Programmation Réseau

Anthony Busson
IUT Info Lyon 1

Plan du cours

- Introduction
  - Rôles du système et des applications
  - Rappel TCP-IP
  - Panorama des langages
- Couche socket : introduction
  - Panorama de la couche réseau du noyau
  - Détails d’une socket
- Couche socket : mise en oeuvre
  - Les appels systèmes et leur rôle
  - Les appels non bloquants
  - Setsockopt
- Annexes
  - Implémentation sous windows
  - Le détail des structures gérées par le noyau
- Pour aller plus loin:
  - Interfaçage avec le noyau: ioctl et netlink
  - Socket raw

Les TDs et TPs

- TD: quelques exercice sur
  - les structures gérées par le noyau
  - la gestion des erreurs

- Un serveur et un client web.

Bibliography

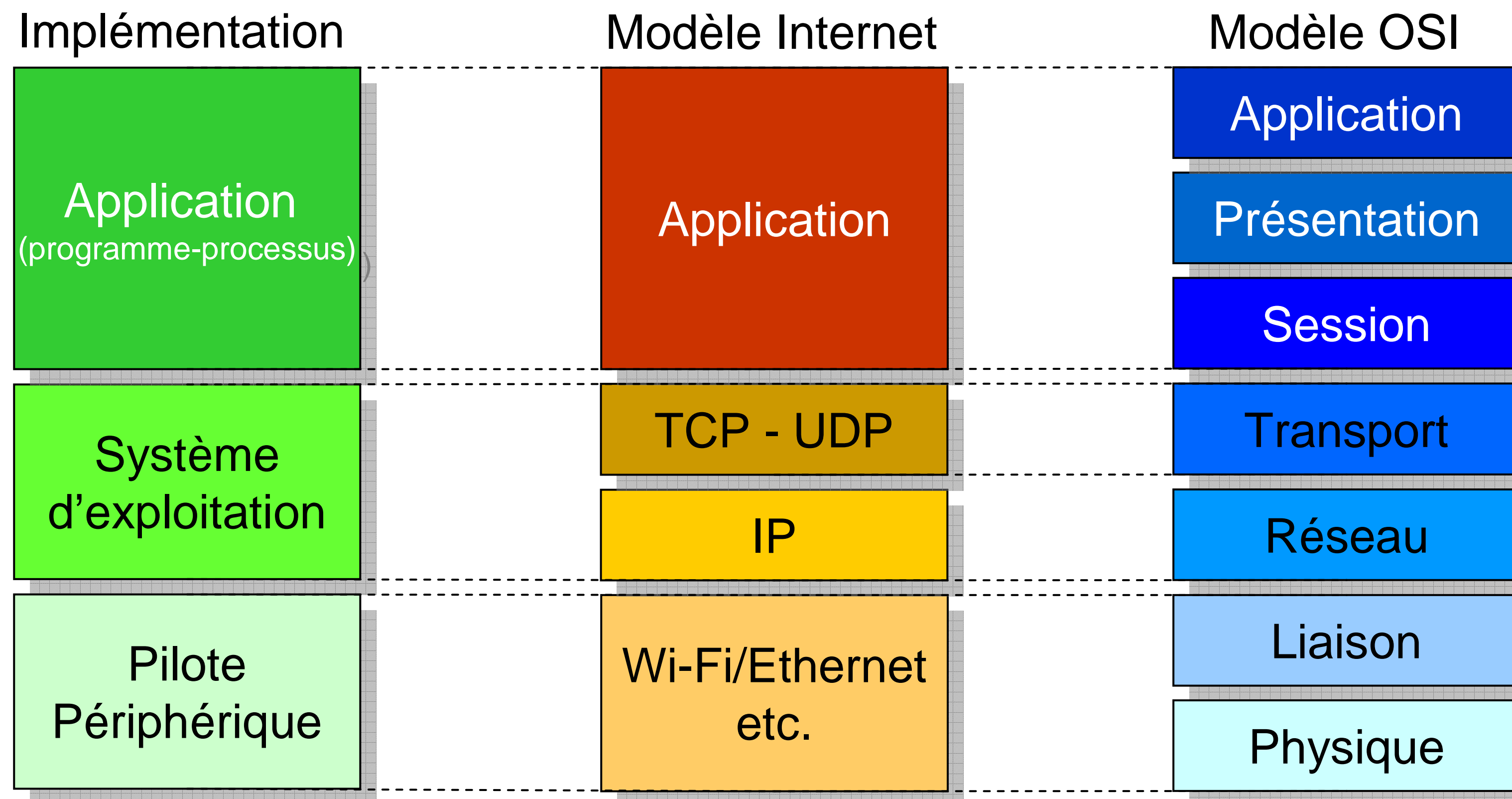
- Bibliography
  - Understanding Linux Network Internals. O'Reilly.
  - TCP/IP illustré. La mise en œuvre (volume 2). Vuibert.
  - Voir les RFC – quelques exemples:
    - RFC 789. Protocole IP.
    - RFC 3493. Basic Socket Interface Extensions for IPv6.

Prérequis

- Connaissance du langage C
  - Utilisation des fonctions et utilisation des prototypes
  - Pointeur
  - Tableau
  - Chaîne de caractères
  - Appel système fork()
  - Les threads (facultatif)

Partie 1: Introduction

Qui fait quoi?



Rappel : protocole IP

- Internet Protocol: couche réseau définissant
  - L’adressage des interfaces
  - Le format des paquets
  - Les procédures d’acheminement
- Le protocole IP est utilisé pour l’interconnexion des réseaux physiques.
- Deux versions du protocole cohabitent aujourd’hui:
  - IPv4 définit au début des années 80 (RFC 791)
  - IPv6 définit au début des années 200 (RFC 2373)

Rappel: adresse IP

IPv4

129.175.237.12 : 4 octets en écriture décimale

127.0.0.1 : Adresse de loopback

IPv6

2001:1:2:3:A01:BCD:2:345A : 16 octets divisé en 8 groupes de 2 octets en écriture hexadécimale

::1 : Adresse de loopback

Rappel: TCP et UDP

- TCP: Transport Control protocol
- Uniquement implémenté par les SE (pas dans le réseau)
- Plusieurs rôles:
  - Fiabiliser le transfert des informations
  - Contrôle de flot / contrôle de congestion
  - Identification des connexions

- Identification des connexions
  - Utilisation des numéros de ports (valeur codée sur 2 octets)
  - Deux ports pour chaque connexion: un port local / un port destination

Côté serveur

(port local, port dest) = (80, 2078)

Côté client

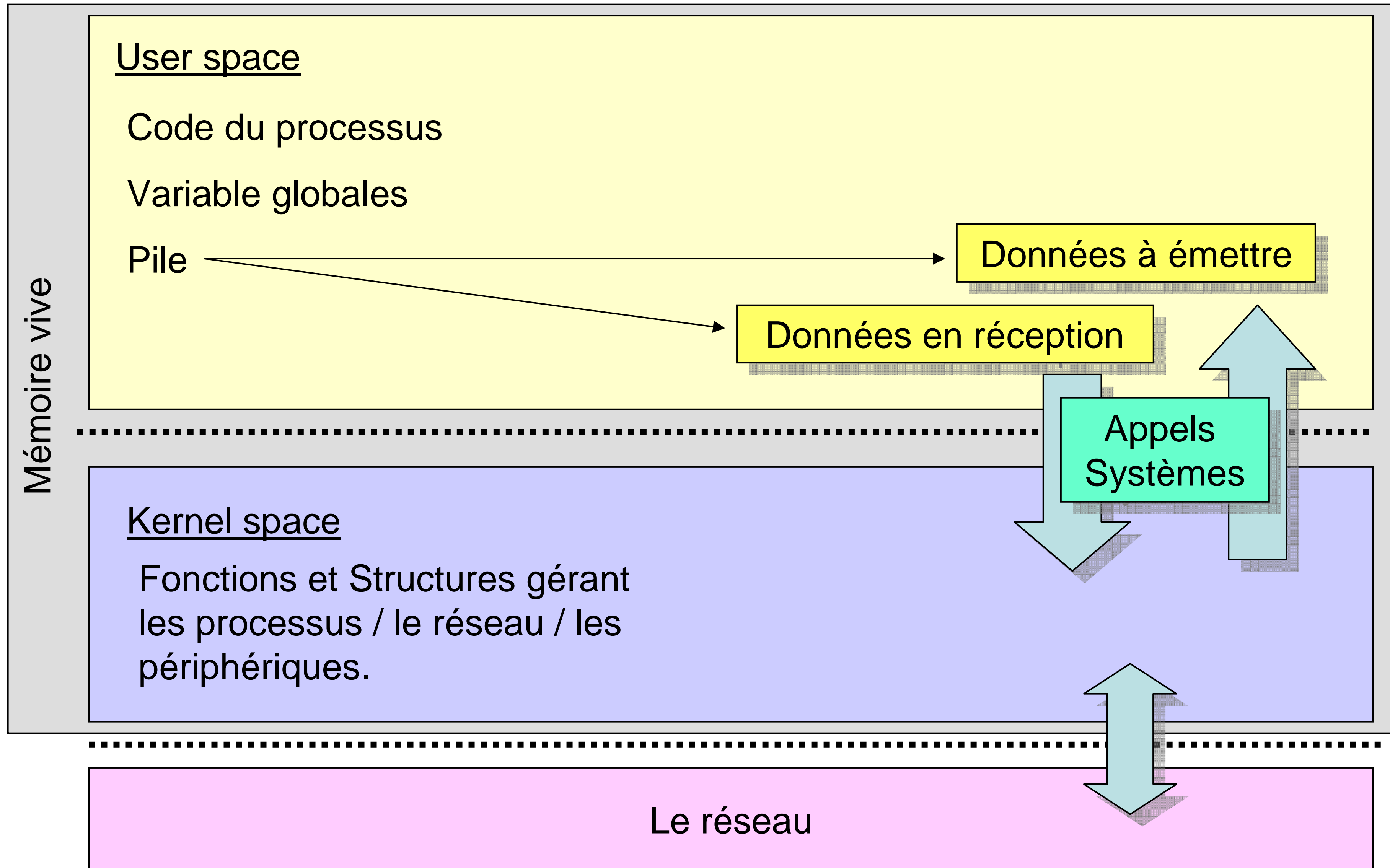
(port local, port dest) = (2078, 80)

Les langages

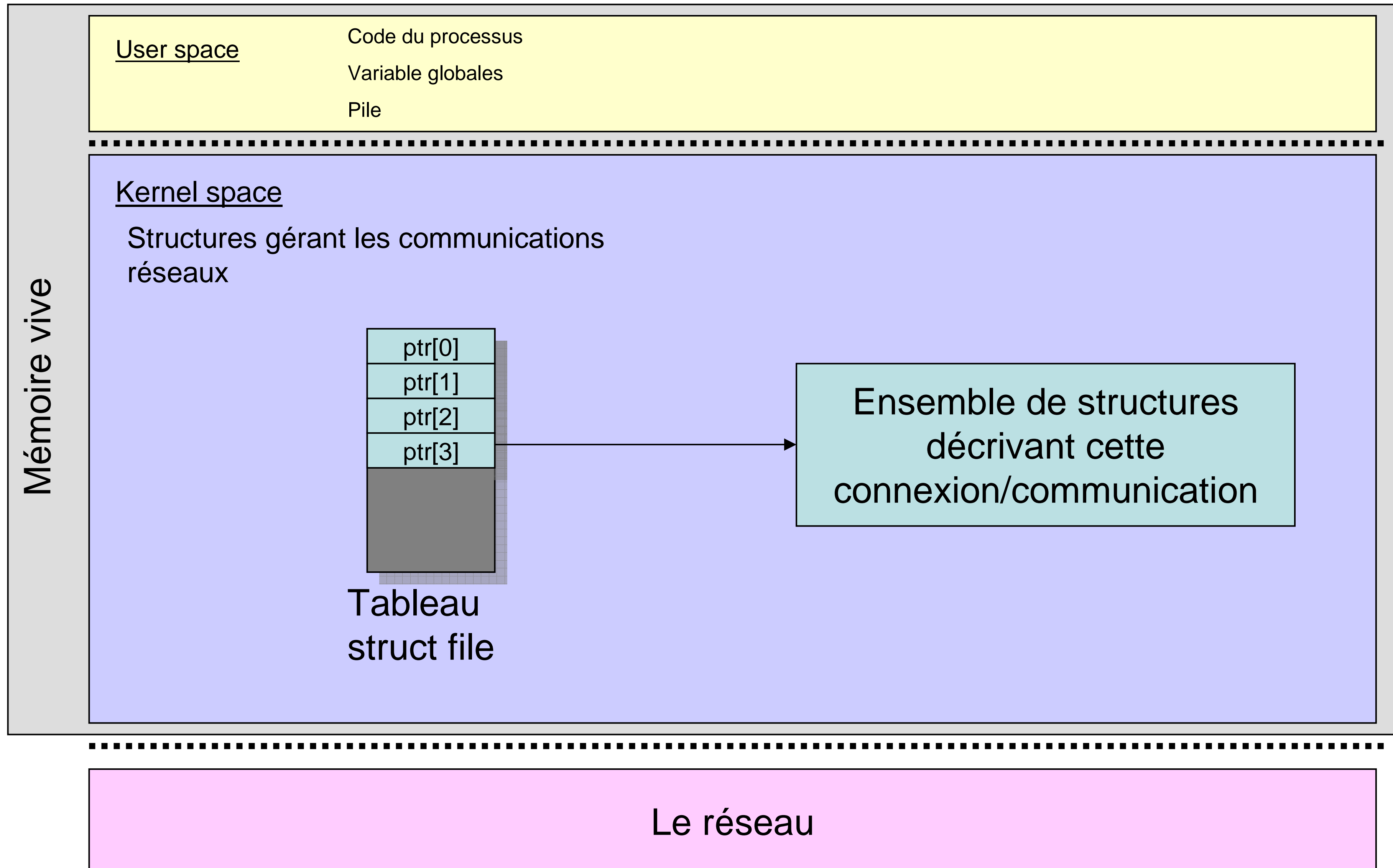
C	C++/java/C#(.net)	Python/Perl/PhP
Utilise les appels systèmes de bas niveau. Assez peu portable (sauf adaptation du code). Formateur.	Utilise des classes créant une abstraction avec les appels systèmes de bas niveau. Portable.	Utilise des librairies créant une abstraction avec les appels systèmes de bas niveau. Plus ou moins portable (fonction des librairies et de l'installation de l'interpréteur).
Protocole de routage Outils de configuration réseau Applications performantes Exemple: Ifconfig, ospfd, ripd, ip, apache, nfs, vsftpd, postfix (mail), etc.	Applications réseaux Applications et services web (plate-formes Java: JEE).	Application réseau simple. PhP: Plateforme ...

Partie 2: Couche socket

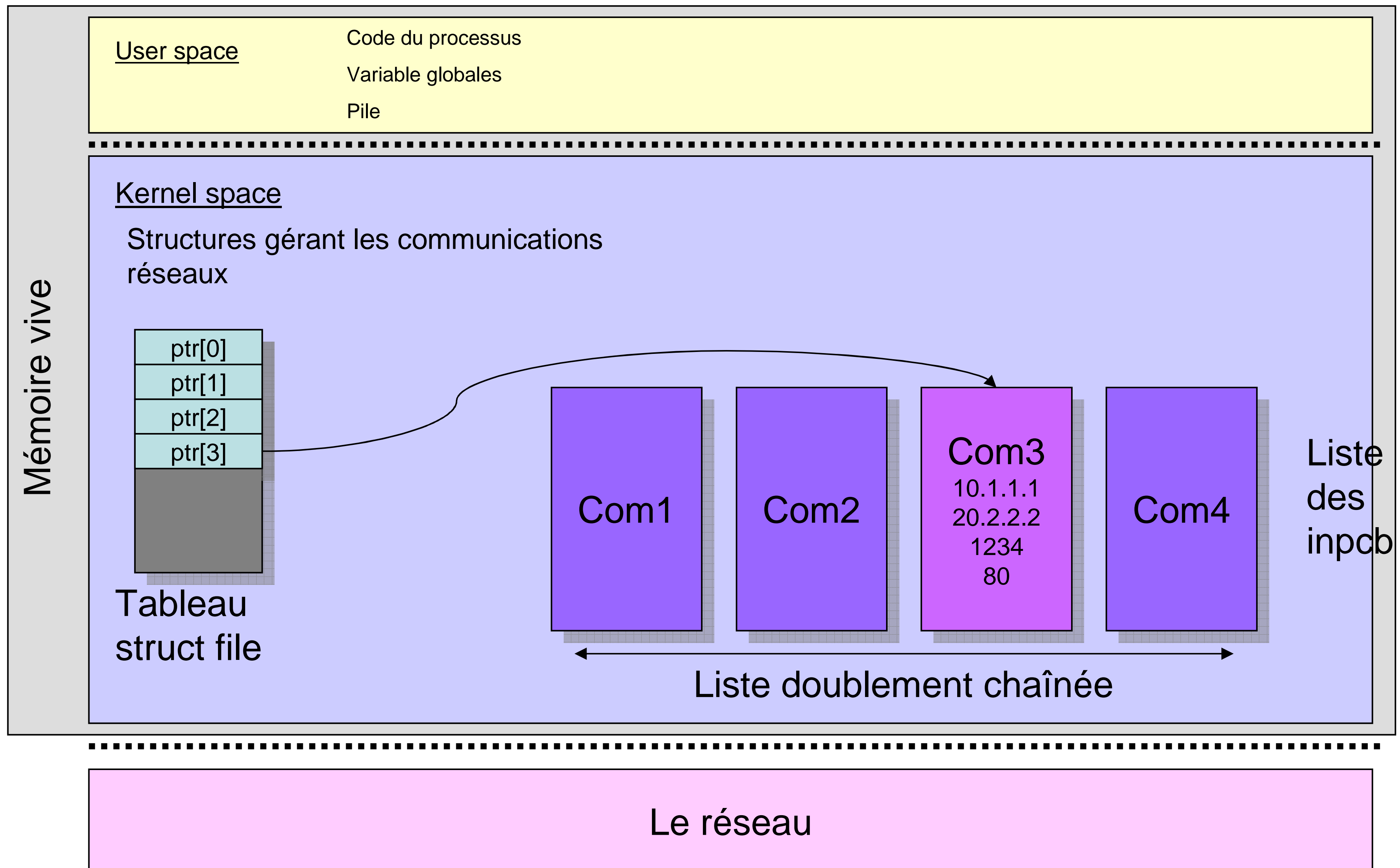
Panorama de l'implémentation réseau (1)



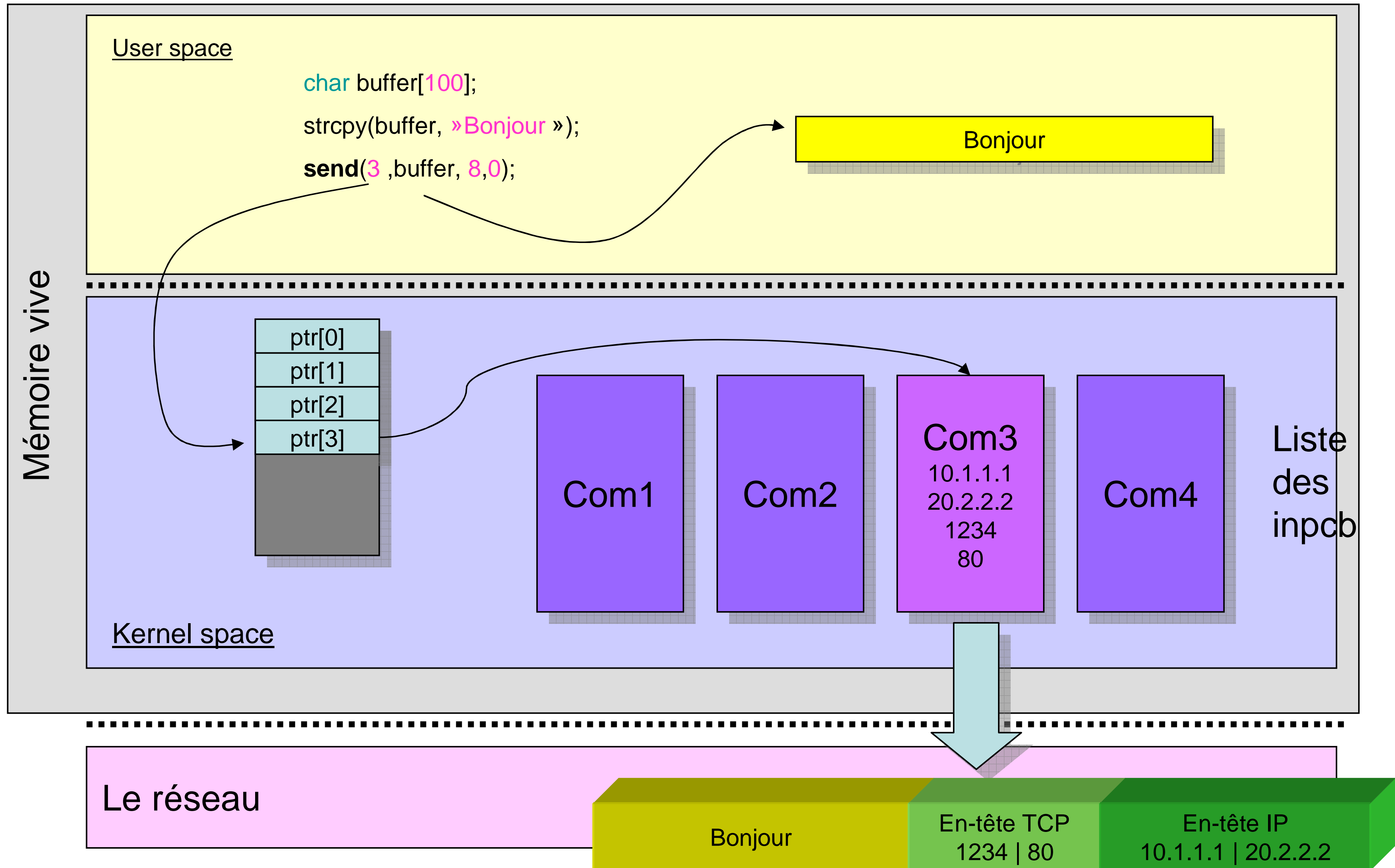
Panorama de l’implémentation réseau (2)



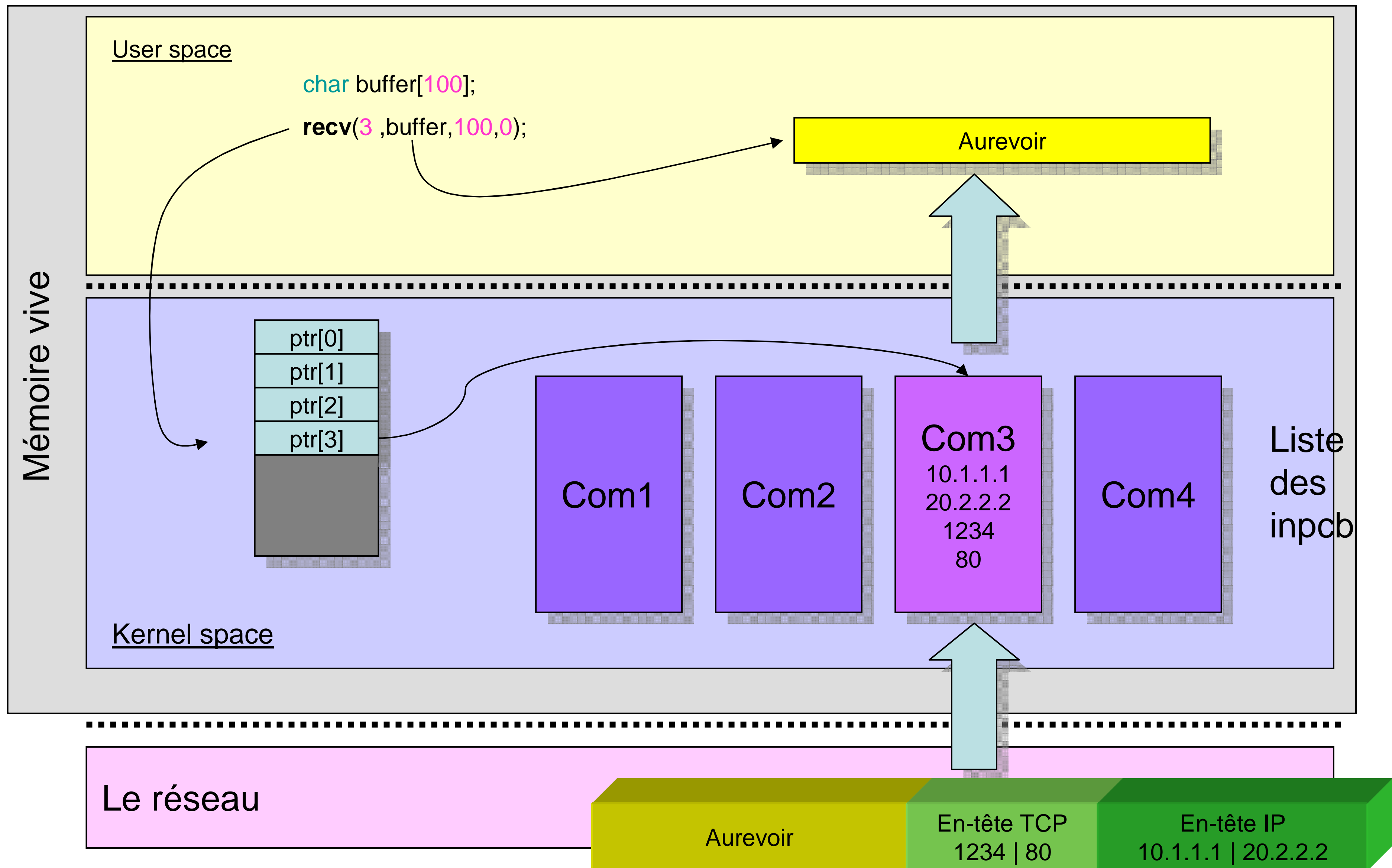
Panorama de l’implémentation réseau (3)



Émission d'un paquet



Émission d’un paquet

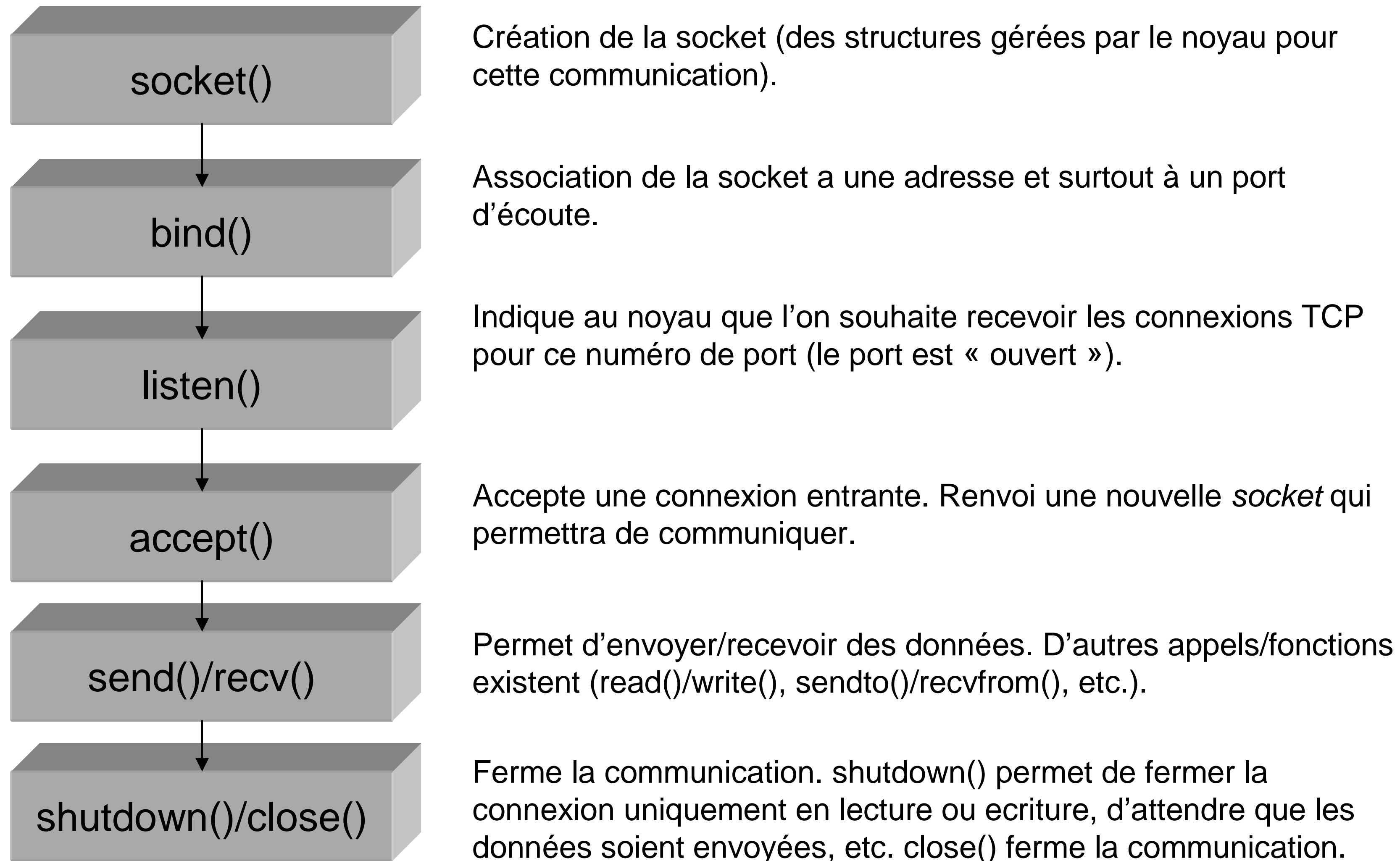


Partie 3: Les appels systèmes

Les appels systèmes

- Cas du serveur TCP (attend les connexions):
  - *socket():* créer le contexte de la communication (les structures *file, socket*, etc.). Renvoi un descripteur de fichier.
  - *bind():* associe à la socket les adresses locales de niveau transport (le port) et de niveau réseau (adresse IP).
  - *listen():* indique que la socket accepte de recevoir des connexions entrantes.
  - *accept():* le processus se met en attente des connexions entrantes. Il créé une nouvelle socket pour chaque nouvelle connexion entrante.
  - *sendto()/write()/writev():* émission des données
  - *recvfrom()/read()/readv():* réception des données
  - *close(), shutdown():* terminaison de le connexion TCP.

Les appels systèmes : serveur TCP



socket()

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

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

L’appel à **socket()** créer une nouvelle *socket*. Il créer les structures (*file, socket, incpb*) et les liens entre ces structures. En d’autres termes, il créé un point de communication. Il utilise le plus petit descripteur disponible et fait pointé le pointeur correspondant sur la structure file.

Les domaines: PF_INET, PF_INET6.

Les types: SOCK_STREAM, SOCK_DGRAM, SOCK_RAW.

Les protocoles: 0 (celui associé au domaine).

<u>Valeur retourné:</u> L’appel retourne le descripteur (un entier) en cas de succès, et -1 en cas d’erreur.

bind()

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

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

L'appel met à jour la socket de descripteur *sockfd* avec une adresse local. L'adresse locale est fournit au travers de son pointeur my_addr.

La structure utilisé dépend en fait du type de protocole réseau. Dans notre cas, ce sera systématiquement sockaddr_in pour l'IPv4. La structure sockaddr_in est la suivante:

```
 struct sockaddr_in {
          sa_family_t    sin_family;    // address family: AF_INET
          in_port_t      sin_port;      // Port in network byte order
          struct in_addr sin_addr;      // Internet address
  };

//Internet address:
  struct in_addr {
          uint32_t  s_addr;             // address in network byte order
  };
```

Valeur retourné: L'appel retourne 0 en cas de succès, et -1 en cas d'erreur.

bind(): ipv6

Dans le cas de l'IPv6, la structure est la suivante:

```
struct sockaddr_in6 {
          u_char            sin6_family;   // AF_INET6
          u_int16m_t        sin6_port;     // Transport layer port
          u_int32m_t        sin6_flowinfo; // IPv6 flow information
          struct in6_addr   sin6_addr;     // IPv6 address
          uint32_t       sin6_scope_id; // scope id
  };
```

A noter que d'après le RFC, il y a un champ en plus qui est sin6_len, mais qui n'apparaît pas sous certaines distributions de Linux. La structure `struct in6_addr` est la suivante:

```
struct in6_addr {
          u_int8_t  s6_addr[16];  // IPv6 address
};
```

listen()

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

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

L’appel indique le désir d’accepter les connexions entrantes. Il s’utilise pour les communications SOCK_STREAM (TCP). Le numéro de port pour lequel on accepte les connexions, à en principe, été mis à jour par *bind()* et est associé à la socket décrit par *sockfd*. La paramètre backlog indique le nombre maximal de connexions en attente d’acceptation (non encore traité par l’appel accept()).

Valeur retourné: L’appel retourne 0 en cas de succès, et -1 en cas d’erreur.

accept()

```
#include <sys/types.h>
#include <sys/socket.h>
int accept(int sock, struct sockaddr *adresse, socklent_t *longueur);
```

L’appel accept() permet d’accepter une connexion. Il est utilisé, en principe, par un serveur en SOCK_STREAM (TCP). Un appel à bind() et listen() doit être fait avant.

Si une connexion (d’un client) est arrivée, accept() renvoi un nouveau descripteur de fichiers pour la nouvelle socket. Le système met à jour les différentes structures liées à cette socket. La plupart des champs sont hérités de la socket initiale.

L’appel à socket peut être bloquant ou non. Si elle est bloquante, l’exécution du programme reste bloqué sur cet appel jusqu’à qu’une connexion arrive. Dans le cas où il est non bloquant et qu’il n’y a pas eu de nouvelles connexions, l’appel renvoi une erreur et errno vaut EWOULDBLOCK.

Le champ adresse permet de récupérer les propriétés du client (adresse IP et numéro de port utilisé). Le format de cette adresse dépend du protocole utilisé (IPv4 ou IPv6 pour ce qui nous concerne).

<u>Valeur retourné:</u> L’appel retourne un descripteur de fichier strictement positif en cas de succès, et -1 en cas d’erreur.

send()

```
#include <sys/types.h>
#include <sys/socket.h>
int send(int clientfd, const void *msg, size_t len, int flags);
```

L'appel *send()* permet d'envoyer des données au destinataire d'une socket. *send()* ne peut être utilisé qu'avec une socket connecté (SOCK_STREAM mais pas SOCK_DGRAM). En principe, *clientfd* est le descripteur qui a été renvoyé par l'appel à accept(). *msg* est un pointeur sur le message a envoyé. *len* est la taille du message. *flags* décrit les options (0 dans la plupart des cas).

Valeur retourné: L'appel retourne le nombre de caractères (d'octets) émis, et -1 en cas d'erreur.

recv()

```
#include <sys/types.h>
#include <sys/socket.h>
int recv(int clientfd, void *buf, int len, unsigned int flags);
int recvfrom(int clientfd, void *buf, int len, unsigned int flags, struct sockaddr *from, socklen_t *fromlen);
```

L'appel *recv()* permet de recevoir des données provenant d'une socket distante. *recv()* ne peut être utilisé qu'avec une socket connecté (SOCK_STREAM mais pas SOCK_DGRAM). En principe, *clientfd* est le descripteur qui a été renvoyé par l'appel à *accept(). buf* est un pointeur sur l'emplacement ou les données reçues doivent être placées. *len* est la taille de cet emplacement. *flags* décrit les options (0 dans la plupart des cas). L'appel à ces deux fonctions sont bloquantes jusqu'à la réception de données.

recvfrom() est identique à *recv()* lorsque *from* est *NULL* et *fromlen* est égale 0.

Valeur retourné: L'appel retourne le nombre de caractères (d'octets) lu, et -1 en cas d'erreur.

close()-shutdown()

```
#include <unistd.h>
int close(int socketfd);


#include <sys/socket.h>
int shutdown(int socketfd, int how);
```

L’appel close() ferme le descripteur de fichiers. Toute action sur la socket génère alors une erreur.

L’appel à shutdown() ferme de manière unidirectionnel ou bidirectionnel la communication suivant la valeur de l’argument *how*:

SHUT_RD, ferme la socket en réception,

SHUT_WR, ferme la socket en émission,

SHUT_RDWR, ferme la socket en réception et émission.

Valeur retourné: Les deux appels retournent 0 en cas de succès, et -1 en cas d’erreur.

Les appels systèmes : client TCP

Appel	Description
socket()	Création de la socket (des structures gérées par le noyau pour cette communication).
connect()	Permet de se connecter au serveur (établissement de la connexion TCP).
send()/recv()	Permet d'envoyer/recevoir des données. D'autres appels/fonctions existent (read()/write(), sendto()/recvfrom(), etc.).
shutdown()/close()	Ferme la communication. shutdown() permet de fermer la connexion uniquement en lecture ou ecriture, d'attendre que les données soient envoyées, etc. close() ferme la communication.

connect()

```
#include <sys/types.h>
#include <sys/socket.h>
int connect(int sockfd, struct sockaddr *serv_addr, socklen_t addrlen );
```

L’appel à *connect()* est utiliser essentiellement en mode SOCK_STREAM (TCP). Il établit la connexion avec l’extrémité (le serveur) décrite dont l’adresse et le numéro de port sont donnés par l’argument *serv_addr*. Il met également à jour les structures internes relatives à la socket (numéro de port source par exemple). L’argument *addrlen* est la taille du champ *struct sockaddr*.

<u>Valeur retourné:</u> L’appel retourne 0 en cas de succès, et -1 en cas d’erreur.

Les appels systèmes : serveur UDP

Création de la socket (des structures gérées par le noyau pour cette communication).

Association de la socket a une adresse et surtout à un port d'écoute.

Permet d'envoyer/recevoir des données. D'autres appels/fonctions existent (recvmsg/sendmsg.).

Ferme la communication. shutdown() permet de fermer la connexion uniquement en lecture ou ecriture, d'attendre que les données soient envoyées, etc. close() ferme la communication.

sendto()
recvfrom()

```
#include <sys/types.h>
#include <sys/socket.h>
int sendto(int clientfd, void *buf, int len, unsigned int flags, struct
sockaddr *from, socklen_t fromlen);
int recvfrom(int clientfd, void *buf, int len, unsigned int flags, struct
sockaddr *from, socklen_t *fromlen);
```

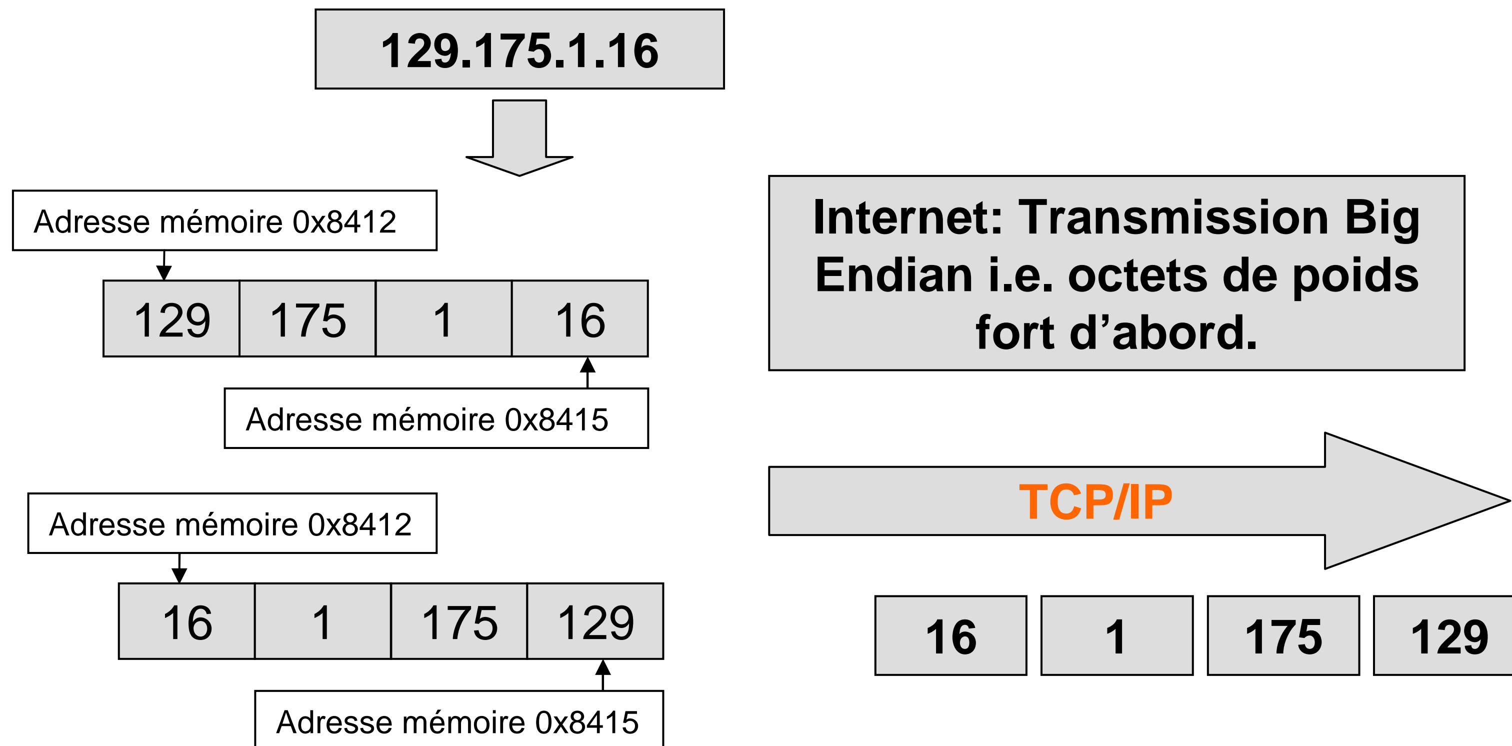
Ces deux appels permettent d'envoyer et recevoir des données sur une socket. Elles fonctionnent à la fois en mode SOCK_DGRAM (UDP) et SOCK_STREAM (TCP). Contrairement o *send()/recv()* utiliser dans le contexte SOCK_STREAM, il y a 2 arguments supplémentaire *from* et *fromlen* décrivant l'adresse de la socket distante (adresse IP et numéro de port) et sa taille. Dans le cas de *recvform(),* cette adresse est mise à jour par l'appel. Dans le cas de *sendto(),* il sert à indiquer l'adresse et le numéro de port de la destination.

Valeur retourné: L'appel retourne le nombre d'octets reçus ou envoyés, et -1 en cas d'erreur.

La mise à jour des adresses

Format réseau: Big Endian

- Il existe deux formats pour stocker les entiers en mémoire:
  - Litlle Endian: l'octet de poids faible est stocké à la plus petite adresse.
  - Big Endian: l'octet de poids faible est stocké à la plus grande adresse.
- L'Internet utilise toujours la transmission Big Endian.



Les fonctions de conversions: Système → Big Endian

```
#include <arpa/inet.h>
uint32_t htonl(uint32_t hostlong);
uint16_t htons(uint16_t hostshort);
uint32_t ntohl(uint32_t netlong);
uint16_t ntohs(uint16_t netshort);
```

- `ntoh` : network to host
- `hton` : host to network
- `s` : short interger (16 bits)
- `l` : long integer (32 bits)

```
int inet_aton(const char *cp, struct in_addr *pin);
char * inet_ntoa(struct in_addr in);
```

- Convertit une chaîne ascii (« 127.0.0.1 ») en format réseau et inversement.

```
int inet_pton(int af, const char *src, void *dst) ;
const char *inet_ntop(int af, const void *src, char *dst, size_t size).
```

- Marche en IPv4 et IPv6. pton() convertit d’une chaîne ascii en binaire et ntop() fait l’inverse.

La fonction getaddrinfo()

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

int getaddrinfo(const char *hostname, const char *servname, const struct
addrinfo *hints, struct addrinfo **res);
void freeaddrinfo(struct addrinfo *ai);
```

La fonction *getaddrinfo()* permet d'obtenir une liste d'adresse IP et de numéro de port. Cette fonction est plus pratique et plus fléxible que les fonctions classiques/anciennes *gethostbyname()* et *getservname().* Cette fonction met à jour les différents champs d'une liste chaînée dont les éléments sont de type struct addrinfo. Cette structure est la suivante:

```
struct addrinfo {
        int ai_flags; // input flags : AI_PASSIVE
        int ai_family; // protocol family for socket: AF_INET (IPv4) AF_INET6 ou AF_UNSPEC (v4 ou v6)
        int ai_socktype; // socket type: SOCK_STREAM, SOCK_DGRAM ou SOCK_RAW
        int ai_protocol; /* protocol for socket : 0 (un seul protocol existe en pratique pour une socket type
        socklen_t ai_addrlen; /* length of socket-address
        struct sockaddr *ai_addr; // un pointeur sur une addresse (struct sockaddr_in ou sockaddr_in6)
        char *ai_canonname; // canonical name for service location
        struct addrinfo *ai_next; // pointer to next in list
};
```

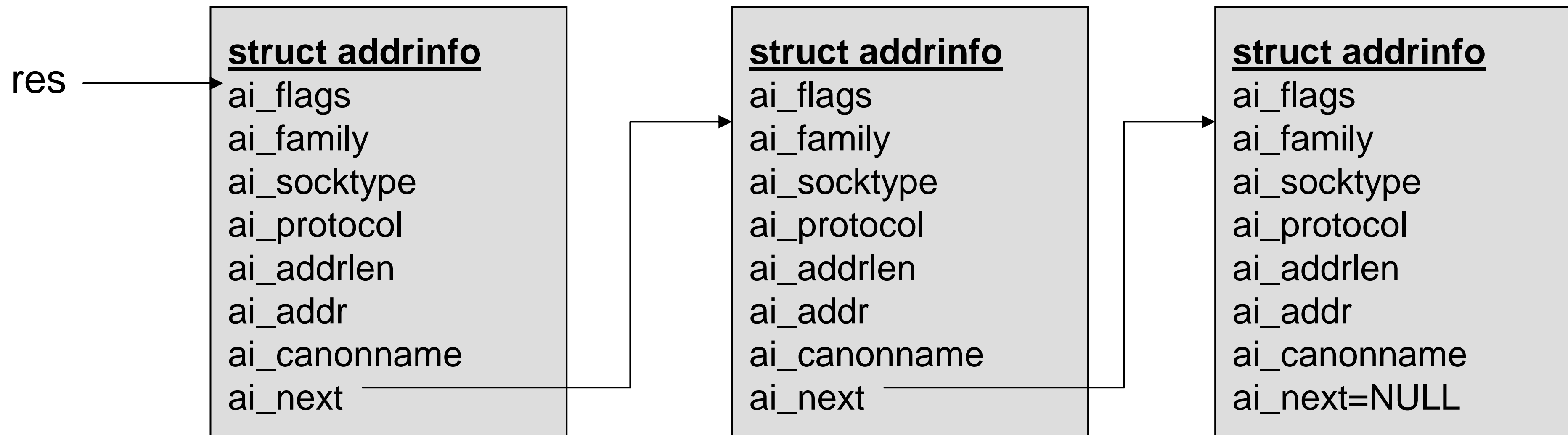
Valeur retourné: L'appel retourne 0 en cas de succès, et -1 en cas d'erreur.

La fonction getaddrinfo()

```
sruct addrinfo *res, hints, *parcours;

//Mise à jour de hints.
...
if(getaddrinfo(« www.anthonybusson.fr », « http », &hints, &res){

for(parcours=res;parcours!=NULL;parcours=parcours->ai_next)
        printf(« Canonical name = %s\n »,parcours->ai_canonname);
.
.
.
```



La fonction getaddrinfo(): serveur

- Initialisation des champs: le serveur

```
struct addrinfo hints, *res;
int sockfd, error;

//Mise à jour de hints (indiquant les préférences: protocole, etc.)
memset(&hints,0,sizeof(hints)); //Met à 0 tous les champs de hints
hints.ai_family = AF_INET6; //Fonctionne en IPv4 et IPv6
hints.ai_socktype = SOCK_STREAM;
hints.ai_flags = AI_PASSIVE; //Côté serveur

if(getaddrinfo(NULL, « 80 », &hints,&res)!=0) gai_strerror(error);

if(sockfd=socket(res->ai_family,res->ai_socktype,res->ai_protocol)<0)
    perror(« Erreur socket »);
…
freeaddrinfo(res); //Ne pas oublier de libérer la liste chaînée
```

La fonction getaddrinfo(): client

- Initialisation des champs: le client

```
struct addrinfo hints, *res;
int sockfd;

//Mise à jour de hints (indiquant les préférences: protocole, etc.)
memset(&hints,0,sizeof(hints)); //Met à 0 tous les champs de hints
hints.ai_family = AF_UNSPEC; //Fonctionne en IPv4 et IPv6
hints.ai_socktype = SOCK_STREAM;

if(getaddrinfo(« www.exemple.com », « 80 », &hints,&res)!=0) gai_strerror(error

if(sockfd=socket(res->ai_family,res->ai_socktype,res->ai_protocol)<0)
     perror(« Erreur socket »);
…
freeaddrinfo(res); //Ne pas oublier de libérer la liste chaînée
```

Gérer les erreurs: errno

- errno est une variable globale indiquant la dernière erreur.
- perror se sert de errno pour savoir quelle erreur s’est produite.
- Pour chaque appel système, un ensemble de constante sont définit.
- Elles décrivent les différentes erreurs possibles.

Gestion de certaines options

La fonction setsockopt()

```
#include <sys/socket.h>
int getsockopt(int socket, int level, int option_name, void *restrict
option_value, socklen_t *restrict option_len);
int setsockopt(int socket, int level, int option_name, const void
*option_value, socklen_t option_len);
```

setsockopt() permet de modifier les options liées à la socket. Ceci peut être fait à différent niveaux. Le champ level spécifie le niveau auquel l'option s 'applique. Par exemple, pour modifier une option au niveau socket, le champ level doit valoir SOL_SOCKET. Pour un niveau donné, il est possible de modifier plusieurs options. Pour le niveau socket, les principales sont:

SO_REUSEADDR //enables local address reuse

SO_KEEPALIVE enables keep connections alive

SO_LINGER linger on close if data present

…

getsockopt() permet d'obtenir les paramètres/options d'une socket.

<u>Valeur retourné:</u> L'appel retourne 0 en cas de succès, et -1 en cas d'erreur.

Les options du niveau IP

- Liste non exhaustive
  - IP_TOS: permet de changer le type de service du paquet IP
  - IP_TTL : permet de changer le TTL du paquet
  - IP_ADD_MEMBERSHIP: ajoute la socket comme membre de ce groupe (envoi un message IGMP)
  - IP_MULTICAST_TTL: fixe le TTL des paquets multicast (pour cette socket)
  - MCAST_LEAVE_GROUPE: on quitte le groupe (émission d'un message IGMP leave).

Les options du niveau TCP

- Liste non exhaustive:
  - TCP_NODELAY: permet d'émettre des segments dès qu'il y a des données dans le buffer (pas d'attente de remplissage du tampon d'émission).
  - TCP_MAXSEG: définit la taille maximale des segments.
  - TCP_SYNCNT: permet de modifier le nombre de TCP SYN retransmit lors de l'ouverture de connexion.
  - TCP_KEEPIDLE – TCP_KEEPINTVL – TCP_KEEPCNT: permet de modifier les paramètres du keepalive.
  - TCP_CONGESTION: permet de modifier l'algorithme de contrôle de congestion (reno, westwood, lp, cubic).

Les appels non bloquants

La fonction select

select()

- Comment faire si on a plusieurs connexions à gérer en même temps.
  - Un seul processus peut gérer plusieurs connexions
  - Un seul processus peut écouter sur plusieurs port
- Les appels systèmes read(), send(), ou accept() sont bloquants.
- Impossibilité de gérer plusieurs connexions simplement avec ces appels.

La fonction select() (1)

```
#include <sys/time.h> #include <sys/types.h>
#include <unistd.h>
int select(int n, fd_set *readfds, fd_set *writefds, fd_set *exceptfds,
struct timeval *timeout);
FD_CLR(int fd, fd_set *set); FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set); FD_ZERO(fd_set *set);
```

L'appel système *select()* indique un changement d'état sur une liste de descripteurs de fichiers. Ce changement d'état peut indiquer:

1. la possibilité de lire des données sur le descripteur ou de manière équivalente inique que l'on a reçu des données sur une socket,
2. l'arrivée d'une connexion,
3. d'écrire des données sur le descripteur (si le tampon était plein),
4. un evénement exceptionnel, l'arrivée de données hors bande pour ce qui concerne les sockets.

Pour les cas 1 et 2, l'argument *readfds* indique l'ensemble des descripteurs que l'on souhait surveiller. Pour les cas 3 et 4, ce sont les arguments *writefds* et *exceptfds* respectivement. Ces arguments doivent initialiser avec la macro FD_ZERO(). Un descripteur est rajouté/retiré à un ensemble avec les macros FD_SET() et FD_CLR().

select() laisse dans les arguments uniquement les descripteurs ayant subi un changement d'état.

<u>Valeur retourné:</u> L'appel retourne le nombre de descripteurs ayant un changement d'état, et -1 en cas d'erreur.

La fonction select() (2)

```
#include <sys/time.h>
#include <sys/types.h>
#include <unistd.h> int select(int n, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

L'argument n décrit le plus grand descripteur de l'ensemble. Select() surveillera les descripteurs appartenant aux ensembles et dont la valeur est comprise entre 0 et n.

Timeout permet de fixer une durée maximale pour l'appel à select(). Select() restera bloqué au maximum durant la durée de timeout. Si timeout est NULL, select() peut rester bloqué indéfiniment. La structure *timeval* est la suivante:

```
struct timeval {
    int    tv_sec;        /* secondes */
    int    tv_usec;    /* microsecondes */
};
```

<u>Valeur retourné:</u> L'appel retourne le nombre de descripteur ayant subit un changement d'état, 0 à l'expiration du timeout et -1 en cas d'erreur.

La fonction select() (3)

- select() se débloque aussi lors de la fermeture d’une connexion.
- Il faut pouvoir savoir quel événement s’est produit sur la socket.
- On peut par exemple utiliser l’appel système ioctl() avec l’option FIONREAD.

Annexe 1

La portabilité Linux/Windows

Portabilité Linux/Windows

- Les librairies ne sont pas les mêmes
- Les types de variable sont différentes
- Les fonctions sont généralement les mêmes mais leur prototype différent souvent.

Les librairies

```
#include <sys/wait.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
```

```
#include <winsock2.h>
#include <ws2tcpip.h>
```

Les librairies : code portable

```
#ifdef WIN32 /* Si on est sous Windows */

        #define _WIN32_WINNT 0x0501 /* Si on est pas sous windows XP et que l'on souhaite utiliser getaddrinfo() */
        #include <winsock2.h> /* En-tete Windows  */
        #include <ws2tcpip.h>

#elif defined linux /* si on est sous Linux */

        #include <errno.h>
        #include <sys/wait.h>
        #include <sys/types.h>
        #include <sys/socket.h>
        #include <netinet/in.h>
        #include <netdb.h>
        #include <sys/types.h>
        #include <sys/stat.h>
        #include <fcntl.h>

#endif
```

Les types : code portable

```
#ifdef WIN32 /* Si on est sous Windows */

        #define _WIN32_WINNT 0x0501 /* Si on est pas sous windows XP et que l'on souhaite utiliser getaddrinfo() */
        #include <winsock2.h> /* En-tete Windows  */
        #include <ws2tcpip.h>

        #define closesocket(s) close(s) /* La fonction close() est closesocket(s) sous windows.*/
        /* Changements des types */
        typedef struct SOCKADDR_IN sockaddr_in;
        typedef struct SOCKADDR_IN6 sockaddr_in6;
        typedef struct SOCKADDR sockaddr;
        typedef struct IN_ADDR in_addr;

#elif defined linux /* si on est sous Linux */

        #include <errno.h> #include <sys/wait.h> #include <sys/types.h>
        #include <sys/socket.h>#include <netinet/in.h>#include <netdb.h>
        #include <sys/types.h>#include <sys/stat.h>#include <fcntl.h>

        typedef int SOCKET;

#endif
```

Windows : initialisation des librairies

- Les librairies ont besoin d’être chargées/initialisées sous windows:

```
static void init(void)
{
#ifdef WIN32
   WSADATA wsa;
   if(WSAStartup(MAKEWORD(2, 2), &wsa)<0) {fprintf(stderr,"WSAStartup a echoue !");  exit(1);
#endif
}

static void end(void)
{
#ifdef WIN32
   WSACleanup();
#endif
}

int main()
{
            SOCKET confd;
            struct sockaddr_in6 serverAddr, clientAddr;
            struct addrinfo hints, *res, *lecture;

            init();
```

Les fonctions portables

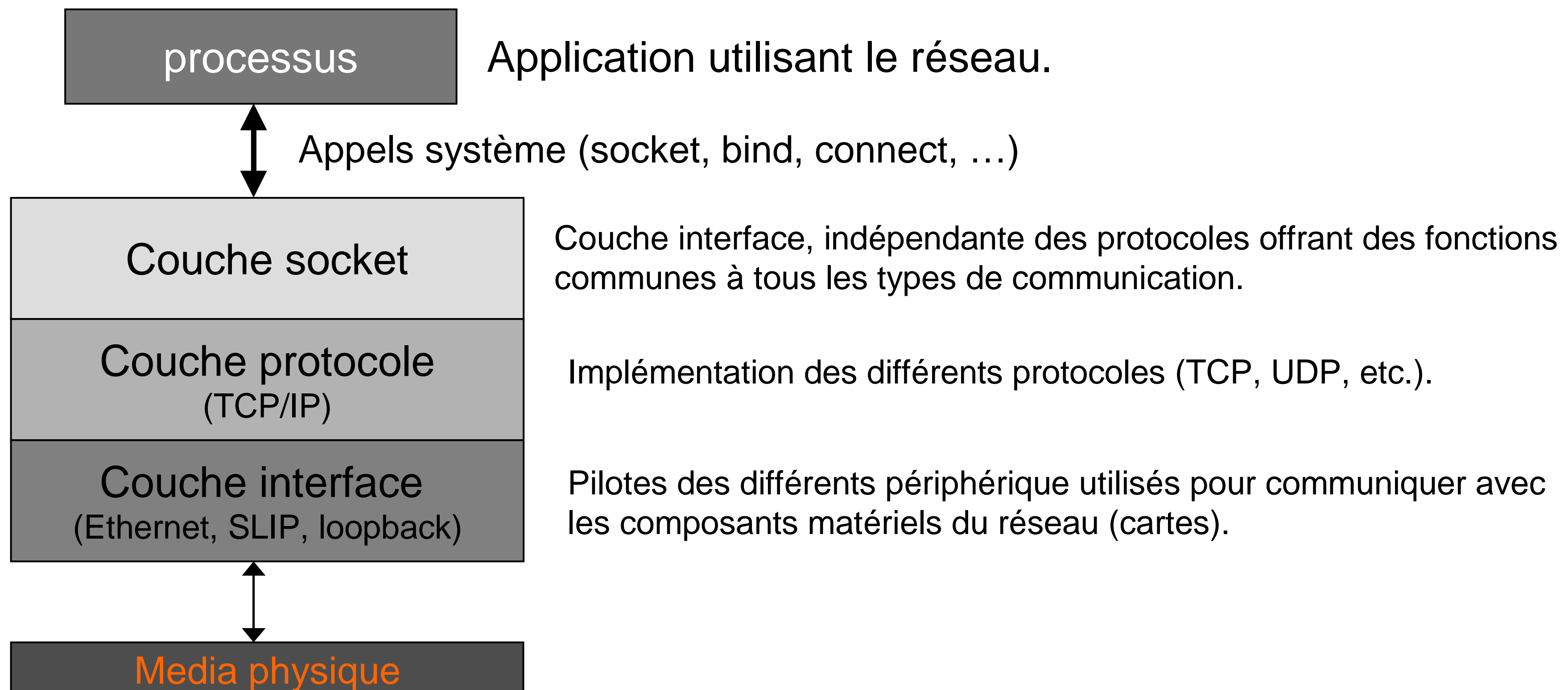
- La liste des appels systèmes portables est la suivante (non-exhaustive):
  - socket(), bind(), listen(), accept(), send(), recv(), sendto(), recvfrom()
  - getaddrinfo() avec certaines précaution.
- Celle qui ne sont pas portable ou qui demande une utilisation spécifique à windows:
  - close/closesocket(), shutdown(), ioctl, fcntl(), fork(), setsockopt().

Annexe 2

Quelques détails sur l’implémentation de la pile

Organisation générale de l’implémentation

- Les communications sous Linux sont exclusivement implémentées par le noyau.
- Le code réseau dans le noyau est organisé en trois couches.

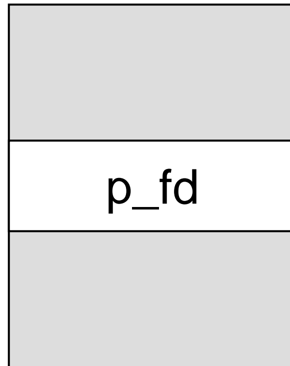


Panorama de l’implémentation réseau (1)

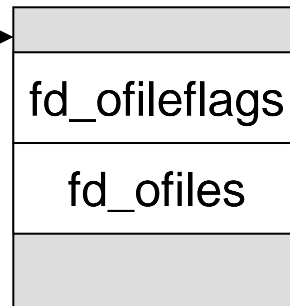
- Les communications du point de vue d’un processus sont gérés comme des fichiers
  - Émission: écriture sur un fichier
  - Réception: lecture sur un fichier
- A chaque processus est associé une table des descripteurs de fichiers.
- Il s’agit d’un tableau de pointeurs, chaque pointeur pointant indirectement sur un fichier (*v-node*) ou sur une communication (*socket*).

Panorama de l’implémentation réseau (1)

Structure décrivant le processus (struct proc)



Structure décrivant les fichiers ouverts (struct filedesc)



Drapeaux associés aux descripteurs (close_on_exec, etc.). Tableau de *char*.

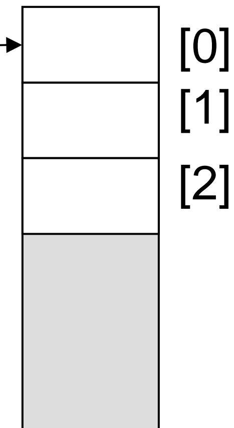
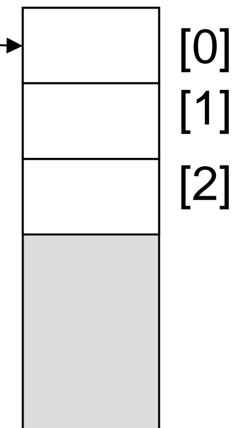
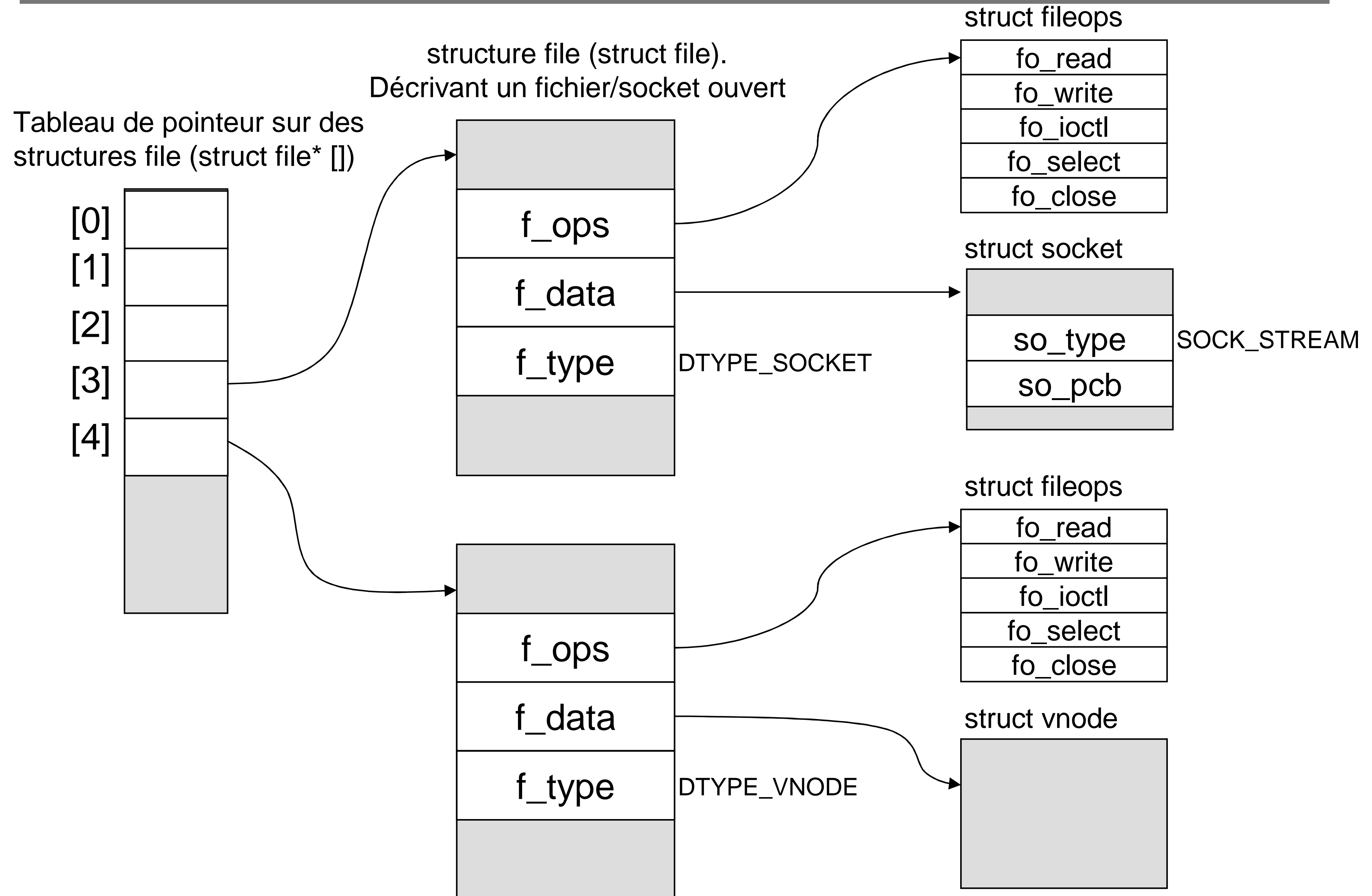


Tableau de pointeur sur des structures file (struct file* [])



Panorama de l’implémentation réseau (2)



Panorama de l'implémentation réseau (3)

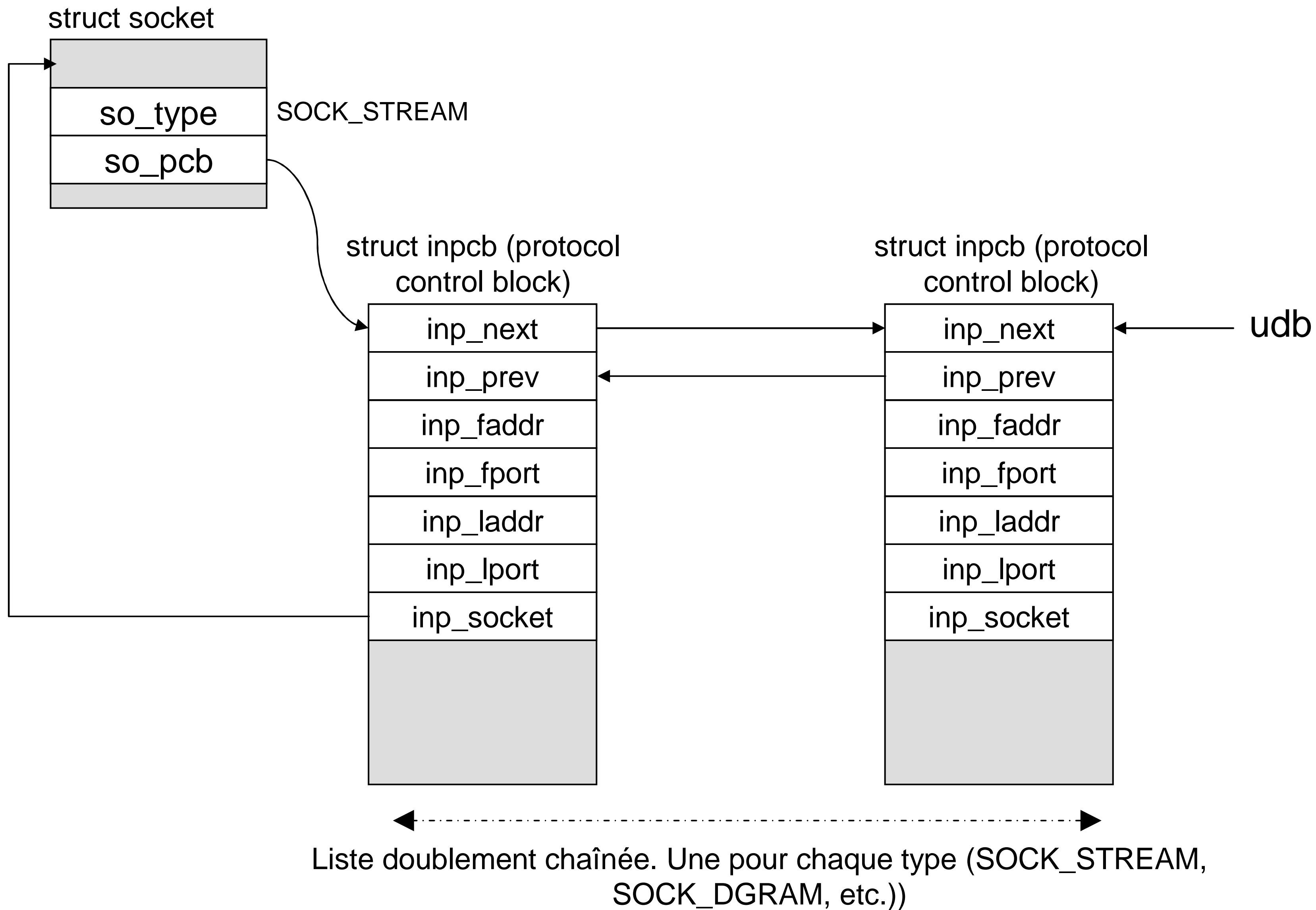
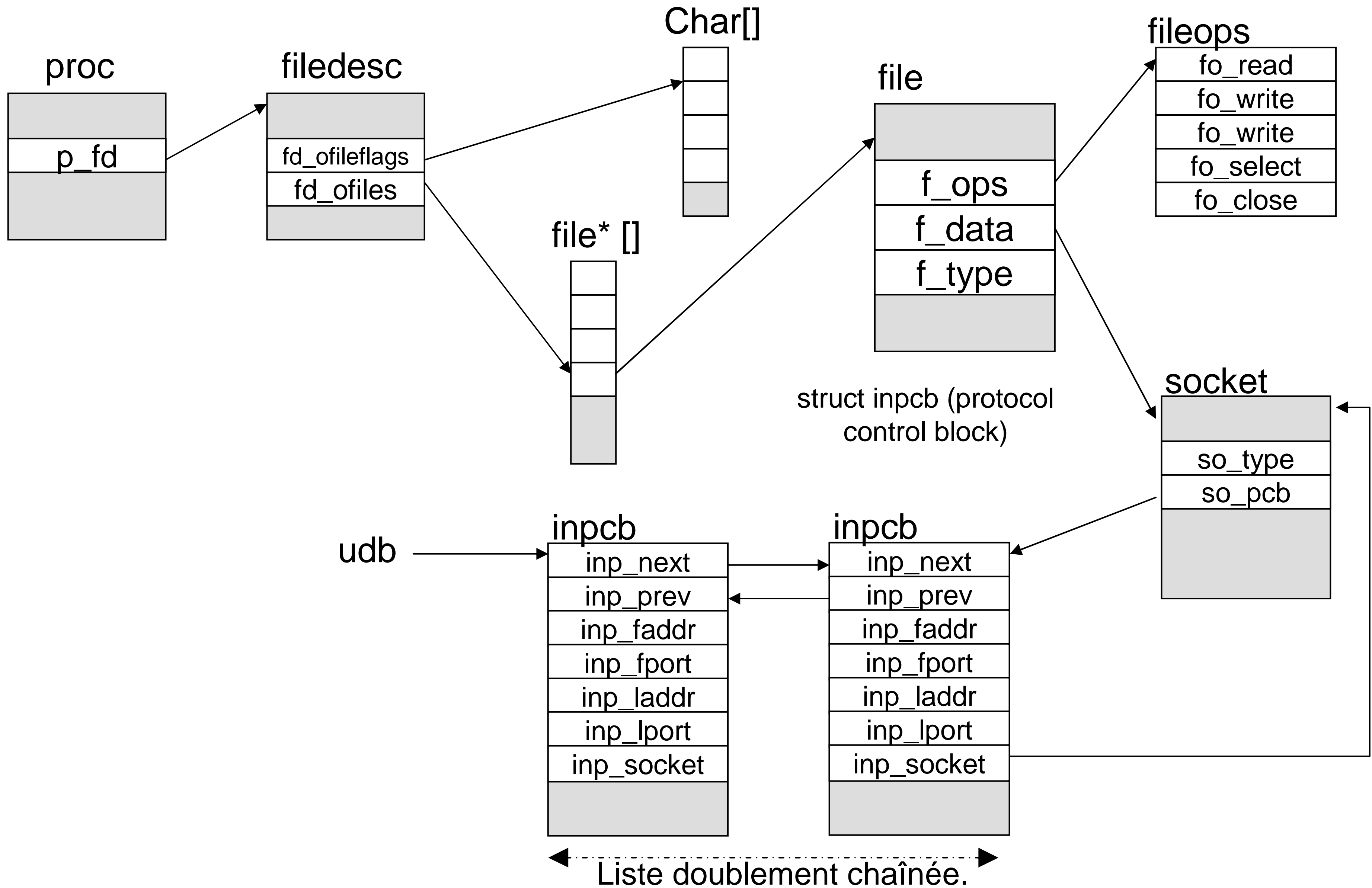
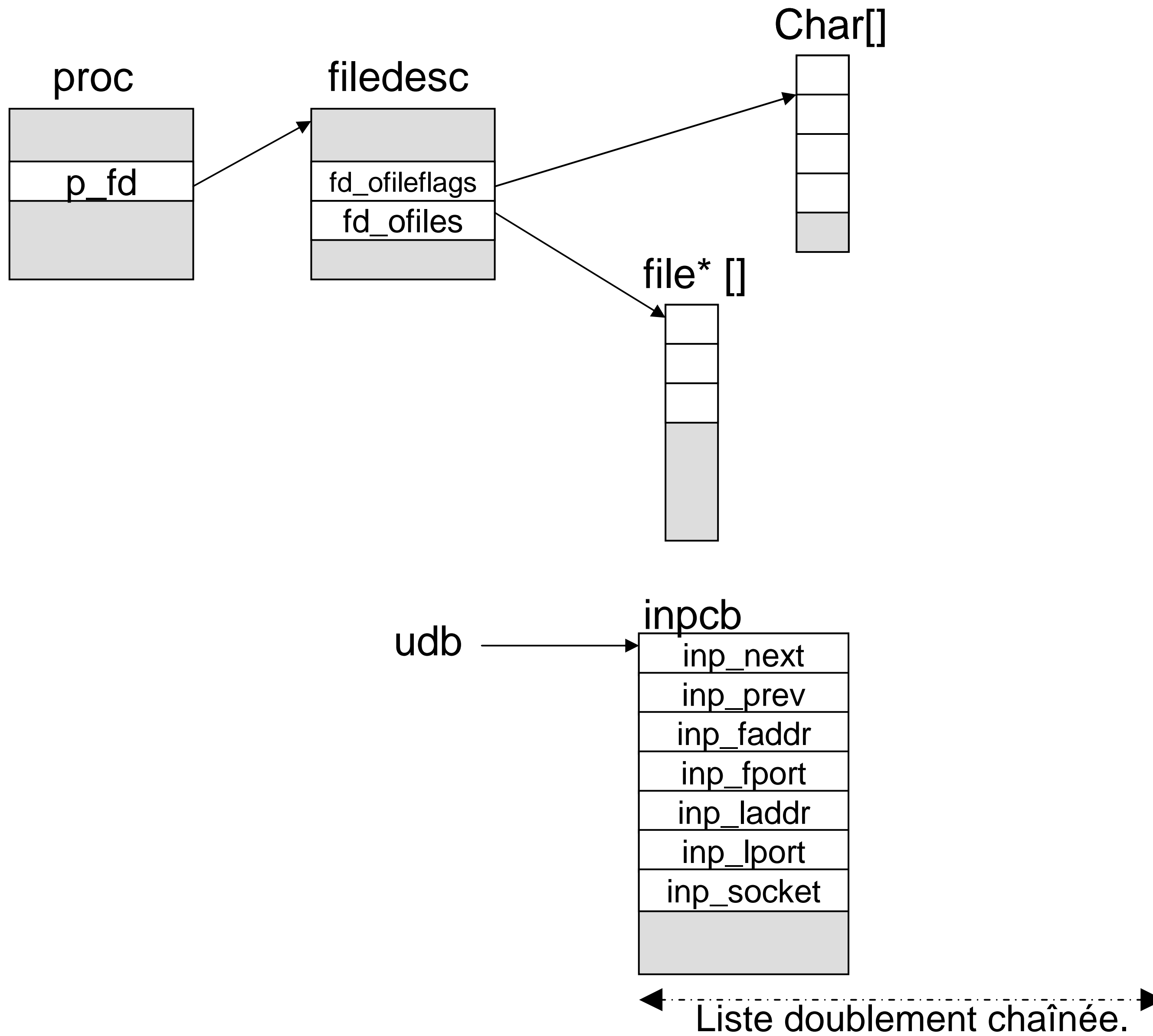


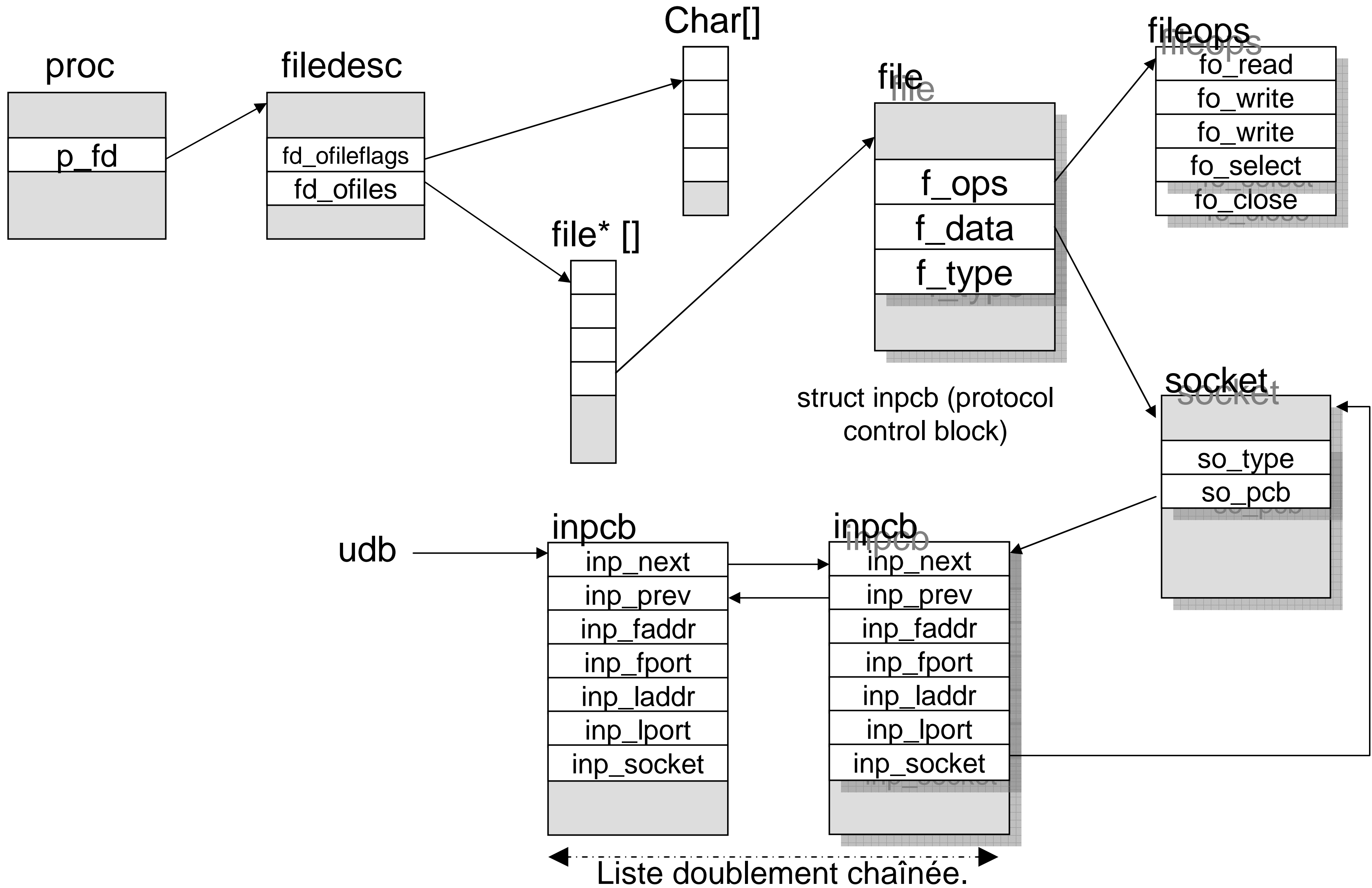
Schéma complet



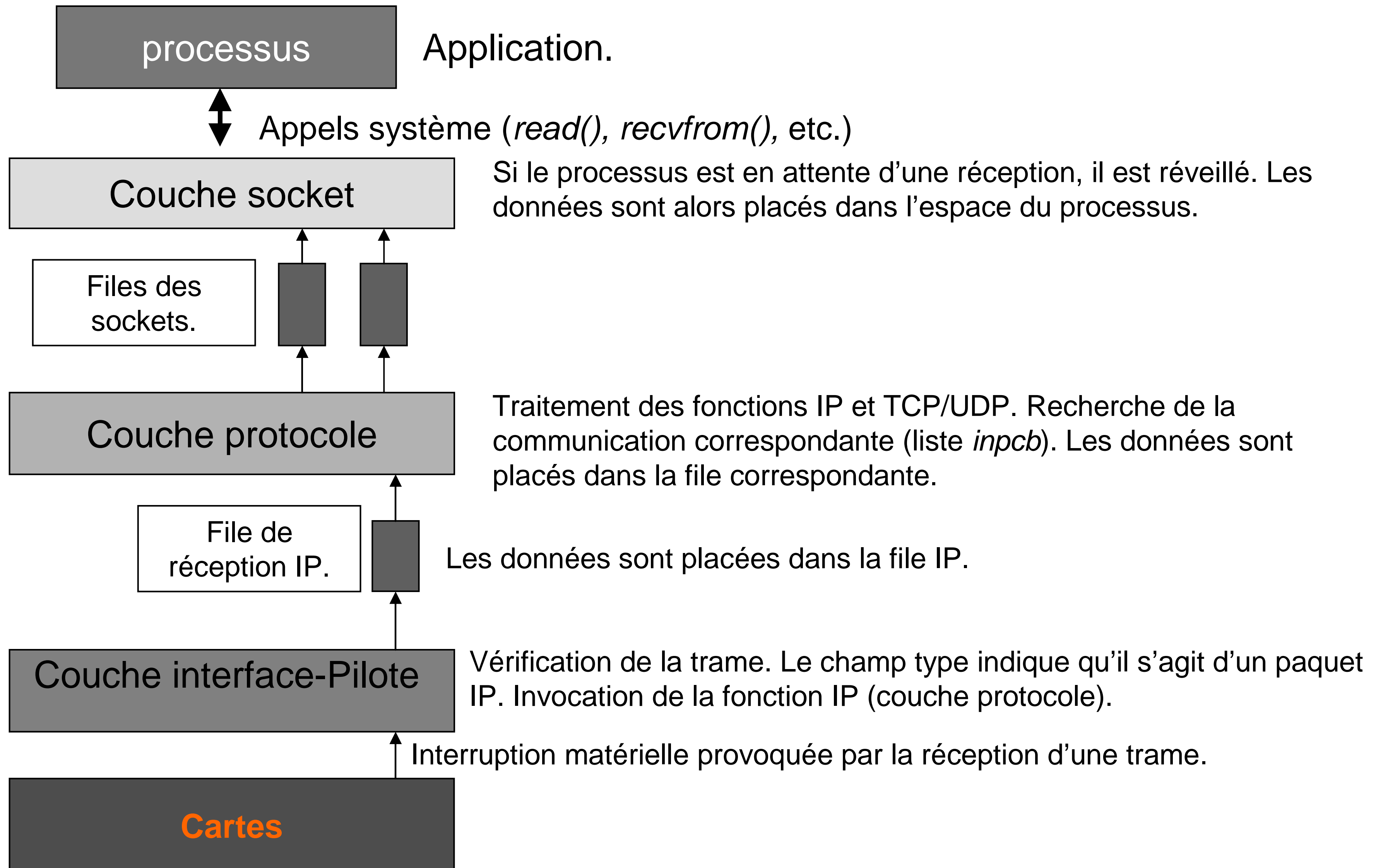
Avant l’appel à ***socket()***



Après l'appel à ***socket()***



Réception d'une trame



Emission d'une trame