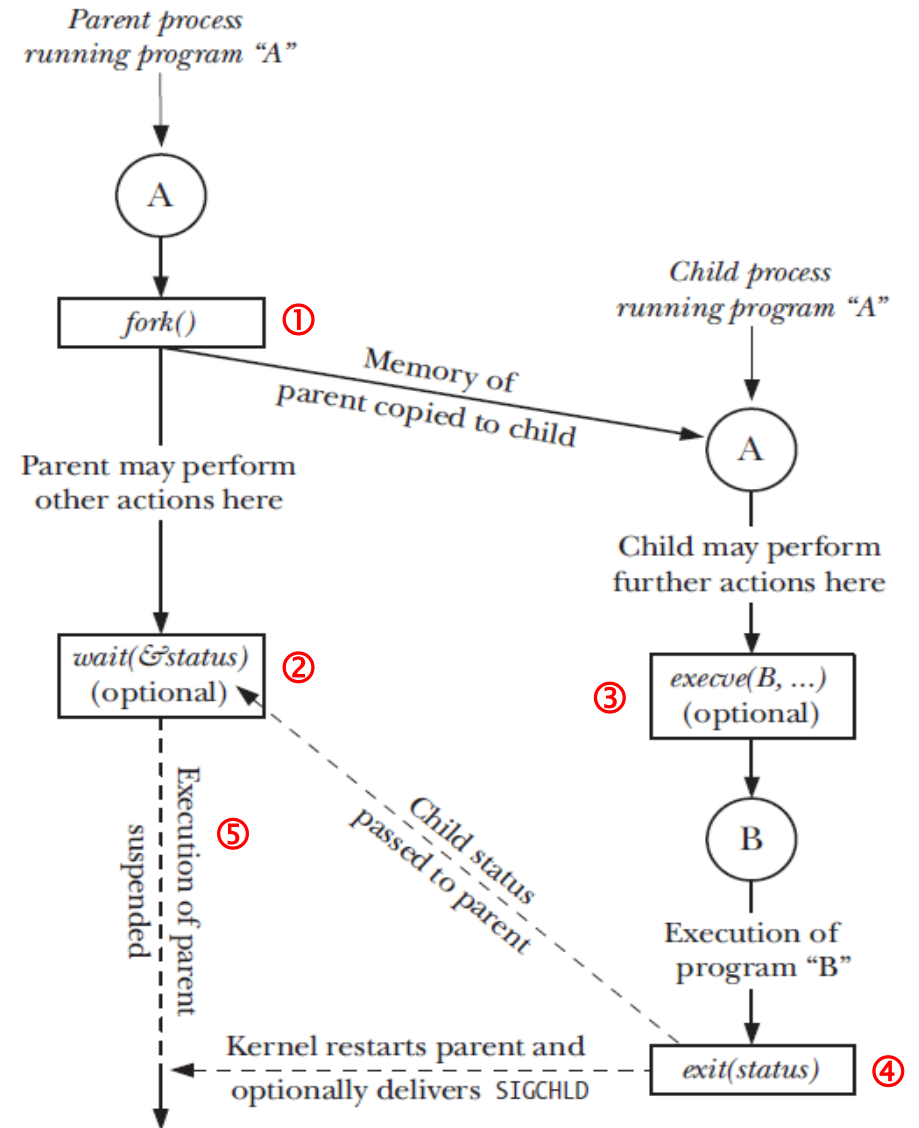
- Appel de la fonction **abort()**
- Réception d'un **signal** : exception, événement, demande explicite

- Terminaison

- Libération des ressources
- Fermeture des fichiers
- Informations de terminaison disponibles pour le processus « père » grâce à **wait()**
- Par convention : le code de retour **0** signifie pas d'erreur (réponse à la question « y a-t-il eu des erreurs ? » avec **0** signifiant « non » en C)

# Schéma d'exécution « classique »

1. Duplication avec **fork()**
2. Le « père » attend la fin du fils avec **wait()**
3. Le « fils » charge un nouveau programme avec **execve()** et l'exécute
4. Le fils se termine avec **exit()**
5. Le « père » récupère le code de terminaison (*status*) du « fils » et continue son exécution



# fork() : clonage d'un processus

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

```
pid_t fork ( void );
```

Retourne (père): le **PID** du fils créé ou **-1** en cas d'erreur; (fils): **0**

- Après l'exécution réussie de **fork()**, deux processus indépendants existent
  - Le « père » qui reçoit la valeur du **PID** du fils
  - Le « fils » (copie du « père ») qui reçoit la valeur **0**
- Ils exécutent le même programme : leurs exécutions continuent avec l'instruction qui suit **fork()**
- Exemple

```
pid_t fils = fork ();  
if ( fils == -1 ) { perror ("fork"); exit (EXIT_FAILURE); }  
/* le père et le fils */  
if ( fils == 0 ) { /* uniquement le fils */ }  
else { /* uniquement le père */ }  
/* le père et le fils */
```

# fork() : exemple commenté

```
int main ( void ) {  
    pid_t pid;  
    pid = fork ();  
    assert ( pid != -1 );  
    if ( pid == 0 ) {  
        printf ("je suis le fils %d (père %d)\n",  
                getpid (), getppid () );  
    } else {  
        printf ("je suis le père %d (fils %d)\n",  
                getpid (), pid );  
        wait(NULL);  
    }  
    return 0;  
}
```

## Après **fork()**

- Allocation d'une entrée dans la table des processus et d'un **PID** au nouveau processus
- Duplication du contexte du processus « père » (données, pile...)
- Retour du **PID** du processus « fils » à son « père » et **0** au processus « fils »

Appel implicite à **exit()** par les deux processus

```
$ ./prog  
je suis le père 29858 (fils 29859)  
je suis le fils 29859 (père 29858)
```

# execve() : recouvrement d'un processus

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

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

Ne retourne jamais si succès, ou -1 en cas d'erreur

- **chemin** : accès au programme exécutable
- **argv** : les paramètres du programme, cf. fonction **main()**
- **envp**: les données d'environnement du programme, cf. fonction **main()**
- Exemple

```
extern char **environ;  
int main ( int argc, char *argv[] ) {  
    int retour = execve ("/bin/ls", argv, environ );  
    /* ici, obligatoirement une erreur (retour vaut -1) */  
    perror ("execve");  
    exit (EXIT_FAILURE);  
}
```

# Famille de fonctions **exec..()**

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

```
int execle (const char *chemin, const char *par, ... , char *const envp[] );
```

```
int execlp (const char *nom, const char *par, ... );
```

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

```
int execv (const char *chemin, char *const argv[] );
```

```
int execl (const char *chemin, const char *arg, ... );
```

Ne retournent jamais si succès, ou -1 en cas d'erreur

- **chemin** : accès au programme exécutable
- **par, ...** : liste des paramètres du programme terminée par **NULL**
- **envp**: les données d'environnement du programme
- **nom** : nom du fichier programme exécutable
- **argv** : les paramètres du programme
- Versions
  - **l** : arguments fournis sous forme de liste
  - **v** : arguments fournis sous forme de tableau
  - **p** : recherche du fichier exécutable avec la variable d'environnement **PATH**
  - **e** : données d'environnement fournies (sous forme de tableau)

# Exemple avec **execv()**

```
/* calcul.c */
int main (void) {
    int pid, status;
    char *param [2];
    pid = fork ();
    if (pid == 0) {
        printf ("avant: je suis le fils (pid=%d)\n",
                getpid () );
        param [0] = "carre";
        param [1] = NULL;
        execv ("carre", param);
        perror("execv");
        exit (EXIT_FAILURE);
    }
    printf ("je suis le pere (pid=%d)\n", getpid());
    wait (&status);
    printf("fin du pere (status = %d)\n", status);
    exit(EXIT_SUCCESS);
}
```

```
/* carre.c*/
int main(int argc, char *argv[]) {
    int x;
    printf ("apres: je suis le fils (pid=%d)\n",
            getpid());
    printf ("pid pere = %d\n",getppid() );
    printf ("x = ");
    scanf ("%d", &x);
    x = x * x;
    printf (("carre = %d\n", x);
    exit(EXIT_SUCCESS);
}
```

```
$ ./calcul
je suis le père (pid=30313)
avant: je suis le fils (pid=30314)
apres: je suis le fils (pid=30314)
pid pere = 30313
x = 21
carre = 441
fin du pere (status = 0)
```



# **\_exit()** : terminaison d'un processus

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

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

Ne retourne jamais

- **status** : code de terminaison du processus
  - Une valeur nulle indique OK
  - Récupérable avec l'appel système **wait()**
  - Récupérable sous le shell avec la variable **\$?**
- **Attention**
  - Les fichiers ouverts ne sont pas fermés (risque de perte de données)
  - Utiliser de préférence la fonction **exit()**

# exit() : terminaison d'un processus

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

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

Ne retourne jamais

- **status** : code de terminaison du processus
  - Même signification que pour **\_exit()**
- Différences avec **\_exit()**
  - Les gestionnaires de terminaison éventuels sont appelés en ordre inverse de leur ajout avec **atexit()**
    - Voir cette fonction (**man atexit**)
  - Les fichiers ouverts sont fermés
  - Les ressources sont libérées (segments mémoire, verrous...)
  - L'appel système **\_exit()** est ensuite invoqué

# wait(), waitpid() : attente de la fin d'un fils

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

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

Retourne le PID du fils terminé, ou -1 en cas d'erreur

- **wait()** attend (appel bloquant) la terminaison d'un fils de l'appelant
- **pid** : -1 → tout fils, > 0 → fils indiqué
- **status** : code de terminaison de ce processus fils (si **status** != **NULL**)
- **options** : si **WNOHANG** alors **waitpid()** n'est pas bloquant (retourne 0 si pas de fils terminé)
- **wait()** et **waitpid()** retournent le **PID** du fils terminé
- Si pas de fils, alors **wait()** retourne -1 et positionne errno à **ECHILD**

```
/* attente de la fin de tous les fils d'un processus */
pid_t pid_fils;
while ((pid_fils = wait (NULL)) != -1) continue;
if ( errno != ECHILD ) {
    perror ("wait");
    exit (EXIT_FAILURE);
}
/* wait(&status) equivalent a waitpid(-1, &status, 0) */
```

# Compléments sur le code de retour

- Pour savoir si un fils s'est terminé normalement (avec `_exit()` ) et non par une exception
  - Utiliser la macro **WIFEXITED (status)**
- Pour retrouver le code de retour de manière portable
  - Utiliser la macro **WEXITSTATUS (status)**

```
/* affichage du code de fin d'un processus fils */
pid_t pid_fils;
int status;
pid_fils = wait (&status);
if (pid_fils == -1) {
    perror ("wait");
    exit (EXIT_FAILURE);
}
if ( WIFEXITED (status) ) { /* terminaison normale */
    printf ("code du fils %d = %d\n", pid_fils, WEXITSTATUS (status));
}
```

# Synchronisation « père » / « fils »

- Cas normal (schéma d'exécution classique)
  - Le « fils » se termine par **exit()** ou une exception
  - Le « père » récupère le code par **wait()**
- Que se passe-t-il si le « père » se termine avant ses « fils » ?
  - Ses « fils » deviennent orphelins
  - Ils sont adoptés par **init** (**getppid()** retournera **1**)
- Que se passe-t-il si le « père » (vivant) ne récupère pas les codes de ses « fils » (terminés) avec **wait()** ?
  - Le système libère les ressources des « fils » mais conserve leurs informations de gestion
  - Les fils deviennent alors zombies
  - Quand le « père » se terminera (ou sera arrêté), les zombies seront adoptés par **init** qui les supprimera définitivement
  - Si le « père » ne s'arrête jamais, consommation de ressources pouvant pénaliser le bon fonctionnement du système ➔ **situation à éviter**

Séance 5 : Multiprogrammation « légère »

# LES THREADS

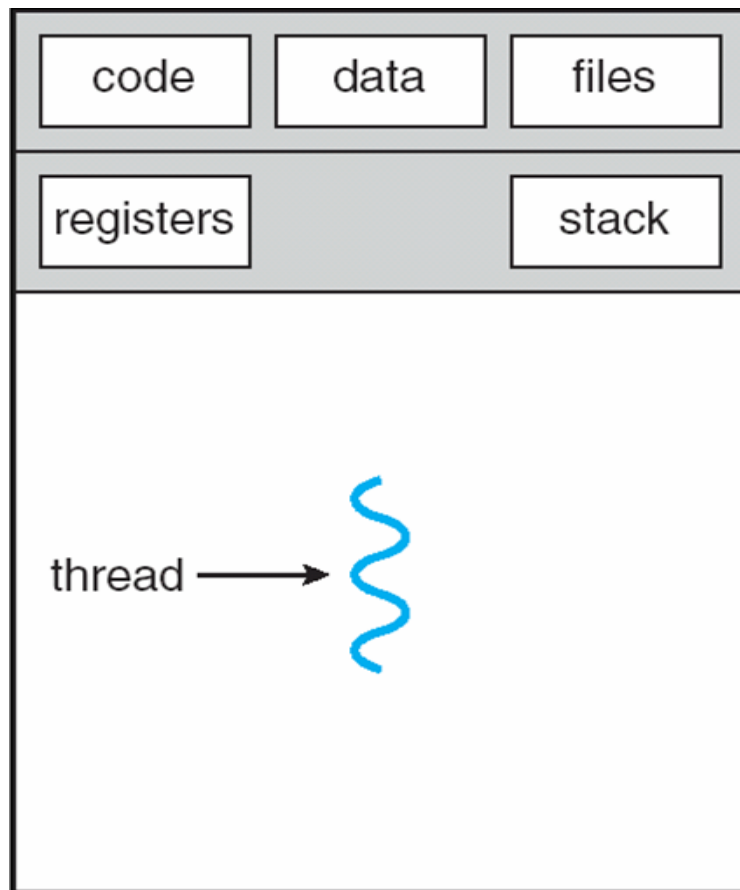
# Application multitâche : bilan

- Utilisation des processus
  - Coûts d'exécution important, si grand nombre d'invocations
  - Mécanismes complexes de communication
- Solution
  - Séparer le processus de son fil d'exécution (*thread*)
  - Permettre plusieurs threads par processus
  - Pas de copie mémoire lors de la création d'un thread
- Comparatif du nombre d'invocations par seconde et le temps sous Linux
  - Nombre de créations de processus/thread par seconde (fréquence) et entre parenthèses, la durée d'invocation en  $\mu s$
  - Colonnes : taille des processus
  - **clone()** : appel système de création de thread sous Linux

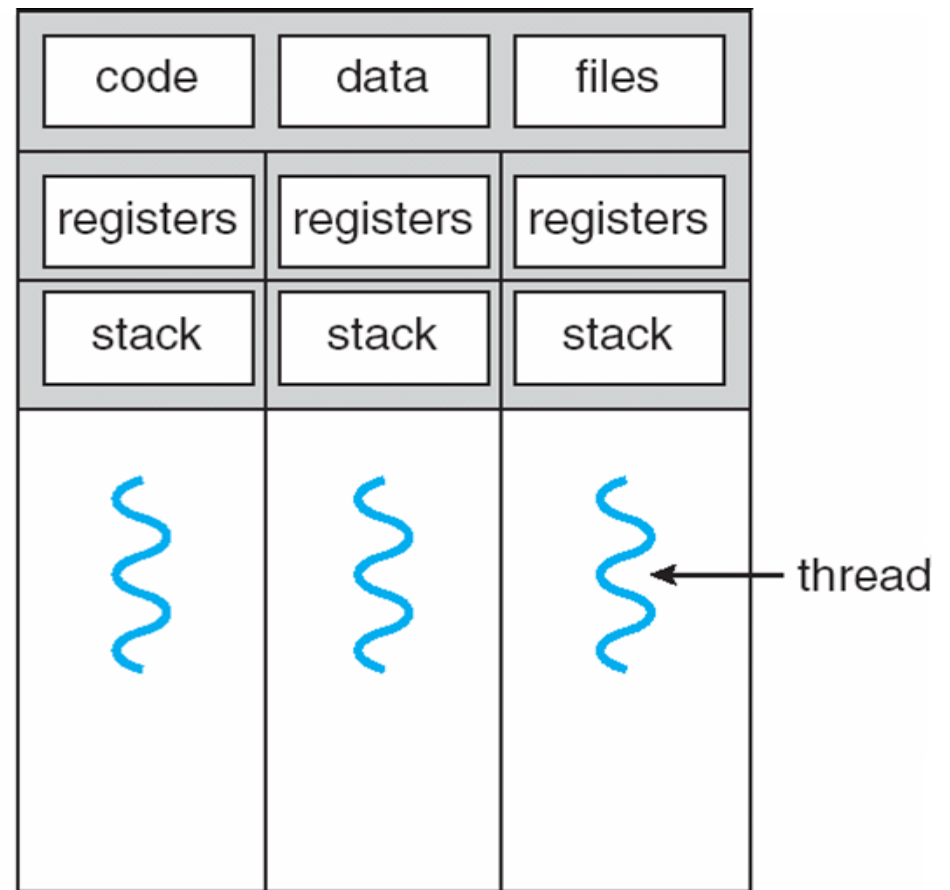
Mécanisme	2 Mo	3 Mo	12 Mo
fork()	4 544 (220 $\mu s$ )	4 135 (241 $\mu s$ )	1 276 (783 $\mu s$ )
clone()	34 533 (29 $\mu s$ )	34 217 (29 $\mu s$ )	34688 (28 $\mu s$ )

Source : LPI, tableau page 610

# Principes des threads



single-threaded process



multithreaded process



# Données partagées entre threads

- Données communes

- Section **DATA**
- PID du processus et de son père
- Terminal associé (de contrôle)
- Descripteurs de fichiers ouverts
- Disposition des signaux
- Informations SGF : répertoire courant, racine...
- Ressources consommées : temps CPU, E/S...
- Priorité du processus

- Données spécifiques

- Identifiant du thread (**TID**)
- Section **STACK**
- La variable **errno**
- Masque pour les signaux

# Programmation threads

- API POSIX : **Pthreads**
  - Spécifications indépendantes du système (logiciel / matériel)
  - Implémentation sous Linux : NTPL
  - Types de données standards : **pthread\_t...**
  - Mécanismes de gestion de la concurrence : Sémaphores, Variables conditionnelles
- Programmation en C
  - La fonction **main()** devient le thread **main**
  - Fichier d'inclusion : **<pthread.h>**
  - Ajout paramètre **-pthread** à la compilation avec **gcc**
  - Valeurs de retour
    - **0** en cas de succès ou valeur positive en cas d'échec (même valeur que dans **errno**)
- Utilisation de fonctions *thread-safe*
  - Invocation sûre par plusieurs threads en « même temps »
  - Problèmes usuels : accès aux mêmes données, écrasement par appels successifs...
    - ➔ mécanismes de gestion de la concurrence à mettre en place
- Consulter le manuel : **man pthreads**

# pthread\_create() : création d'un thread

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

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

- Le nouveau thread commence son exécution avec l'appel de la fonction **start(arg)** ; la valeur de retour de la fonction **start()** pourra être récupérée par un autre thread ( cf. **pthread\_join()** )
- **thread** : identifiant du thread créé (ne pas présumer du type, opaque)
- **attr** : attributs pour la création ( cf. **pthread\_attr\_init()** ), valeurs par défaut si **NULL**
- Exemple

```
void * maFonction ( void * val) {  
    char * chaine = (char *) val;  
    printf ("chaine transmise: %s\n", chaine);    ...  
}  
...  
pthread_t idThread;  
int ret = pthread_create ( &idThread, NULL, maFonction, "Bonjour a tous");  
if (ret != 0) { perror ("pthread_create"); exit (EXIT_FAILURE); }
```

# Identifiants d'un thread

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

```
pthread_t pthread_self (void);  
int pthread_equal (pthread_t t1, pthread_t t2);
```

- **pthread\_self()** retourne l'identifiant du thread appelant
- **pthread\_equal()** teste si les deux identifiants **t1** et **t2** sont égaux (retourne **0** s'ils sont différents, **!= 0** sinon)  
→ à utiliser pour être portable
- Exemple

```
pthread_t idThread;  
...  
if ( pthread_equal ( idThread, pthread_self () ) != 0 ) {  
    printf ("ils sont identiques\n");  
}
```

# pthread\_join() : attente de la fin d'un thread

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

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

- **thread** : le thread appelant attend la terminaison (appel bloquant) du thread indiqué dans **thread**
- **retval** : code (ou valeur) de retour du thread terminé
- Attention : les threads terminés non détachés ( cf. **pthread\_detach()** ) deviennent zombies tant qu'un **pthread\_join()** n'a pas été fait
- Différences avec **wait()**
  - Pas de notion d'arborescence : tout thread peut attendre la fin d'un thread quelconque (la connaissance de son **TID** suffit)
  - Nécessairement un thread identifié et bloquant
- Exemple

```
int * codeRetour;  
pthread_t idThread;  
...  
if ( pthread_join ( idThread, (void **) &codeRetour) != 0) { perror("join"); exit(1); }  
printf ( "code = %d\n", *codeRetour ) ;
```

# pthread\_detach() : détachement d'un thread

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

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

- **thread** : thread à détacher par le thread appelant
- Caractéristiques d'un thread détaché
  - Lors de sa terminaison, ses ressources sont automatiquement libérées
  - Il n'est plus joignable avec **pthread\_join()**
  - Il est automatiquement arrêté lors du **pthread\_exit()** du dernier thread joignable
- Exemple

```
/* detachement du thread */  
if ( pthread_detach ( pthread_self () ) != 0 ) {  
    perror ("detach");  
    exit (EXIT_FAILURE);  
}
```

# pthread\_exit() : terminaison d'un thread

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

```
void pthread_exit (void *retour);
```

- Le thread appelant est terminé : libération des ressources (on peut terminer aussi à la fin ou par un **return** de la fonction **start()**)
- Le dernier thread qui appelle **pthread\_exit()** termine le processus (pas nécessairement le thread **main**)
- Attention : appeler **exit()** termine tous les threads
- **retour** : adresse du code de terminaison du thread (à ne pas placer dans la pile d'exécution)
- Exemple

```
static int codeRetour; /* variable locale statique (attention a la concurrence) */  
codeRetour = 0;  
pthread_exit ( &codeRetour);
```

# pthread\_attr\_\*() : gestion des attributs

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

```
int pthread_attr_init ( pthread_attr_t *attr );  
int pthread_attr_getNAME ( pthread_attr_t *attr , PARAMS );  
int pthread_attr_setNAME ( pthread_attr_t *attr , PARAMS);  
int pthread_attr_destroy ( pthread_attr_t *attr );
```

- **pthread\_attr\_init()** initialise **attr** (éventuelle allocation mémoire)
- **pthread\_attr\_destroy()** libère les ressources allouées
- Les fonctions **pthread\_attr\_set** positionnent un attribut et les fonctions **pthread\_attr\_get** permettent de le récupérer (consulter le manuel)
- Exemple : **pthread\_attr\_setdetachstate()** positionne le mode détaché (**PTHREAD\_CREATE\_DETACHED**) ou non (**PTHREAD\_CREATE\_JOINABLE**)

```
pthread_attr_t attr;  
if ( pthread_attr_init ( &attr ) != 0 ) { perror("attr_init"); ... }  
if ( pthread_attr_setdetachstate (&attr, PTHREAD_CREATE_DETACHED) != 0 ) { ... }  
if ( pthread_create ( &idThread, &attr, maFonction, "Bonjour a tous" ) != 0 ) { ... }  
if ( pthread_attr_destroy ( &attr ) != 0 ) { ... }
```



# Application multiprocessus ou multithreads ?

- Quels sont les facteurs qui peuvent guider le choix ?
- Avantages de l'approche « multithread »
  - Le partage des données entre threads est simple (plus complexe pour les processus)
  - La création de threads est plus performante
  - Le changement de contexte entre threads est plus rapide
- Inconvénients de l'approche « multithread »
  - Les fonctions utilisées doivent être *thread-safe* (les processus ne sont pas concernés)
  - Le partage de mémoire peut engendrer des problèmes difficiles à déboguer
  - L'espace mémoire du processus (limité) est partagé par l'ensemble des threads
- Autres points pour le choix de l'approche
  - Dans l'approche multithread, tous les threads exécutent le même programme (différentes fonctions chargées en même temps que le programme)
  - Le partage d'informations de gestion peut engendrer des difficultés de programmation (descripteurs de fichiers, répertoire courant...), mais cela peut être un avantage...

Séance 6 : architecture clients/serveur

# **SERVEURS MULTIPROGRAMMES**

# Architecture client-serveur

- Une application repose sur une architecture client-serveur si elle se décompose en deux parties distinctes :
  - Une partie « cliente » qui soumet des requêtes à une partie serveur
  - Une partie serveur qui fournit le service en réponse
- Ces parties sont des tâches (processus ou thread), placées éventuellement sur des machines différentes
  - Mécanismes de communication (en général synchrones)
    - Mémoire partagée, tubes, sockets, etc.
  - Mécanismes de synchronisation
    - Sémantiques de communication, événements, primitives du noyau (mutex, sémaphores...)
- **Nota bene:** cette définition diffère de la définition classique, qui concerne les applications en réseau (applications distribuées, peer-to-peer, etc.), associées à Internet. Celle-ci-dessus est plus générale, et s'applique également aux systèmes d'exploitation et aux services qu'ils rendent aux programmes utilisateurs.

# Pourquoi cette architecture ?

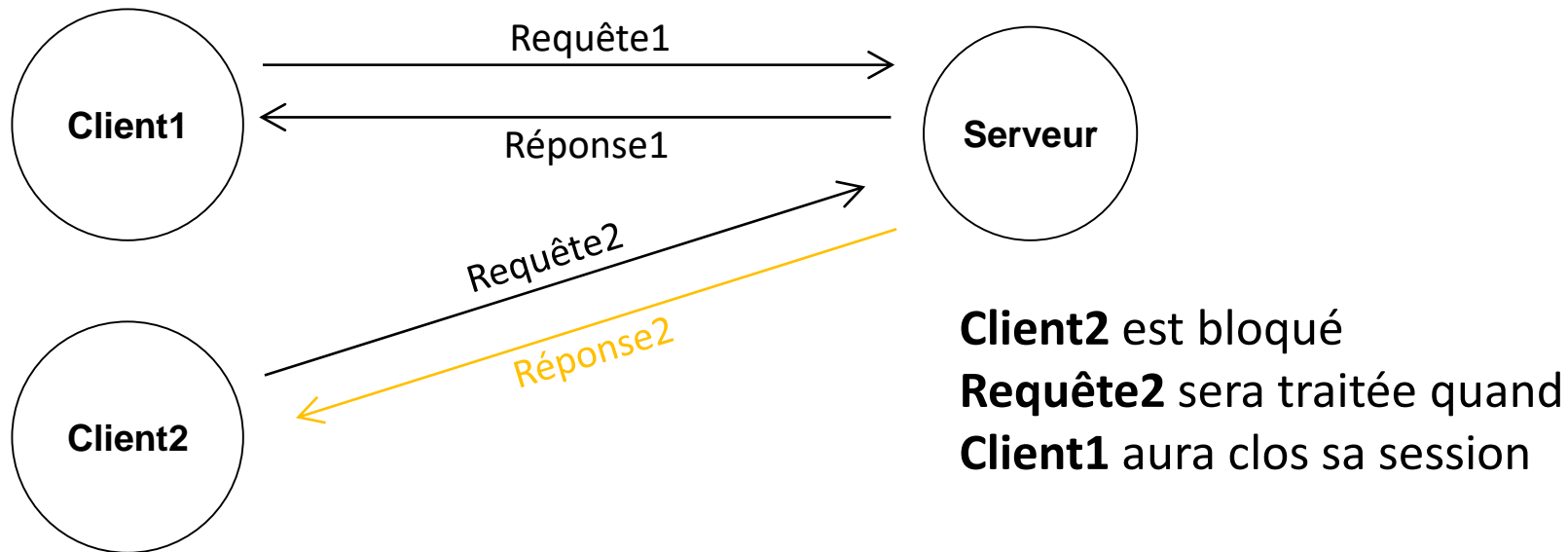
- La partie « cliente » gère l'interface avec l'utilisateur
  - Soumission de requêtes sécurisée et simplifiée
  - Séparation des rôles
- Le serveur fournit les ressources
  - Accès aux données, éventuellement partagées
  - Accès à la puissance de calcul et de traitement
  - Accès aux composants matériels
- Le client et le serveur peuvent :
  - Être asynchrones
  - Avoir des bandes passantes différentes
  - Être faiblement couplés
- Inconvénients
  - Tolérance aux fautes
  - Mécanisme asymétrique (maître/esclave)

# Principes de fonctionnement

- Le serveur est préalablement lancé
  - Il s'exécute en *background* (voir daemon)
  - Il crée un canal de communication
  - Et se met en attente de requêtes
- Le client se connecte au serveur
  - Ouverture du canal de communication
  - Création d'une session (par exemple, gestion des droits)
  - Soumission de requêtes en mode
    - Question / réponse (1-1)
    - *Publish / subscribe* (1-N)
    - Événement (1-0 ou 0-1)

# Modèle de base

- La tâche serveur traite les sessions des clients en séquence



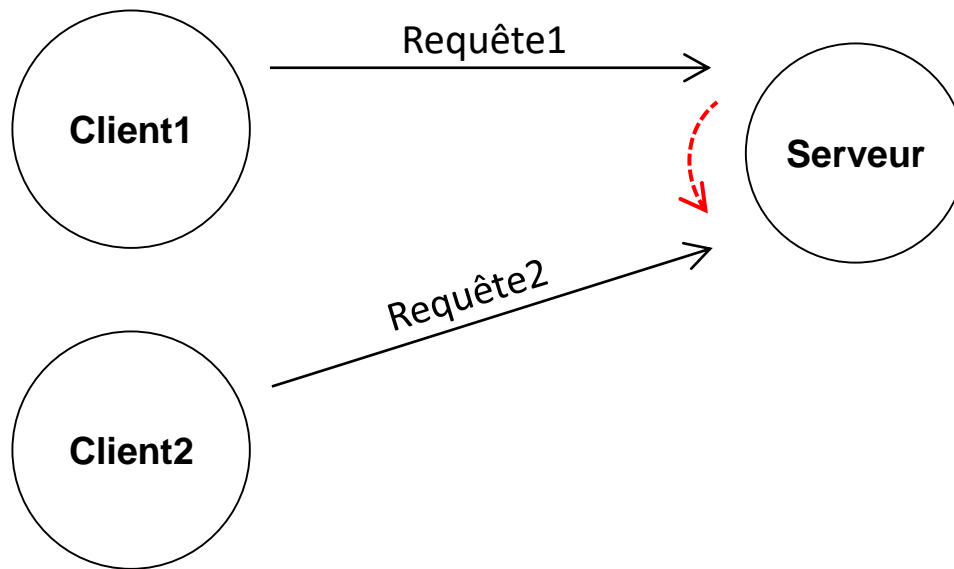
Avantages ?  
Inconvénients ?

# Pseudocode modèle de base

```
/* boucle infinie */  
while (true) {  
    /* attente session client (le serveur est  
       bloqué) */  
    client = accept ( écoute, &adresse, &len );  
  
    /* traiter la(es) requête(s) du client */  
    traiter_session ( client );  
  
    /* déconnexion client et passage à la  
       session suivante */  
    close ( client );  
}
```

# Modèle événementiel

- La tâche serveur traite les requêtes des clients en attente



Les canaux prêts (disponibles pour une entrée-sortie) sont traités selon un algorithme de sélection (*round-robin...*)

Utilisation de mécanismes système : multiplexage des E/S (**select**, **epoll**), E/S par interruption (**SIO**) ; ou utilisateur : **libevent**

Avantages ? Inconvénients ?



# select() : sélection d'un canal prêt

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

```
int select (int nfd, fd_set *readfds, fd_set *writefds,  
            fd_set *exceptfds, struct timeval *timeout) ;
```

Retourne le nombre de canaux prêts, ou 0 si timeout, ou -1 en cas d'erreur

- **select()** retourne le nombre de canaux (descripteurs de fichier) pour lesquels une opération d'entrée-sortie ne sera pas bloquante, en lecture (**readfds**), en écriture (**writefds**) ou si une exception (pas une erreur) s'est produite (**exceptfds**)
- **nfd** : plus grand numéro de descripteur présent + 1
- **readfds**, **writefds** et **exceptfds** : ensembles de bits positionnés avec les macros **FD\_ZERO**, **FD\_SET**, **FD\_CLR** et **FD\_ISSET** (voir slide suivant)
- **timeout** : si **NULL**, appel bloquant, sinon délai d'attente en secondes et microsecondes (0 indique retour immédiat)

# Gestion des descripteurs de fichiers

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

```
void FD_ZERO ( fd_set *fdset );  
void FD_SET ( int fd, fd_set *fdset );  
void FD_CLR ( int fd, fd_set *fdset );  
int  FD_ISSET ( int fd, fd_set *fdset );
```

- **fd\_set** : type ensemble de bits (opaque)
- **FD\_ZERO** : positionne à 0 tous les bits de l'ensemble
- **FD\_SET** : positionne à 1 le bit correspondant au descripteur **fd**
- **FD\_CLR** : positionne à 0 le bit correspondant au descripteur **fd**
- **FD\_ISSET** : retourne vrai si le bit correspondant au descripteur **fd** est positionné à 1

# Pseudocode modèle événementiel

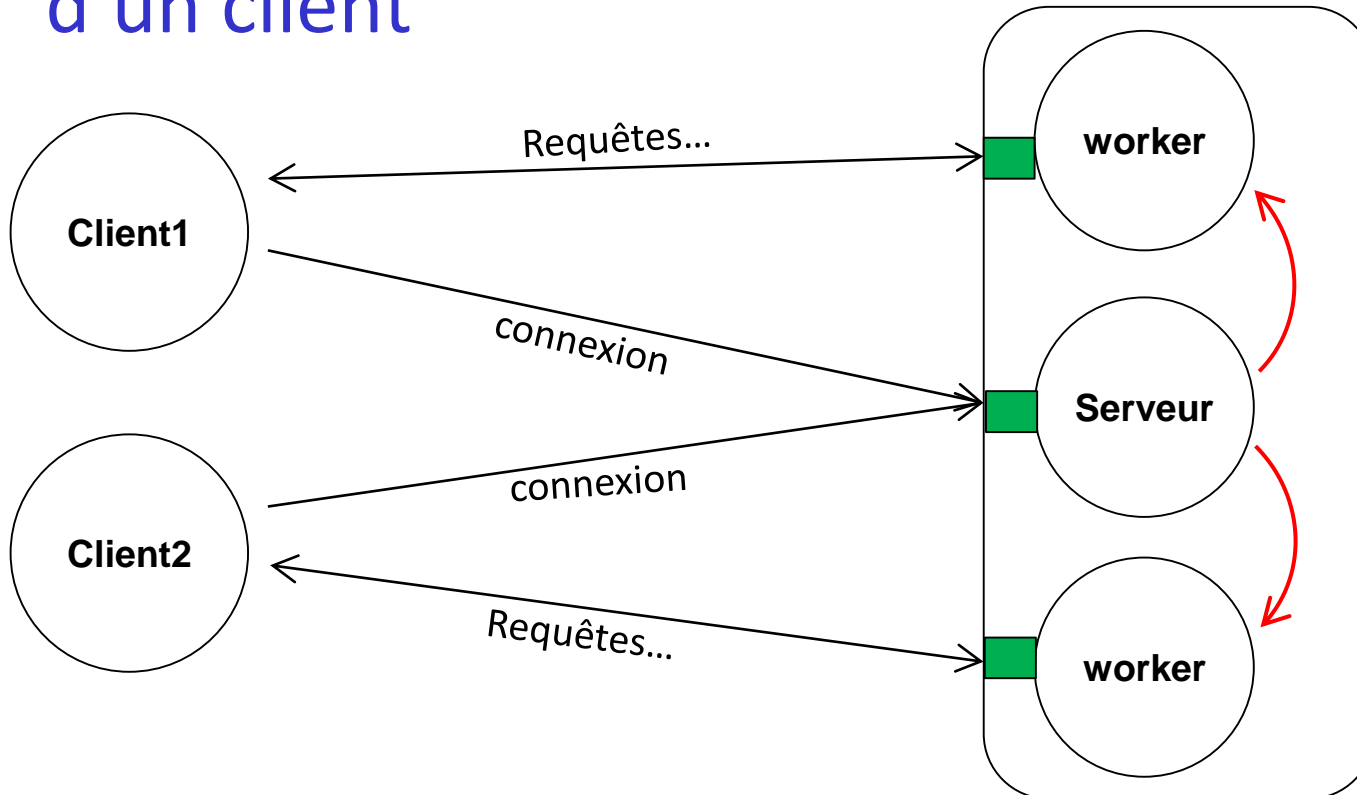
```
/* boucle infinie */
while ( true ) {
    /* positionner à 1 les descripteurs, y compris
       le canal d'écoute */
    nprets = select ( max, &lecture, NULL, NULL );
    pour nprets descripteurs fd à 1 dans lecture faire
        si canal d'écoute prêt alors
            canal = accept ( fd, &adresse, &len );
            ajouter canal à l'ensemble des descripteurs
        finsi
        /* traiter une requête du client */
        traiter_requete ( fd );
        ...
    finpour
}
```

# Modèle à *workers*

- La tâche serveur délègue les requêtes des clients à des tâches « enfant », appelées workers
- Gestion des workers : dynamique ou statique
  - Dynamique : les workers sont créés à la demande
  - Statique : un pool de workers est créé au démarrage du serveur
- Gestion des connexions : par client ou par requête
  - Par client : le worker gère la session du client, i.e. la connexion, toutes ses requêtes et la déconnexion
  - Par requête : le worker récupère le contexte du client et applique la requête (nécessite un protocole sans état) ; concerne surtout les applications Web (cf 3A).

# Modèle à workers dynamique

- La tâche serveur crée un worker à chaque connexion d'un client



Les workers sont créés avec **fork()** (processus) ou avec **pthread\_create()** (thread).  
Modèle vu dans l'application multithread du TP5.  
Voir **inetd**.

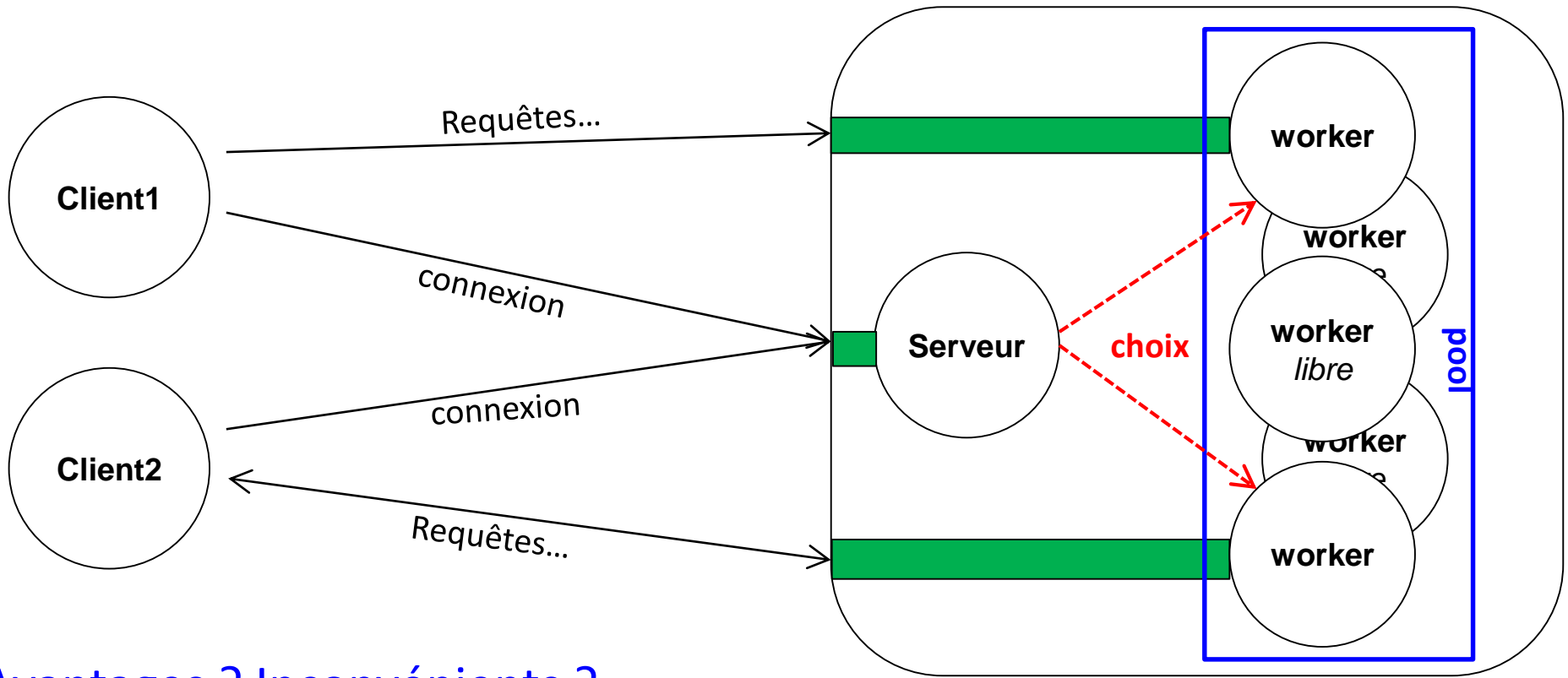
Avantages ? Inconvénients ?

# Pseudocode modèle worker dynamique

```
/* boucle infinie */
while (true) {
    /* attente session client (le serveur est
       bloqué) */
    client = accept ( écoute, &adresse, &len );
    créer un worker avec fork() ou pthread_create()
    si worker alors
        /* traiter la(es) requête(s) du client */
        traiter_session ( client );
        close ( client );
        pthread_exit() ou exit()
    finsi
}
```

# Modèle à workers statique

- Le serveur crée un pool de workers au démarrage, en attente. À chaque connexion d'un client, le serveur choisit un worker libre, sinon il attend



Avantages ? Inconvénients ?

# Pseudocode modèle worker statique

**serveur :**

```
    créer pool de workers
    while ( true ) {
        accepter une connexion entrante
        si pas de worker libre dans le pool alors attendre
        finsi
        sélectionner un worker libre et le marquer occupé
        lui communiquer le numéro de canal
    }
```

**worker :**

```
    while ( true ) {
        attendre numéro de canal
        gérer la session du client
    }
```



Séance 7 : gestion de la concurrence

# EXCLUSION MUTUELLE

# Programmation concurrente

- Programmation concurrente : deux ou plusieurs actions progressent en même temps
- Programmation parallèle : deux ou plusieurs actions s'exécutent simultanément
- Parallèle  $\Rightarrow$  Concurrent
  - Parallèle : machine multiprocesseurs
  - Concurrent : exécution sur 1 ou plusieurs processeurs
- Programmation multithread  $\Rightarrow$  Algorithmique concurrente
  - Plusieurs flots d'exécution concurrents
  - Coordination pour garantir les résultats
  - Comportement non-déterministe et asynchrone
  - Partage des ressources *thread-safe*
  - Modèles de programmation usuels

# Quelques modèles multithread

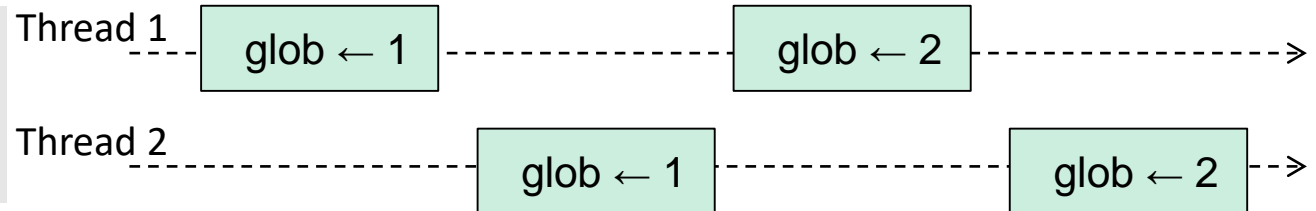
- Modèle à worker (maître/esclave) : un thread (maître) accepte des requêtes en entrée et affecte les tâches à effectuer aux threads (esclaves) qui produisent les résultats (← cf. cours précédent)
  - Variante 1 : les threads sont créés à la demande par le maître
  - Variante 2 : le maître crée un *pool* de threads esclave au démarrage de l'application, puis affecte le travail dans les files d'attente des esclaves
  - Exemple : serveur web, dans lequel les esclaves sont chargés de répondre aux requêtes d'un client précis
- Pipeline : l'application est découpée en tâches exécutées séquentiellement pour un travail donné, mais en concurrence par plusieurs threads pour des travaux différents
  - Gain lié au nombre d'étages du pipeline
  - « Travail à la chaîne » (exemple : traitement d'images ou de vidéos)
- Calcul parallèle : partitionnement statique du travail aux threads
  - Application aux multiprocesseurs (exemple : calcul numérique, simulation...)

# Difficultés de la programmation concurrente

- Entrelacement des séquences d'instructions

- Soit le code

```
int glob = 1;  
int getNext ( void ) {  
    return glob++;  
}
```

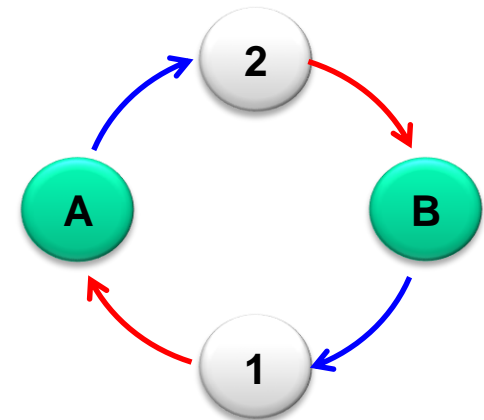


- Étreinte (*deadlock*)

- Blocage dû à la gestion des ressources

- Le thread 1 **détient** la ressource A
- Le thread 2 **détient** la ressource B
- La ressource B **est demandée** par le thread 1
- La ressource A **est demandée** par le thread 2

- Existence d'un cycle dans le graphe des allocations de ressources

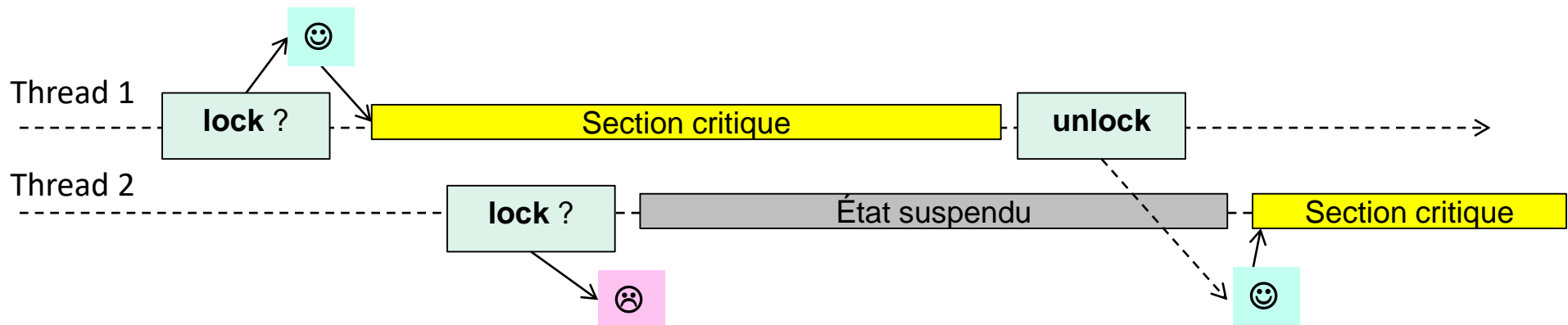


- Famine : un thread n'obtient jamais la ressource qu'il demande

- Un autre thread est toujours plus prioritaire que lui
- La ressource n'est jamais libérée

# Exclusion mutuelle

- Ressource partagée (mémoire) par plusieurs threads et au moins l'un d'entre eux veut la modifier  $\Rightarrow$  synchronisation
- Exclusion mutuelle
  - Cela consiste à s'assurer qu'un seul thread fait une tâche/action à un moment donné
  - Cette tâche/action est une section critique du thread dans la coopération avec les autres threads
  - La modification d'une variable partagée, la mise à jour d'une table, l'écriture d'un fichier sont des exemples d'actions qui doivent être effectuées en exclusion mutuelle.
- Nécessité de disposer d'opérations atomiques
  - Exécution indivisible des instructions
  - Les interruptions sont masquées
- Mécanismes
  - Verrou : un seul thread peut l'ouvrir, les autres threads sont suspendus (section critique : entre le verrouillage et le déverrouillage)  $\rightarrow$  utilisation de **mutex** avec les **pthread** (POSIX)
  - Sémaphore : entier et gestion de la file d'attente  $\rightarrow$  utilisation de **semaphore** (POSIX)



# pthread\_mutex\_init() : initialisation mutex

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

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

- **mex** : mutex à initialiser
- **attr** : attributs du mutex à initialiser avec **pthread\_mutexattr\_init()** (si NULL alors valeurs par défaut)
- Initialisation dynamique  $\Rightarrow$  libération mémoire avec **pthread\_mutex\_destroy()**
- Initialisation statique possible (valeurs par défaut) en initialisant le mutex avec **PTHREAD\_MUTEX\_INITIALIZER**
- Exemple

```
/* initialisation statique */  
pthread_mutex_t verrouStat = PTHREAD_MUTEX_INITIALIZER;  
  
/* initialisation dynamique */  
pthread_mutex_t verrouDyn;  
if ( pthread_mutex_init ( &verrouDyn, &attr ) != 0 ) {  
    perror ("mutex_init"); exit (EXIT_FAILURE); }  
...  
/* ATTENTION : pthread_mutex_destroy doit être fait sur un mutex déverrouillé !!! */  
if ( pthread_mutex_destroy ( &verrouDyn ) != 0 ) {  
    perror ("mutex_destroy"); exit (EXIT_FAILURE); }  
  
/* NE PAS FAIRE pthread_mutex_destroy sur verrouStat !!! */
```

# pthread\_mutex\_destroy() : suppression mutex

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

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

- **mutex** : mutex à détruire
- N'est à faire que si l'initialisation a été dynamique (avec **pthread\_mutex\_init()** )
- Consignes
  - Ne pas le faire si le mutex a été initialisé avec **PTHREAD\_MUTEX\_INITIALIZER**
  - **mutex** doit être déverrouillé pour le détruire
- Exemple

```
/* ATTENTION : pthread_mutex_destroy doit être fait sur un mutex déverrouillé !!! */  
if ( pthread_mutex_destroy ( &verrouDyn) != 0) {  
    perror ("mutex_destroy"); exit (EXIT_FAILURE); }
```

```
/* NE PAS FAIRE pthread_mutex_destroy sur verrouStat !!! */
```

# mutexattr\_init() et destroy() : gestion attributs

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

```
int pthread_mutexattr_init (pthread_mutexattr_t *attr);  
int pthread_mutexattr_destroy (pthread_mutexattr_t *attr);
```

- **attr** : attributs du mutex
- Allocation dynamique  $\Rightarrow$  détruire avec **pthread\_mutexattr\_destroy()**
- Exemple

```
pthread_mutexattr_t attr;  
if ( pthread_mutexattr_init ( &attr) != 0) {  
    perror ( "mutexattr_init"); exit (EXIT_FAILURE); }  
if ( pthread_mutexattr_settype(&attr, PTHREAD_MUTEX_ERRORCHECK) != 0) {  
    perror ("mutexattr_settype"); exit (EXIT_FAILURE); }  
...  
if ( pthread_mutexattr_destroy ( &attr ) != 0) {  
    perror ("mutexattr_destroy"); exit (EXIT_FAILURE); }
```



# (Dé)Verrouillage d'un mutex

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

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

- **mutex** : mutex à verrouiller ou à déverrouiller
- **pthread\_mutex\_lock()** est bloquant
- Mais pas **pthread\_mutex\_trylock()** qui retourne une erreur (**EBUSY**) si mutex est déjà verrouillé

```
pthread_mutex_t verrouStat = PTHREAD_MUTEX_INITIALIZER;
```

```
if ( pthread_mutex_lock ( &verrouStat ) != 0 ) {  
    perror ("mutex_lock"); exit (EXIT_FAILURE); }
```

```
/* section critique ... */
```

```
if ( pthread_mutex_unlock ( &verrouStat) != 0 ) {  
    perror ("mutex_unlock"); exit (EXIT_FAILURE); }
```

```
....
```

```
if ( ( code = pthread_mutex_trylock ( &verrouStat) ) != 0 ) {  
    if ( code == EBUSY ) { /* déjà verrouillé... reessayer plus tard ... */ }  
    else { perror ("mutex_trylock"); exit (EXIT_FAILURE); }  
}
```

# Barrière de synchronisation

- Principe

- N threads se donnent rendez-vous à une barrière
- Le dernier à y arriver débloque tous les autres

- Usage

- Type : **pthread\_barrier\_t**
- **pthread\_barrier\_init()** pour initialiser la barrière : elle a en paramètre le nombre N de threads qui doivent attendre avec **pthread\_barrier\_wait()**
- **pthread\_barrier\_destroy()** pour détruire
- **pthread\_barrier\_wait()** : attente à la barrière
  - Au déblocage, les threads reçoivent la valeur **0** , sauf un qui recevra la valeur **PTHREAD\_BARRIER\_SERIAL\_THREAD**
  - La barrière est ensuite remise dans l'état initial

# Variables condition

- Principe

- Permet à un thread de signaler un changement d'état d'une ressource partagée
- Permet à d'autres d'attendre le changement d'état de cette ressource

- Fonctions

- Type : **pthread\_cond\_t**
- Initialisation
  - statique avec **pthread\_cond\_t cond = PTHREAD\_COND\_INITIALIZER;**
  - ou dynamique avec **pthread\_cond\_init()** (et **pthread\_cond\_destroy()** pour détruire)
  - La variable est initialement en attente d'un signal
- **pthread\_cond\_signal()** permet de débloquent un thread en attente
- **pthread\_cond\_broadcast()** permet de débloquent tous les threads en attente
- **pthread\_cond\_wait()** bloque le thread appelant
  - Cette fonction doit avoir un mutex verrouillé en paramètre
  - Elle le déverrouille et bloque le thread sur la condition
  - Au déblocage, elle verrouillera le mutex avant de retourner

# Schéma d'utilisation des variables condition

```
/* thread Producteur */
```

```
① pthread_mutex_lock ( &mutex );
```

```
/* gérer la ressource partagée */
```

② pthread\_mutex\_unlock (&mutex );

③ pthread\_cond\_signal ( &cond );

```
/* thread Consommateur */
```

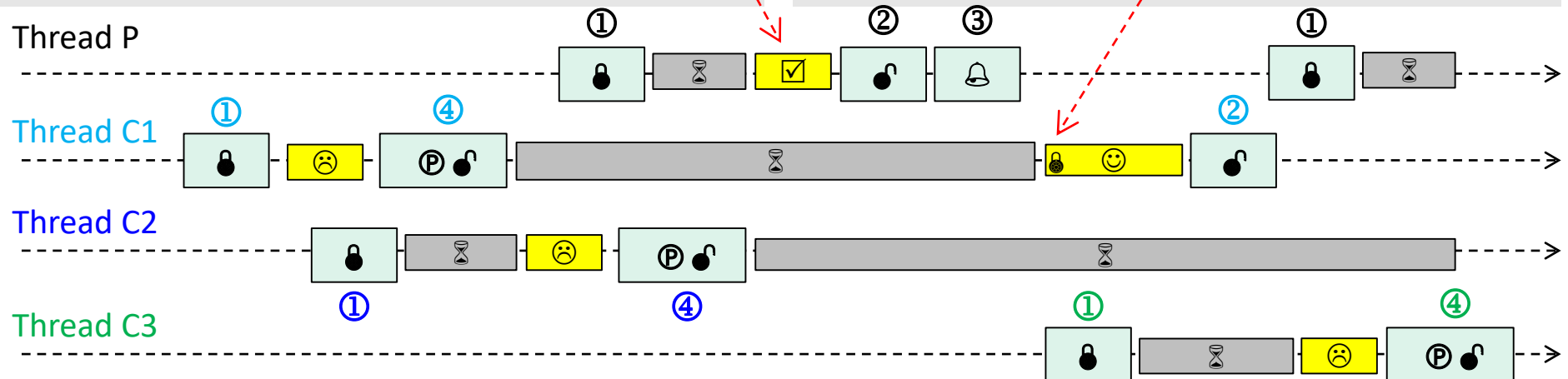
```
pthread_mutex_lock ( &mutex );
```

```
/* tester si la ressource est OK */
while ( /* ressource pas OK */ ) {
    pthread_cond_wait ( &cond, &mutex);
}
```

```
/* mutex est dans l'état verrouillé */
```

```
/* obtenir la ressource partagée */
```

```
pthread_mutex_unlock (&mutex );
```



# Les sémaphores

- Inventés par Dyjsktra en 1965
- Trois opérations sont proposées sur un type structuré Semaphore, comprenant un nombre de jetons et une file d'attente :
  - **init** ( sem, n ) : initialiser sem avec le nombre de jetons n
  - **P** ( sem ) : prendre un jeton ou s'endormir dans la file d'attente si pas de disponible
  - **V** ( sem ) : si une tâche attend dans la file, la réveiller en lui passant le jeton, sinon déposer le jeton
- Les opérations **P** et **V** sont atomiques : une seule tâche à la fois

# Algorithmes des opérations P et V

**type Semaphore :**

```
struct { Entier jetons ; File file; }
```

**operation init (Semaphore sem; Entier nbjeton )**

```
sem.jetons ← nbjeton  
sem.file ← créer file vide
```

**operation P (Semaphore sem )**

```
si ( sem.jetons <= 0 ) alors  
    insérer la tâche appelante dans sem.file  
    bloquer la tâche  
sinon sem.jetons ← sem.jetons - 1
```

**operation V (Semaphore sem )**

```
si ( sem.file est non vide ) alors  
    retirer une tâche de sem.file  
    mettre la tâche dans l'état prêt  
sinon sem.jetons ← sem.jetons + 1
```

# Les sémaphores POSIX

- Fichier entête sous Linux
  - Sémaphores POSIX → `<semaphore.h>`
- Un sémaphore contient un entier dont la valeur ne peut pas être négative
- Opérations possibles sur le type
  - Opération **init** avec `sem_init()` : initialiser un sémaphore à une valeur positive ou nulle
  - Opération **P** avec `sem_wait()` : si le sémaphore est supérieur à 0, enlever 1, sinon bloquer le thread
  - Opération **V** avec `sem_post()` : si le sémaphore vaut 0, débloquent éventuellement un thread, sinon ajouter 1
- Les opérations `sem_wait()` et `sem_post()` sont atomiques

# Exclusion mutuelle avec un sémaphore

- Une exclusion mutuelle est garantie pour un sémaphore initialisé à **1** (binaire)...
  - **sem\_wait()** : entrée en section critique (blocage éventuel), équivalent à **mutex\_lock()**
  - **sem\_post()** : sortie de section critique (déblocage éventuel d'un thread), équivalent à **mutex\_unlock()**
- ... et si chaque tâche les appelle successivement

```
/* thread principal */  
  
sem_t sem_mutex;  
  
sem_init ( &sem_mutex, 0, 1 ) ;  
  
/* lancement des threads */
```

```
/* thread worker */  
  
/* entrée */  
sem_wait ( &sem_mutex ) ;  
    /* section critique */  
sem_post ( &sem_mutex ) ;  
/* sortie */
```



# Signaler événement avec des sémaphores

- Initialisation du sémaphore à 0
- Appel de **sem\_post()** pour signaler un événement (par exemple, production d'une ressource)
- Appel de **sem\_wait()** pour être informé de l'événement (consommation de la ressource)
- Extension producteur / consommateur (voir #140)

```
/* thread principal */  
sem_t sem_event;  
sem_init ( &sem_event, 0, 0 ) ;
```

```
/* thread producteur*/  
  
/* production événement */  
sem_post ( &sem_event) ;
```

```
/* thread consommateur */  
  
sem_wait ( &sem_event ) ;  
  
/* consommation événement */
```

Séance 8 : gestion de la concurrence

# ALGORITHMES « CLASSIQUES »

# Producteur / Consommateur

- Deux threads doivent se transmettre des informations à travers un tampon
  - Un thread producteur qui ne doit produire que s'il y a des places libres dans le tampon
    - Places vides ou données consommées
  - Un thread consommateur qui ne doit consommer que s'il y a des données dans le tampon
    - Les données sont consommées une seule fois
    - Les données sont consommées dans l'ordre de la production
- Variantes
  - Tampon à une place ou à P places
  - Plusieurs consommateurs ou producteurs : 1:N
  - Plusieurs consommateurs et producteurs : N:M

# Invariant de production / consommation

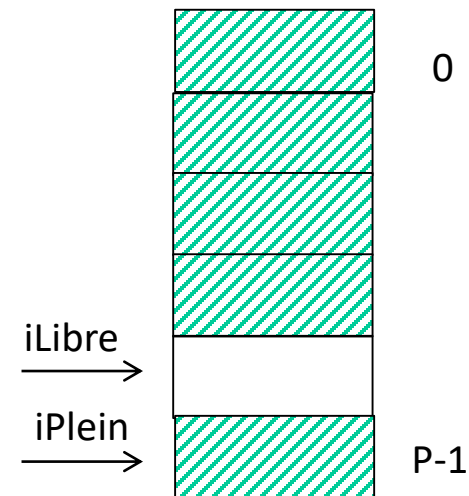
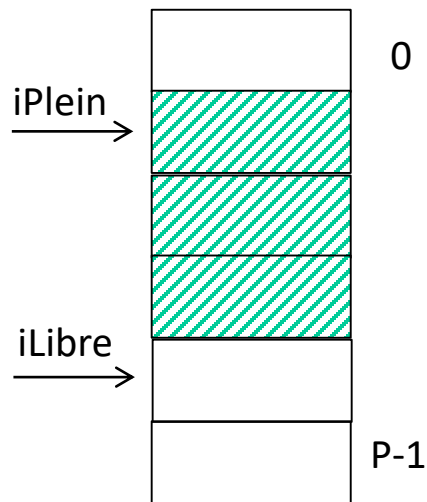
- Soit ***produit(t)***, le nombre d'éléments introduits dans le tampon jusqu'au temps  $t$
- Soit ***consommé(t)***, le nombre d'éléments retirés du tampon jusqu'au temps  $t$ ,
- Soit ***P***, la capacité du tampon
- Alors

$$\forall t \geq 0, 0 \leq \textit{produit}(t) - \textit{consommé}(t) \leq P$$

- Les actions des threads deviennent alors
  - producteur : attendre que ***produit(t) - consommé(t) < P***, puis déposer la donnée dans le tampon
  - consommateur : attendre pour retirer une donnée du tampon que ***produit(t) - consommé(t) > 0***

# Gestion du tampon

- Utilisation d'un tableau à **P** places
- Gestion circulaire
  - **iLibre** : indice de la prochaine place libre
  - **iPlein** : indice de la prochaine donnée
  - Additions modulo **P**



# Utilisation de sémaphores

- Gestion de la production : **sem\_t semP**
  - Initialisé à N
  - Le producteur demande avec **sem\_wait()** s'il peut produire une donnée
  - Après avoir consommé une donnée, le consommateur signale avec **sem\_post()** qu'il est possible d'en produire une autre
- Gestion de la consommation : **sem\_t semC**
  - Initialisé à 0
  - Le consommateur demande avec **sem\_wait()** s'il peut consommer une donnée
  - Après avoir produit une donnée, le producteur signale avec **sem\_post()** qu'il est possible d'en consommer une autre

# Pseudocode producteur/consommateur 1:1

```
/* thread principal */  
sem_t semP, semC  
sem_init( &semP, 0, P )  
sem_init( &semC, 0, 0 )  
iLibre ← 0  
iPlein ← 0
```

```
/* thread producteur */  
répéter  
  production donnée  
  sem_wait ( &semP )  
  tampon ← donnée  
  sem_post ( &semC )  
finrépéter
```

```
/* gestion tampon */
```

```
/* producteur */  
iLibre ← (iLibre + 1) % P  
Tampon[iLibre] ← donnée
```

```
/* consommateur */  
iPlein ← (iPlein + 1) % P  
Donnée ← Tampon[iPlein]
```

```
/* thread consommateur */  
répéter  
  sem_wait ( &semC )  
  donnée ← buffer  
  sem_post ( &semP )  
  consommation donnée  
finrépéter
```

# Passage à producteur/consommateur N:M

- Pas de changement pour la gestion des places libres et occupées
- Problème pour l'accès aux variables **iLibre** et **iPlein**
  - **iLibre** : les producteurs ont accès à la même variable
  - **iPlein** : les consommateurs ont accès à la même variable
  - Leur incrémentation est une section critique  
→ Protection des variables avec deux **mutex**
- Exemple pour **iLibre**  
`pthread_mutex_lock ( &mutex_libre )`  
`iLibre ← (iLibre + 1) % P`  
`tampon[iLibre] ← donnee`  
`pthread_mutex_unlock ( &mutex_libre )`



# Autres problèmes classiques

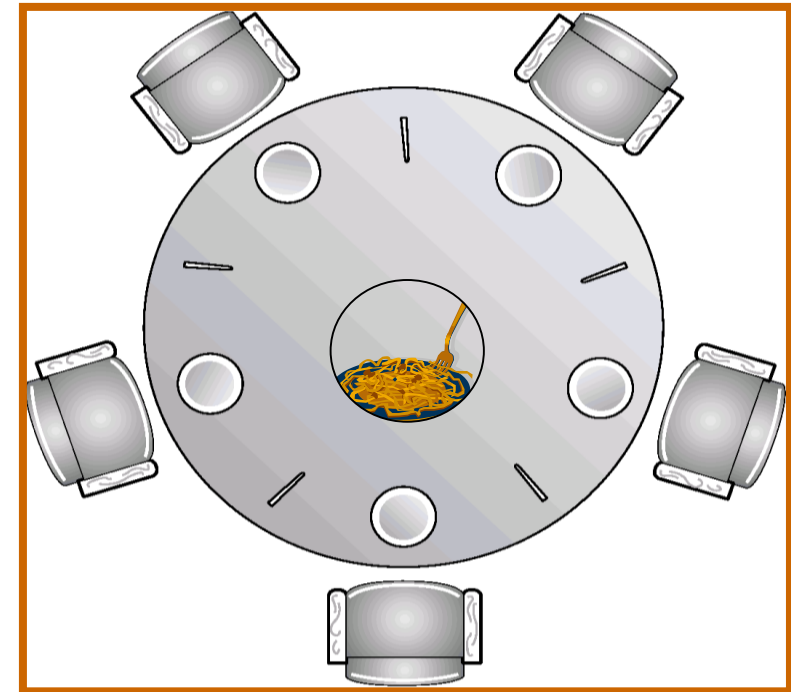
- Lecteurs / écrivain
  - Utilisation dans les bases de données
  - Prise en compte de mécanismes de priorité
- Diner des philosophes
  - Gestion de ressources partagées
  - Éviter les « *deadlocks* »
  - Éviter la famine

# Problème des lecteurs / écrivains

- Une ressource est partagée entre plusieurs lecteurs et plusieurs écrivains
- Plusieurs lecteurs peuvent accéder à la ressource « en même temps »
- Un seul écrivain peut accéder à la ressource à un instant donné (exclusion mutuelle)
- Nécessite deux verrous : un pour la lecture et un pour l'écriture
  - Un seul thread peut acquérir celui en écriture
  - Celui en lecture peut être acquis si celui en écriture est déverrouillé
  - Attention à la famine pour les écrivains
- Voir **`pthread_rwlock_*`**() pour une implantation POSIX

# Diner des philosophes

- 5 philosophes sont assis à une table, contenant 5 assiettes et 5 fourchettes (et un plat de spaghettis)
- Chaque philosophe pense pendant un temps variable, puis mange pendant un temps variable
  - Il ne peut manger que s'il dispose des deux fourchettes, celle de droite et celle de gauche
  - Il libère les deux fourchettes quand il pense
- Proposer une solution
  - où les philosophes sont représentés par des threads et les fourchettes par des ressources partagées
  - La solution devra garantir l'équité entre les philosophes, l'absence de famine et de deadlocks



**FIN**