Communication Inter-Processus

Aloÿs Dufour

ATER, LIPN équipe LoCal
Université Paris-Nord XIII

10 février 2026

Communication inter-processus

- Variables et fichiers communs
- Signaux
- Message
  - Tubes sans noms
  - Tubes nommés
  - Sockets

Vairables et fichiers communs

- P_1 et P_2 s'exécutent en temps partagé
- La valeur initiale de v=1.

```
P1 : v=++;
P2 : v=--;
```

- Les instructions correspondantes en assembleur sont les suivantes :

```
P1 : movl $v, %eax              #loadv
movw (%eax), %bx
incw %bx                        # bx++;
movw %bx, (%eax)                # store v
P2 : movl $v, %eax              #loadv
movw (%eax), %bx
decw %bx                        # bx--;
movw %bx, (%eax)                # store v
```

- P_1 interrompu après l'instruction `incw %bx` $\Longrightarrow$ v=2 !!

Communication inter-processus

- Accès concurrent
- Le résultat dépend de l'ordonnacement
- Problème de cohérence des données partagés (objet critique)
- Synchronisation des processus

Les signaux

- Les interruptions logicielles ou signaux sont utilisées par le système d'exploitation pour aviser les processus utilisateurs de l'occurrence d'un événement important (division par 0, une adresse non valide, ...etc)
- Ce mécanisme permet à un processus de réagir à cet événement sans être obligé d'en tester en permanence l'arrivée.
- Les processus peuvent indiquer au système ce qui doit se passer à la réception d'un signal.

Les signaux

- Ignorer le signal.

Les signaux

- Ignorer le signal.
- Appeler une routine de traitement fournie par le noyau. Cette procédure provoque normalement la mort du processus. Dans certains cas, elle provoque la création d'un fichier core, qui contient le contexte du processus avant de recevoir le signal. Ces fichiers peuvent être examinés à l'aide d'un débogueur.

Les signaux

- Ignorer le signal.
- Appeler une routine de traitement fournie par le noyau. Cette procédure provoque normalement la mort du processus. Dans certains cas, elle provoque la création d'un fichier core, qui contient le contexte du processus avant de recevoir le signal. Ces fichiers peuvent être examinés à l'aide d'un débogueur.
- Appeler une procédure spécifique créée par le programmeur. Tous les signaux ne permettent pas ce type d'action.

Les signaux

- Tous les signaux ont une routine de service, ou une action par défaut.
  - A : terminaison du processus
  - B : ignorer le signal
  - C : Créer un fichier core
  - D : Stopper le processus
  - E : La procédure de service ne peut pas être modifiée
  - F : Le signal ne peut pas être ignoré

Les signaux

- Tous les signaux ont une routine de service, ou une action par défaut.
  - A : terminaison du processus
  - B : ignorer le signal
  - C : Créer un fichier core
  - D : Stopper le processus
  - E : La procédure de service ne peut pas être modifiée
  - F : Le signal ne peut pas être ignoré

Signal	Num	Type	Description
SIGKILL	9	AEF	Termine l'exécution du processus
SIGSTOP	19	DEF	Stoppe le processus
SIGCONT	18		Continue l'exécution du processus
SIGCHILD	17	B	Un processus fils a fini
SIGUSR1	10	A	Signal d'utilisateur
SIGUSR2	12	A	Signal d'utilisateur

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

- `kill()` retourne -1 en cas d'erreur, et 0 autrement. `pid` peut prendre comme valeurs :

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

- `kill()` retourne -1 en cas d'erreur, et 0 autrement. `pid` peut prendre comme valeurs :
  - Si `pid > 0` : processus `pid`.

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

- `kill()` retourne -1 en cas d'erreur, et 0 autrement. `pid` peut prendre comme valeurs :
  - Si `pid > 0` : processus `pid`.
  - Si `pid = 0` : groupe de l'émetteur.

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

- `kill()` retourne -1 en cas d'erreur, et 0 autrement. `pid` peut prendre comme valeurs :
  - Si `pid > 0` : processus `pid`.
  - Si `pid = 0` : groupe de l'émetteur.
  - Si `pid = -1` : tous les processus (seulement root peut le faire).

Les signaux

- Dans le cas du système Unix/Linux, un processus utilisateur peut envoyer un signal à un autre processus (deux processus d'un même propriétaire, ou bien émetteur du signal = `root`). Les services `Posix` de gestion des signaux utilisent la bibliothèque `<signal.h>`.
- L'envoi d'un signal peut se faire avec l'appel système `kill()` :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid t pid, int signal);
```

- `kill()` retourne -1 en cas d'erreur, et 0 autrement. `pid` peut prendre comme valeurs :
  - Si `pid > 0` : processus `pid`.
  - Si `pid = 0` : groupe de l'émetteur.
  - Si `pid = -1` : tous les processus (seulement root peut le faire).
  - Si `pid < 0` : au groupe `gid`= |`pid`|

Les signaux

- Gestion des signaux
- deux possibilités
  - `signal` : standard `C`
  - `sigaction` : `POSIX` (voir tp)

Les signaux : exemples

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void sighandler(int signum)
{
    printf("Masquage du signal SIGTERM\n");
}

int main(void)
{
    char buffer[256];
    if (signal(SIGTERM, &sighandler) == SIG_ERR) {
        printf("Ne peut pas manipuler le signal\n");
        exit(1);
    }
    while (1) {
        fgets(buffer, sizeof(buffer), stdin);
        printf("Input: %s", buffer);
    }
    return 0;
}
```

Les signaux : exemples

```c
#include <stdio.h>
#include <signal.h>
#include <unistd.h>
void sighandler(int signum)
{
    printf("Masquage du signal SIGTERM\n");
}

int main(void)
{
    char buffer[256];
    if (signal(SIGTERM, &sighandler) == SIG_ERR) {
        printf("Ne peut pas manipuler le signal\n");
        exit(1);
    }
    while (1) {
        fgets(buffer, sizeof(buffer), stdin);
        printf("Input: %s", buffer);
    }
    return 0;
}
```

- `kill pid` envoi SIGTERM
- `kill -SIGKILL pid` est non ignorable et tue le processus.

Les signaux : sleep() et pause()

- ```c
  #include <unistd.h>
  int pause(void);
  int sleep(unsigned int seconds);
  ```
  ```

Les signaux : sleep() et pause()

- ```c
  #include <unistd.h>
  int pause(void);
  int sleep(unsigned int seconds);
  ```
- `sleep()` permet à un processus de se mettre en veille et attendre l'arrivée d'un signal particulier pour exécuter le gestionnaire, ou à un émetteur de se synchroniser avec le récepteur.
  ```

Les signaux : sleep() et pause()

- ```
  #include <unistd.h>
  int pause(void);
  int sleep(unsigned int seconds);
  ```
- `sleep()` permet à un processus de se mettre en veille et attendre l'arrivée d'un signal particulier pour exécuter le gestionnaire, ou à un émetteur de se synchroniser avec le récepteur.
- `pause()` permet que le processus passe à l'état suspendu. Il va quitter cet état au moment de recevoir un signal. Après avoir reçu le signal, le processus recevra un traitement adéquat associé au signal et continuera son exécution à l'instruction qui suit `pause()`.
  ```

Les signaux : exemple

```c
void trait_sigint()
{
    printf("bien recu SIGINT, mais j'osef\n");
}
void trait_sigquit()
{
    printf("bien recu SIGQUIT, mais j'osef\n");
}
int main()
{
    int pid = fork();
    if (pid == 0) {
        signal(SIGINT, trait_sigint);
        signal(SIGQUIT, trait_sigquit);
        printf("je suis toujours en vie\n");
        sleep(20);
    }
    else {
        printf("premier reveil du fils\n");
        sleep(120);
        printf("second reveil du fils\n");
        sleep(500);
        printf("troisieme reveil du fils\n");
    } else {
        sleep(1);
        kill(pid, SIGINT);
        sleep(2);
        kill(pid, SIGQUIT);
        sleep(5);
        kill(pid, SIGKILL);
    }
    return 0;
```

Les signaux : trap

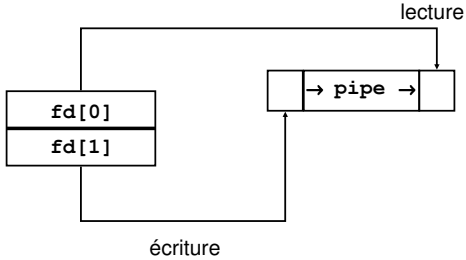
Dans les scripts shell, la commande `trap` permet d'attraper des signaux et d'exécuter des commande ou fonction.
Voir le manuel.

Les tubes

- Tubes sans nom
  - Les tubes sans nom sont des liaisons unidirectionnelles de communication.
  - La taille maximale d'un tube sans nom varie d'une version à une autre d'Unix, mais elle est approximativement égale à 4 Ko.
  - Les tubes sans nom peuvent être créés par le shell (`|`) ou par l'appel système `pipe()`.
    - `int pipe(int descripteur[2]);`
    - `descripteur[0]` : accès en lecture et `descripteur[1]`a : accès en écriture.
    - Seul le processus créateur du tube et ses descendants (ses fils) peuvent accéder au tube.
    - Si le système ne peut pas créer de tube pour manque d'espace, `pipe()` retourne -1, sinon 0.

Les tubes sans nom

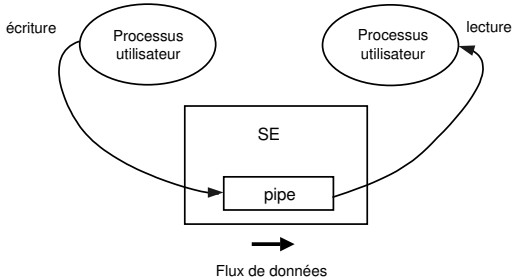
- ```
  int fd[2];
  pipe(fd);
  ```



- Communication père fils
  ```

Les tubes sans nom

- Le processus père crée un tube de communication sans nom en utilisant l'appel système `pipe()`
- Le processus père crée un ou plusieurs fils en utilisant `fork()`
- Le processus écrivain ferme l'accès en lecture du tube
- De même, le processus lecteur ferme l'accès en écriture du tube
- Les processus communiquent en utilisant les appels système `write()` et `read()`
- Chaque processus ferme son accès au tube s'il veut mettre fin à la communication.

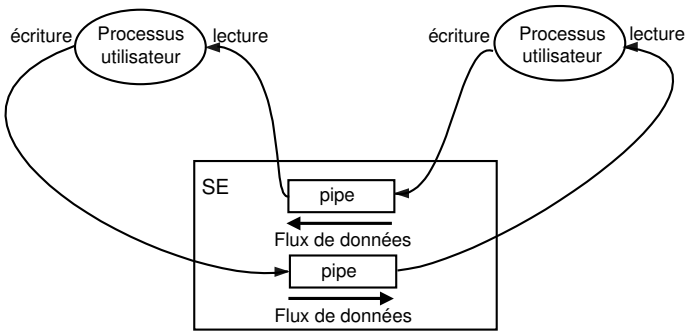


- Communication père fils

Les tubes sans nom : exemple

```c
#include <sys/types.h>
#include <unistd.h>  /* fork , pipe , read , write , close */
#include <stdio.h>
#define R 0
#define W 1
int main()
{
    int nboctets, fd[2];
    char *message[256];
    char phrase = "message envoye au pere par le fils";
    pipe(fd);
    if (fork() == 0) {
        close(fd[R]);
        write(fd[W], phrase, strlen(phrase) + 1);
        close(fd[W]);
    } else {
        close(fd[W]);
        nboctets = read(fd[R], message, 256);
        printf(" Lecture %d octets : %s\n", nboctets, message);
        close(fd[R]);
    }
    return 0;
}
```

Les tubes sans nom : communication bidirectionnelle



Les tubes nommés

- FIFO
- Chacun ont un nom qui existe dans le système de fichiers.
- Considérés comme des fichiers spéciaux.
- Ils peuvent être utilisés par des processus indépendants, à condition qu'ils s'exécutent sur une même machine.
- Ils existeront jusqu'à ce qu'ils soient supprimés explicitement.
- Capacité plus grande, d'environ 40 Ko.

Les tubes nommés : création

La commande shell `mkfifo`

- `mkfifo tube`
- `chmod g+rw tube`

L'appel système

```
int mkfifo(char *nom, mode_t mode);
```

```
if (mkfifo(nom_tube, 0666) != 0) {
    printf("Impossible de creer %s\n", nom_tube);
    exit(1);
}
```

Les tubes nommés : exemple

writer.c

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main(void)
{
    int fd;
    char message[256];
    sprintf(message, "bonjour du writer [%d]\n", getpid());
    fd = open("tuyau", O_WRONLY);
    printf(" ici writer[%d] \n", getpid());
    if (fd != 1) {
        write(fd, message, strlen(message));
    } else
        printf("desole, le tube n'est pas disponible\n");
    close(fd);
    return 0;
}
```

Les tubes nommés : exemple

reader.c

```c
#include <unistd.h>
#include <stdio.h>
#include <fcntl.h>
int main()
{
    int fd, n;
    char input;
    fd = open("tuyau", O_RDONLY);
    printf("ici reader[%d] \n", getpid());
    if (fd != 1) {
        printf("Recu par le lecteur:\n");
        while ((n = read(fd, &input, 1)) > 0) {
            printf("%c", input);
        }
        printf("Le lecteur se termine!\n", n);
    } else
        printf("desole, le tube n'est pas disponible\n");
    close(fd);
    return 0;
}
```

Les tubes nommés : exemple

- un writer et un reader

```
> mkfifo tuyau
> gcc -o writer writer.c
> gcc -o reader reader.c
> writer& reader&
ici reader[5831]
ici writer[5830]
Recu par le lecteur: bonjour du writer [5830]
Le lecteur se termine!
```

- deux writer et un reader

```
> ./writer & ./writer & ./reader &
ici reader[5827]
ici writer[5826]
ici writer[5825]
Recu par le lecteur: bonjour du writer [5826]
Recu par le lecteur: bonjour du writer [5825]
Le lecteur se termine!
```

Rappels des prototypes

```c
#include <sys/types.h> /* Typedefs : mode_t, ssize_t, pid_t, off_t, …      */
#include <fcntl.h>     /* Symbolic constants : F_DUPFD… O_CREAT… O_RDONLY… */
int open(const char *pathname, int flags, ...  /* mode_t mode */ );
int creat(const char *pathname, mode_t mode);

#include <unistd.h>
extern char **environ;
int execl  (const char *pathname, const char *arg, ...  /*, (char *) NULL */);
int execvpe(const char *file, char *const argv[], char *const envp[]);
ssize_t read(int fd, void buf[.count], size_t count);
ssize_t write(int fd, const void buf[.count], size_t count);
off_t lseek(int fd, off_t offset, int whence);
int close(int fd);
int dup(int oldfd);
int dup2(int oldfd, int newfd);
pid_t fork(void);
pid_t getpid(void);
int pause(void);
int pipe(int [2]);

#include <sys/wait.h>
pid_t wait(int *_Nullable wstatus);

#include <signal.h>
int kill(pid_t pid, int sig);
void (*signal(int, void (*)(int)))(int);
```

Exercices

1. À l'aide de `pipe, dup2, fork`, écrire un petit programme C qui fait la même chose que `$ ls / | grep u`
2. Écrire une commande `tuyaunul cmd1 cmd2 [args]` qui utilise un fichier en guise de communication inter-processus, pour stocker temporairement la sortie standard de la première commande, la deuxième commande l'utilisant comme entrée standard.
3. Écrire une commande `vraituyau cmd1 cmd2 [args]` qui met en place un tube entre deux processus pour relier la sortie de `cmd1` à l'entrée de `cmd2` (en faisant en sorte qu'elle retourne la valeur de sortie `cmd1` lorsque celle-ci n'est pas null, celle de `cmd2` sinon).
4. On souhaite avoir la commande
   `log file cmd [args]`
   à disposition, qui récupère une copie de la sortie standard de `cmd [args]` dans `file`.
   L'implémenter en C, puis en shell.
5. Communication bidirectionnelle : écrire deux programmes, un qui génère un nombre secret, et l'autre qui le devine (celui qui fait deviner peut même donner des indications à celui qui devine), le tout, avec une communication transparente (on voit ce qu'il se passe dans le terminal).

Sockets de Berkeley

Tradition Unix : tout est un fichier.

Sockets de Berkeley

Tradition Unix : tout est un fichier.
Communication réseau (couche de transport) via des fichiers : les *sockets*.

Rapides rappels réseaux — Généralités

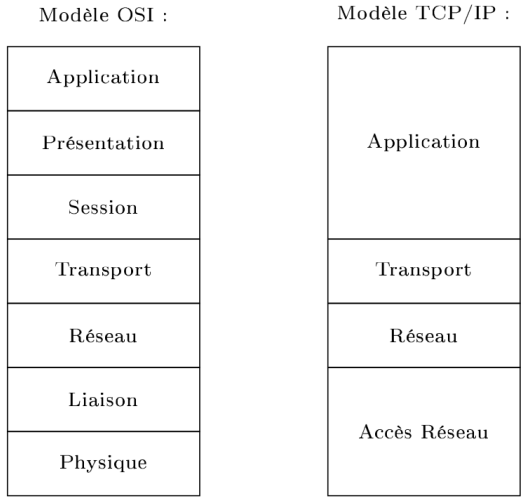
Réseau : ensemble de nœuds interconnectés (a une topologie en tant que graphe).
Exemples : LAN, WAN, réseaux téléphoniques (GSM, 3-5G), de distribution d'eau, routier, "Internet"...

Rapides rappels réseaux — Généralités

Réseau : ensemble de nœuds interconnectés (a une topologie en tant que graphe).
Exemples : LAN, WAN, réseaux téléphoniques (GSM, 3-5G), de distribution d'eau, routier, "Internet"...

Internet : réseau de réseaux, à commutation de paquets,
+ protocole de communication TCP/IP,
+ DNS.

Rapides rappels réseaux — Modèles par couches



OSI (Open Systems Intectonnection)
TCP/IP (Transmission Control Protocol / Internet Protocol)

Rapides rappels réseaux — Modèles et Protocoles

Chaque couche a ses protocoles :

- Accès réseau : Ethernet, IEEE 802, Bluetooth
- Réseau : ARP, ICMP, RIP, IP
- Transport : TCP, UDP, QUIC, RTP
- Application : HTTP, IMAP, SMTP, FTP, SSH, ...
- Entre deux chaises : BGP, DHCP, DNS

Rapides rappels réseaux — IP

Encapsulation : chaque protocole de la couche n communique via un protocole de la couche $n-1$.

Exemple : TCP communique à travers IP (vérification des données via CRC...),

En-tête Ethernet	En-tête IP	En-tête TCP	Message TCP

Rapides rappels réseaux — IP

Encapsulation : chaque protocole de la couche n communique via un protocole de la couche $n-1$.

Exemple : TCP communique à travers IP (vérification des données via CRC...),

En-tête Ethernet	En-tête IP	En-tête TCP	Message TCP

Pour IP en particulier, deux versions :

- IPv4 : `81.194.43.244`, $255^4 = 2^{32}$ addresses disponibles, plus assez...
- IPv6 : `fe80::e6f8:9cff:fe67:ee22`, 2^{128} addresses possibles.

Chaque réseau a une adresse, et chaque machine dans le réseau également : Classless Inter-Domain Routing (CIDR), combien de *bits* utilise le réseau (détermine sa "taille"). Addresses réserervées (privées/publiques), masque de sous-réseau, NAT...

Rapides rappels réseaux — IP

Encapsulation : chaque protocole de la couche n communique via un protocole de la couche $n-1$.
Exemple : TCP communique à travers IP (vérification des données via CRC...),

En-tête Ethernet	En-tête IP	En-tête TCP	Message TCP

Pour IP en particulier, deux versions :

- IPv4 : `81.194.43.244`, $255^4 = 2^{32}$ addresses disponibles, plus assez...
- IPv6 : `fe80::e6f8:9cff:fe67:ee22`, 2^{128} addresses possibles.

Chaque réseau a une adresse, et chaque machine dans le réseau également : Classless Inter-Domain Routing (CIDR), combien de *bits* utilise le réseau (détermine sa "taille").
Addresses réserervées (privées/publiques), masque de sous-réseau, NAT...

Transport des paquets IP : *routage & routeurs*.
Routage hiérarchique, algorithmes de flot ou de plus court chemin dans un graphe (Dijkstra, Bellman-Ford).

Algorithmes de contrôle de congestion (Jacobson, 1988).

Rapides rappels réseaux — Internet

En plus des aspects techniques (câbles, routeurs, TCP/IP, DNS), il y a les aspects :

- juridiques,
- politiques,
- écologiques,
- sociaux.

Réseaux de réseaux : *systèmes autonomes* (AS),
Routage entre réseaux : *border gateway protocol* (BGP), et géopolitique.

Sockets : appels et types

Extrémité de communication pour un *processus*.
Canal potentiellement bidirectionnel (un processus peut y lire et y écrire) pour la communication entre processus :

- sur la même machine (sockets Unix, `AF_UNIX`) ou
- sur des machines potentiellement différentes connectées sur un même réseau, avec protocole réseau IPv4 (sockets ipv4, `AF_INET`) ou
- sur des machines potentiellement différentes connectées sur un même réseau, avec protocole réseau IPv6 (sockets IPv6, `AF_INET6`).

`AF` (address family).

[1] il faut être `root` pour pouvoir créer ces sockets

Sockets : appels et types

Extrémité de communication pour un *processus*.
Canal potentiellement bidirectionnel (un processus peut y lire et y écrire) pour la communication entre processus :

- sur la même machine (sockets Unix, `AF_UNIX`) ou
- sur des machines potentiellement différentes connectées sur un même réseau, avec protocole réseau IPv4 (sockets ipv4, `AF_INET`) ou
- sur des machines potentiellement différentes connectées sur un même réseau, avec protocole réseau IPv6 (sockets IPv6, `AF_INET6`).

`AF` (address family).
Un socket peut être de type :

- `SOCK_RAW` pour un accès « brut » au réseau (paquets IP)[1]
- `SOCK_STREAM` pour un *flux* de données bidirectionnel et fiable, bref... TCP
- `SOCK_DGRAM` pour des *datagrammes*, bref... UDP

[1] il faut être `root` pour pouvoir créer ces sockets

Créer un socket

```c
#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
int socketpair(int domain, int type, int protocol, int socket_vector[2]);
```

`DOMAIN` la famille d'adresse (`AF_INET`, ...)

`TYPE` du socket (`SOCK_STREAM` pour TCP et `SOCK_DGRAM` pour UDP)

`PROTOCOL` toujours 0 car on se restreint à TCP et UDP.

- Valeur de retour : le descripteur de fichier.

Sockets : Exemples

- Socket TCP/IPv4

```
int sock = socket(AF_INET, SOCK_STREAM, 0);
if (sock < 0) {
    perror("socket");
    exit(1);
}
```

- Socket UDP/IPv6

```
int sock = socket(AF_INET6, SOCK_DGRAM, 0);
if (sock < 0) {
    perror("socket");
    exit(1);
}
```

Appel système `bind`

Une fois le socket créé, il faut, au moins côté serveur :

- préciser *le port* utilisé par le serveur pour recevoir les messages,
- si besoin, restreindre la réception des messages à ceux destinés à une certaine adresse IP.
- Par exemple, un serveur pourrait ne vouloir recevoir que les messages venant de l'interface *loopback* et se restreindre aux adresses `127.0.0.1/8`.
- Mais le plus souvent, le socket d'un serveur sera attaché à *toutes* les interfaces réseaux de la machine.
- Enfin, en général, il n'est pas nécessaire de `bind` explicitement un socket côté client.

Appel système bind, en pratique

```c
#include <sys/socket.h>
int bind(int sockfd, const struct sockaddr *addr, socklen_t addrlen);
```

`SOCKFD` le socket en question

`ADDR` l'adresse en mémoire d'une structure contenant les informations (port, adresses...) nécessaires

`ADDRLEN` la taille de cette structure (les adresses IPv4 et IPv6 n'ont pas la même taille donc il faut préciser).

- Valeur de retour : 0 si ok, -1 sinon (avec `errno` mis à jour).

Les sockaddr (IPv4)

- Le second paramètre de `bind` est de type `struct sockaddr`.
- Destiné essentiellement à contenir une adresse IP (v4 ou v6) et un numéro de port.
- Problème : ces adresses sont très différentes... on a donc une structure « abstraite » (et assez inutile) :

```
struct sockaddr {
    sa_family_t sa_family;
    char        sa_data[14];
};
```

- Pour IPv4, la structure concrète (qui sera ensuite castée) est

```
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 */
};
```

- Voir `man 7 ip`.

Les sockaddr (IPv6)

```
struct sockaddr_in6 {
    sa_family_t     sin6_family;   /* AF_INET6 */
    in_port_t       sin6_port;     /* numéro de port */
    uint32_t        sin6_flowinfo; /* information de flux IPv6 */
    struct in6_addr sin6_addr;     /* adresse IPv6 */
    uint32_t        sin6_scope_id; /* Scope ID (nouveauté 2.4) */
};

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

Voir `man ipv6`.

Les sockaddr en pratique

Quelques pratiques courantes :

- Utiliser la fonction

```
#include <arpa/inet.h>
int inet_pton(int af, const char *src, void *dst);
```

  pour passer d'une représentation sous forme de chaîne de caractère à une `struct in_addr` (pour IPv4) ou une `struct in6_addr` (pour IPv6) directement dans le bon boutisme.
- Utiliser `htons` pour convertir le numéro de port, si besoin, en gros-boutiste.
- Utiliser les constantes `INADDR_ANY` (`0.0.0.0`, n'importe quelle interface IPv4 du serveur), `INADDR_LOOPBACK` (`127.0.0.1`) avec `htonl`.
- Pour les `sockaddr` distantes, utiliser `getaddrinfo`.

Lire dans un socket UDP

Appel système bloquant par défaut :

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

SOCKFD descripteur de fichier du socket

BUF à quelle adresse mémoire du processus écrire le datagramme reçu

LEN combien d'octets *au maximum* on peut recevoir

FLAGS 0 pour l'instant

SRC_ADDR pointeur vers une `sockaddr` que le système remplit avec l'adresse et le port *de l'expéditeur*, pour pouvoir éventuellement lui répondre

ADDR_LEN pointeur vers un entier (attention, paramètre lecture/écriture), en lecture, la taille de ce qui est pointé par `src_addr`, en écriture la taille de la `sockaddr` reçue dans le message.

Valeur de retour : Nombre d'octets effectivement reçus, ou -1 en cas d'erreur.

Écrire dans un socket UDP

Appel système bloquant par défaut :

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

`SOCKFD`	descripteur de fichier du socket
`BUF`	adresse du premier octet (dans la mémoire du processus) du message à envoyer
`LEN`	longueur du message, en octets
`FLAGS`	0 pour l'instant
`SRC_ADDR`	pointeur vers une `sockaddr` qui contient l'adresse et le port *du destinataire*.
`ADDR_LEN`	taille de la `sockaddr` précédente

Valeur de retour : Nombre d'octets effectivement écrits (en général, pour UDP, autant que `len`), ou -1 en cas d'erreur.