API socket et programmation réseau

Généralités
Sockets locales et Internet
Client/serveur, scrutation passive
Programmation multi-processus
Programmation multi-threads

Fabrice Harrouet
École Nationale d'Ingénieurs de Brest
harrouet@enib.fr
http://www.enib.fr/~harrouet/

Les *sockets* ...

- **Procédé de communication très général**
  - Communication locale à la machine ou avec des machines distantes
    - Accessibles à travers une API homogène
  - Conçu à l'origine dans *4.2BSD* mais adopté sur tous les systèmes
    - Non seulement les principes mais également l'API
    - Y compris sous *Window$* ! (à quelques détails près)
  - Exploite l'abstraction *descripteur de fichier*
    - Très cohérent vis-à-vis de la conception des systèmes (lecture/écriture, scrutation, partage, duplication ...)
    - Dissimule à l'utilisateur les subtilités sous-jacentes (échanges de trames réseau, gestion de tampons ...)

Les *sockets* ...

- **Création d'une** *socket* **(`man 2 socket`)**
  - `#include <sys/types.h>`
    `#include <sys/socket.h>`
    `int socket(int domain,int type,int proto);`
  - `domain` : famille de protocoles (local, *IP*, *IPv6*, noyau ...)
    - `PF_LOCAL`, `PF_INET`, `PF_INET6`, `PF_NETLINK` ...
  - `type` : manière de délivrer les données
    - `SOCK_DGRAM` : échange de messages unitaires et distincts
    - `SOCK_STREAM` : flot continu d'octets
    - éventuellement des variantes (assez rare)
  - `proto` : choix d'une protocole de communication
    - Utile si plusieurs possibilités pour `domain+type`, généralement `0`
  - Retour : descripteur de fichier si OK, `-1` si erreur (consulter `errno`)

Les *sockets* ...

- **Opérations sur une** *socket*
  - Semblable à la plupart des descripteurs de fichier (voir le module *SE1*)
    - Synchronisation par `fsync()`
    - Fermeture par `close()`
    - Lecture/écriture par `read()`/`write()` (descripteur bidirectionnel !) (selon le `type` choisi, d'autres appels peuvent être nécessaires)
    - Duplication par `dup()`/`dup2()`
    - Paramétrage par `fcntl()` et `ioctl()`
    - Scrutation passive par `select()`
    - Consultation des attributs par `fstat()`
    - **PAS** d'accès aléatoire avec `lseek()` ou `mmap()` !
  - Quelques opérations spécifiques
    - `shutdown()`, `bind()`, `listen()`, `accept()`, `connect()` ...

Les *sockets* ...

- **Le type** `SOCK_DGRAM`
  - Échange de messages unitaires et distincts
    - Les limites de messages sont préservées
    - Les messages ont une taille maximale (laquelle ?)
    - La délivrance n'est pas nécessairement garantie (selon `domain`)
    - L'ordre d'arrivée n'est pas nécessairement garanti (selon `domain`)
  - Il n'y a pas de connexion, il faut préciser la destination à chaque envoi
    - Permet l'envoi à des destinations variées
    - Permet la reception depuis des sources variées
    - Possibilité de réaliser une *pseudo-connexion* (confort)
      - Permet d'indiquer la destination une fois pour toutes
      - On ne peut alors recevoir que ce qui provient de cette source

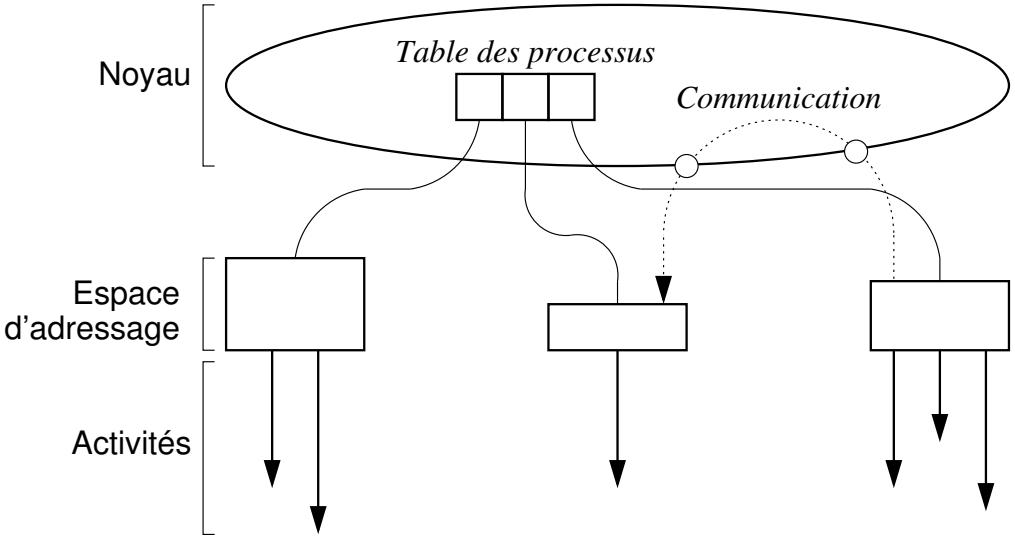
Les *sockets* ...

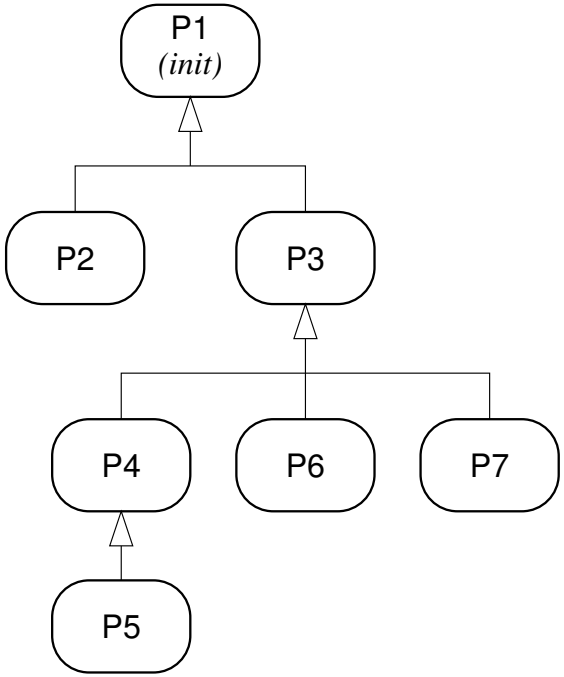
- **Le type** `SOCK_STREAM`
  - Flot continu d'octets
    - Semblable à un *tube* de communication (`pipe`) mais bidirectionnel
    - Les limites ne sont pas préservées (concaténation/découpage)
      (ex : N octets en E envois → N octets en R receptions, E ≠ R)
    - La délivrance est garantie
    - L'ordre d'arrivée est garanti
  - La communication n'a lieu qu'après une étape de *connexion*
    - Les deux extrémités sont identifiées une fois pour toutes
  - Les rôles *client* et *serveur* sont généralement distincts
    - Un client est connecté à un serveur
    - Un serveur peut être connecté simultanément à plusieurs clients

Les *processus* ...

- **Notion centrale du système**
- **Un programme en cours d'exécution**
  - Le programme exécutable
  - Une activité (ou plusieurs)
  - Des données (plusieurs zones mémoire)
  - Un ensemble de registres
  - Des propriétés (environnement, répertoire courant, utilisateur ...)
  - Des moyens de communication
- **Membre d'une hiérarchie**
  - Un parent
  - Zéro, un ou plusieurs enfants

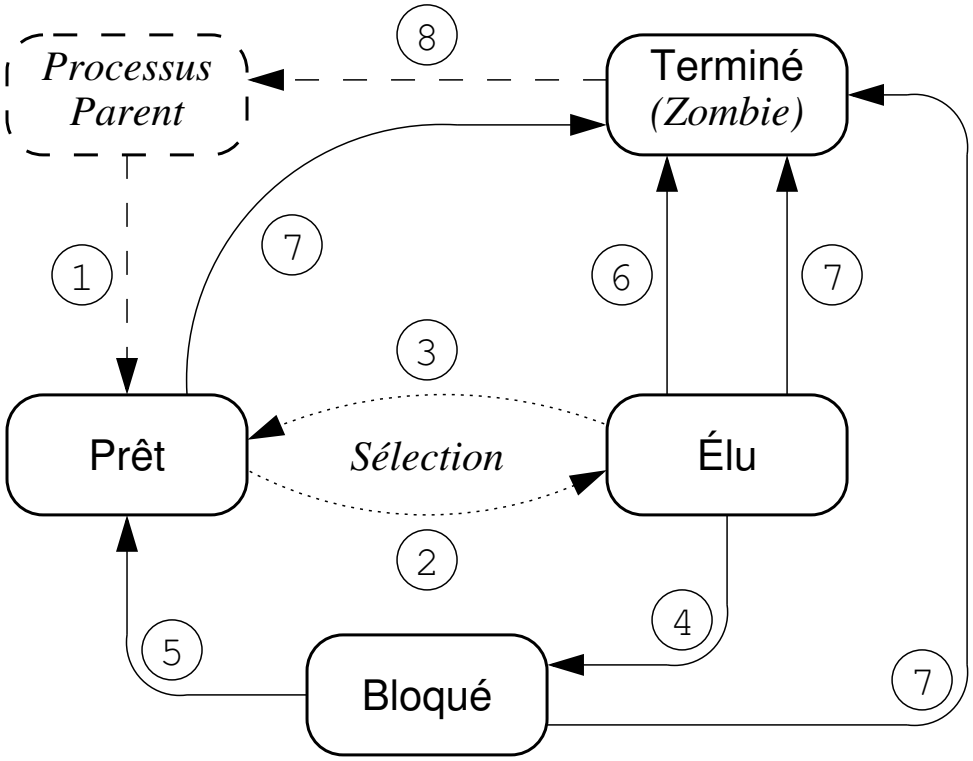
Les *processus* ...





Les *processus* ...

▷ Cycle de vie



1 : *Création*

2 : *Utilisation du processeur*

3 : *Un autre utilise le processeur*

4 : *Suspension ou attente de ressource*

5 : *Relance ou ressource disponible*

6 : *Terminaison normale*

7 : *Destruction*

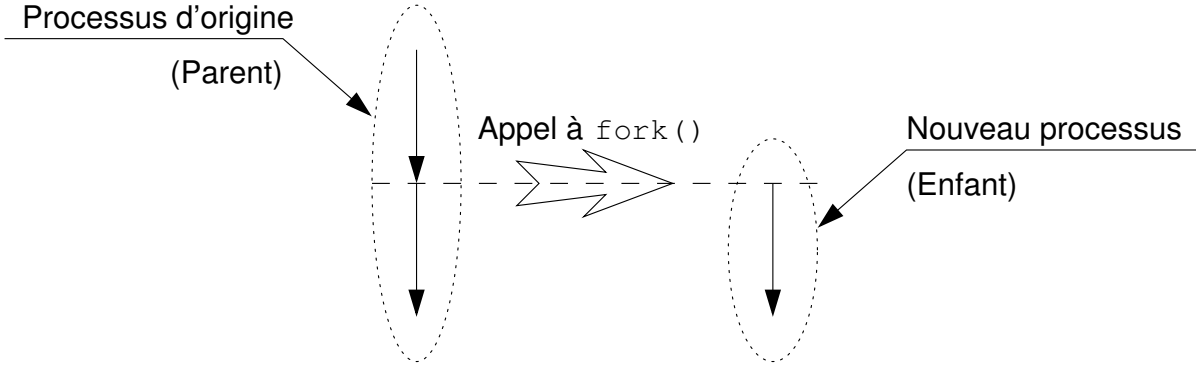
8 : *Attente par le parent*

Les *processus* ...

- **Identification : le** `PID` (*Process IDentifier*)
  - Le type `pid_t` ($\simeq$ entier)
    ```
    #include <sys/types.h>
    ```
  - Récupérer l'identifiant (`man 2 getpid`)
    ```
    #include <unistd.h>
    pid_t getpid(void); // processus courant
    pid_t getppid(void); // processus parent
    ```
  - Toujours un résultat valide

Les *processus* ...

- **Création d'un processus** (`man 2 fork`)
  - `#include <unistd.h>` `pid_t fork(void);`
  - Tous créés ainsi (sauf `init`)
  - Réplique quasi identique du processus d'origine
    - Espaces d'adressage identiques mais **indépendants**
    - Descripteurs de fichiers dupliqués
    - Propriétés identiques (sauf `getpid()` et `getppid()` !)
  - Dédoublement du flot d'exécution (1 passé, 2 avenirs)



Les *processus* ...

- **Création d'un processus**
  - `fork()` renvoie deux valeurs de retour différentes !
    - Dans le parent : identifiant de l'enfant ou `-1` si erreur
    - Dans l'enfant : `0`
    - Permet de distinguer le parent de l'enfant
  - Causes d'échec
    - `ENOMEM` : pas assez de mémoire pour dupliquer l'espace d'adressage
    - `EAGAIN` : plus de place dans la table des processus !

Les *processus* ...

▷ Création d'un processus

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

int main(void)
{
pid_t result;
fprintf(stderr,"Begin %d\n",getpid());
result=fork();
switch(result)
  {
  case -1: perror("fork");                                                    break;
  case  0: fprintf(stderr,"Child : %d (parent=%d)\n",getpid(),getppid()); break;
  default: fprintf(stderr,"Parent : %d (child=%d)\n",getpid(),result);    break;
  }
sleep(1);
fprintf(stderr,"End %d\n",getpid());
return 0;
}
```

```
$ ./prog
Begin 14756
Child : 14757 (parent=14756)
Parent : 14756 (child=14757)
End 14757
End 14756
$
```

Les *processus* ...

▷ Duplication des descripteurs de fichier

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

int main(void)
{
int fd=open("log.txt",O_WRONLY|O_CREAT|O_TRUNC,0666);
if(fd==-1) { perror("open"); return 1; }
write(fd,"COMMON\n",7); /* Should check result ! */
switch(fork())
  {
  case -1: perror("fork"); break;
  case  0: write(fd,"CHILD\n",6); sleep(1); write(fd,"CHILD\n",6); close(fd); break;
  default: sleep(1); write(fd,"PARENT\n",7); close(fd); break;
  }
return 0;
}
```

```
$ ./prog
$ cat log.txt
COMMON
CHILD
PARENT
CHILD
$
```

Les *processus* ...

▷ Duplication de l'espace d'adressage (1/3)

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

void showVar(const char * title,int * var)
{ fprintf(stderr,"%s\t0x%.8X\t[%d]\n",title,(unsigned int)var,*var); }

int var1=100;

int main(void)
{
int var2=200;
int * var3=(int *)malloc(sizeof(int));
*var3=300;
showVar("common/var1",&var1);
showVar("common/var2",&var2);
showVar("common/var3",var3);
```

Les *processus* ...

▷ Duplication de l'espace d'adressage (2/3)

```
switch(fork())
  {
  case -1: perror("fork()"); return EXIT_FAILURE;
  case 0:
    var1+=1; var2+=2; (*var3)+=3; sleep(1);
    showVar("child/var1",&var1);
    showVar("child/var2",&var2);
    showVar("child/var3",var3);
    break;
  default:
    var1+=10; var2+=20; (*var3)+=30; sleep(1);
    showVar("parent/var1",&var1);
    showVar("parent/var2",&var2);
    showVar("parent/var3",var3);
    break;
  }
return EXIT_SUCCESS;
}
```

Les *processus* ...

▷ Duplication de l'espace d'adressage (3/3)

- ◇ Les processus manipulent des adresses *virtuelles*
- ◇ Le système programme la *MMU* pour les traduire en adresses réelles
- ◇ Pas de mémoire partagée implicite
  - ○ Manipuler les mêmes adresses ≠ manipuler les mêmes données !
  - ○ Utiliser des mécanismes dédiés (`man 2 mmap`, `man 2 shmget`)

```
$ ./prog
common/var1     0x08049880      [100]
common/var2     0xBF8903BC      [200]
common/var3     0x0804A008      [300]
child/var1      0x08049880      [101]
child/var2      0xBF8903BC      [202]
child/var3      0x0804A008      [303]
parent/var1     0x08049880      [110]
parent/var2     0xBF8903BC      [220]
parent/var3     0x0804A008      [330]
$
```

Les *processus* ...

- **Terminaison d'un processus**
  - Libérer les ressources du processus
    - Détruire l'espace d'adressage
    - Fermer les descripteurs de fichier
  - Envoyer le signal `SIGCHLD` au parent
  - Le parent doit faire une opération d'attente
    - Permet de savoir comment l'enfant s'est terminé (code de retour ou numéro de signal)
    - Processus terminé mais non attendu → état *zombie*
  - S'il reste des enfants, ils deviennent orphelins !
    - Le système leur attribue alors `init` comme parent d'adoption
    - `init` fait des opérations d'attente pour éliminer les *zombies*

Les *processus* ...

- **Terminaison par retour de la fonction** `main()`
  - Terminaison recommandée !
  - Appel implicite de la fonction `exit()` avec le résultat
    `int main(void) { return 0; }`
- **Terminaison par la fonction** `exit()` (`man 3 exit`)
  - Depuis n'importe quel point du programme
  - `#include <stdlib.h>` `void exit(int status);`
  - Fermeture *"propre"*
    - Fermeture des flux (synchronisation des tampons)
    - Appel des traitements de `atexit()`
  - Puis invocation de l'appel système `_exit()`
    - `#include <unistd.h>` `void _exit(int status);`
    - Terminaison du processus comme décrit précédemment

Les *processus* ...

- **Terminaison par réception d'un signal** (`man 2 kill/sigaction`)
  - Envoi explicite (programme) ou suite à une erreur (système)
  - Si non ignorés et non capturés → terminaison brutale
    - semblable à `_exit()` mais sans code de retour
- **Terminaison par la fonction** `abort()` (`man 3 abort`)
  - `#include <stdlib.h>` `void abort(void);`
  - Fermeture *"propre"* comme dans la fonction `exit()`
  - Puis envoi du signal `SIGABRT` au processus courant

Les *processus* ...

- **Opération d'attente par le processus parent** (`man 2 wait`)
  - ```
    #include <sys/types.h>
    #include <sys/wait.h>
    pid_t waitpid(pid_t pid,int * status,int options);
    pid_t wait(int * status); /* =waitpid(-1,status,0) */
    ```
  - Met à jour l'entier pointé par `status` (si pointeur non nul)
  - `pid` : enfant à attendre, quelconque si `-1`
  - `options` : `0` ou combinaison bit à bit
    - `WNOHANG` : lecture immédiate, sans attente
    - `WUNTRACED` : enfants stoppés également
  - Retour : *PID* de l'enfant ou `0` si `WNOHANG` et pas d'enfant terminé
  - Causes d'erreurs : (retour vaut `-1`)
    - Enfant inconnu, mauvaises options, interruption (`EINTR` → relance)
    ```

Les *processus* ...

- **Opération d'attente par le processus parent**
  - Si l'enfant est déjà terminé avant l'appel à `waitpid()`
    - Le parent n'est pas bloqué, obtention immédiate du résultat
- **Lecture du résultat de l'opération d'attente**
  - Macros pour décoder l'entier pointé par `status`
  - `WIFEXITED(status)` est vrai si terminaison normale de l'enfant
    - `WEXITSTATUS(status)` : valeur renvoyée par l'enfant
  - `WIFSIGNALED(status)` est vrai si l'enfant est terminé par un signal
    - `WTERMSIG(status)` : signal reçu par l'enfant
  - `WIFSTOPPED(status)` est vrai si l'enfant est bloqué
    - `WSTOPSIG(status)` : signal bloquant reçu par l'enfant
  - Option `WUNTRACED` nécessaire pour `WIFSTOPPED` et `WSTOPSIG`

Les *processus* ...

▷ Opération d'attente par le processus parent (1/3)

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

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

int main(void)
{
int status;
pid_t result,pid;
fprintf(stderr,"Begin %d\n",getpid());
result=fork();
switch(result)
  {
  case -1: perror("fork"); return 1;
```

Les *processus* ...

▷ Opération d'attente par le processus parent (2/3)

```
  case 0:
    fprintf(stderr,"Child : %d (parent=%d)\n",getpid(),getppid());
    sleep(1);
    /* *((int *)0)=1234; */ /* Erreur de segmentation ! */
    fprintf(stderr,"Child End %d\n",getpid());
    return 35; /* code de retour */
  }
fprintf(stderr,"Parent : %d (child=%d)\n",getpid(),result);
RESTART_SYSCALL(pid,waitpid(result,&status,0));
fprintf(stderr,"After waitpid() : %d\n",pid);
if(WIFEXITED(status)) fprintf(stderr,"  exit %d\n",WEXITSTATUS(status));
else if(WIFSIGNALED(status)) fprintf(stderr,"  signal %d\n",WTERMSIG(status));
else fprintf(stderr,"  waitpid ?\n");
fprintf(stderr,"Parent End %d\n",getpid());
return 0;
}
```

Les *processus* ...

▷ Opération d'attente par le processus parent (3/3)

Sans l'erreur volontaire

```
$ ./prog
Begin 1751
Child : 1752 (parent=1751)
Parent : 1751 (child=1752)
Child End 1752
After waitpid() : 1752
  exit 35
Parent End 1751
$
```

Avec l'erreur volontaire

```
$ ./prog
Begin 1838
Child : 1839 (parent=1838)
Parent : 1838 (child=1839)
After waitpid() : 1839
  signal 11
Parent End 1838
$
```

Les *processus* ...

▷ Réception de signaux

- ◇ Métaphore des interuptions mais procédé entièrement logiciel
- ◇ Un numéro représenté par une constante (`man 7 signal`)
- ◇ Peuvent être émis explicitement par un programme (`man 2 kill`)
- ◇ Peuvent être émis implicitement par le système
  - ○ `SIGSEGV` : erreur de segmentation (accès à une adresse incorrecte)
  - ○ `SIGINT` : interruption par `Ctrl-C`
  - ○ `SIGCHLD` : terminaison d'un enfant
  - ○ `SIGPIPE` : écriture impossible dans un *tube* ou une *socket*
  - ○ ...
- ◇ La réception provoque généralement la terminaison du processus
- ◇ Possibilité d'ignorer ou d'intercepter des signaux (`man 2 sigaction`)
- ◇ Possibilité de bloquer temporairement des signaux (`man 2 sigprocmask`)

Les *processus* ...

- **Gestionnaire de signaux** (`man 2 sigaction`)
  - `#include <signal.h>`
    `int sigaction(int signum,const struct sigaction * act,`
    `struct sigaction * oldact);`
  - `signum` : numéro du signal concerné (`SIGCHLD`, `SIGPIPE`, ...)
  - `act` : nouveau comportement à adopter
  - `oldact` : pour récupérer l'ancien comportement (si pointeur non nul)
  - `struct sigaction` : description du comportement
    - `void (*sa_handler)(int)` : gestionnaire de signal à invoquer
      - une fonction de notre programme recevant le numéro de signal
      - la constante `SIG_IGN` pour ignorer le signal
      - la constante `SIG_DFL` pour le comportement par défaut
    - d'autres champs non vus ici (on les initialisera à `0`)
  - Retour : `0` si OK, `-1` si erreur (consulter `errno`)

Les *processus* ...

▷ Capturer le signal SIGINT

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

int nb=0;

void myHandler(int signum)
{ fprintf(stderr,"signal %d caught (nb=%d)\n",signum,++nb); }

int main(void)
{
struct sigaction act;
memset(&act,0,sizeof(struct sigaction)); /* tout mettre a 0 (man 3 memset) */
act.sa_handler=&myHandler;
if(sigaction(SIGINT,&act,(struct sigaction *)0)==-1) { perror("sigaction"); return 1; }
while(nb<3) { pause(); } /* attendre un signal (man 2 pause) */
return 0;
}
```

```
$ ./prog
Ctrl-C  -->  signal 2 caught (nb=1)
Ctrl-C  -->  signal 2 caught (nb=2)
Ctrl-C  -->  signal 2 caught (nb=3)
$
```

Les *processus* ...

- **Ignorer le signal** `SIGPIPE`
  - Impossibilité d'écrire dans un *tube* ou une *socket*
    - L'autre extrémité a été fermée
  - Ne mérite pas la terminaison (comportement par défaut)
  - L'erreur `EPIPE` est suffisante
    - Reportée lors de l'opération d'écriture

```
struct sigaction act;
memset(&act,0,sizeof(struct sigaction));
act.sa_handler=SIG_IGN;
if(sigaction(SIGPIPE,&act,(struct sigaction *)0)==-1)
  {
  perror("sigaction");
  return -1;
  }
```

Les *processus* ...

▷ Attendre les enfants automatiquement

◇ Si la terminaison des processus enfants est peu intéressante
→ le processus parent poursuit son propre traitement sans attendre

◇ L'opération d'attente **doit** cependant avoir lieu (éliminer les *zombies*)
→ réagir à `SIGCHLD` par des attentes non-bloquantes

```
void waitChildren(int signum)
{
int r; (void)signum;
do
  {                                /* pid=-1          --> un enfant quelconque  */
  r=waitpid(-1,(int *)0,WNOHANG); /* options=WNOHANG --> attente non-bloquante */
  } while((r!=-1)||(errno==EINTR)); /* pour tous les enfants termines */
}

struct sigaction act;
memset(&act,0,sizeof(struct sigaction));
act.sa_handler=&waitChildren;
if(sigaction(SIGCHLD,&act,(struct sigaction *)0)==-1)
  { perror("sigaction"); return -1; }
```

Les *sockets* locales

- **Communication entre les processus d'une même machine**
  - Famille de protocoles (ou domaine) `PF_LOCAL`
    - Anciennement désignée par `PF_UNIX`
  - Plus de fonctionnalités que les *tubes* de communication (`pipe`)
    - Communication bidirectionnelle
    - Choix du type `SOCK_STREAM` ou `SOCK_DGRAM`
  - Accessibilité comparable à celle des *tubes*
    - Nommées et visibles dans le système de fichiers (création par `socket()` puis `bind()`)
    - Anonymes et partagés par filiation (création par `socketpair()`)
  - Mise en œuvre similaire à la communication par réseau

Les *sockets* locales anonymes

▷ **Création d'une paire anonyme de** *sockets* (`man 2 socketpair`)

◇
```
#include <sys/types.h>
#include <sys/socket.h>
int socketpair(int domain,int type,int proto,
               int fd[2]);
```

◇ `domain=PF_LOCAL`, `type=SOCK_DGRAM` ou `SOCK_STREAM`, `proto=0` (voir la description de `socket()` page 3)

◇ `fd` : reçoit deux descripteurs de fichier bidirectionnels

- Deux extrémités d'une (*pseudo-*)connexion anonyme
- Transmis par filiation, utilisables indifféremment (parent:`fd[0]`/enfant:`fd[1]` ou parent:`fd[1]`/enfant:`fd[0]`)
- Il faut **fermer** l'extrémité **inutilisée** ! (pour pouvoir atteindre `EOF`)

◇ Retour : `0` si OK, `-1` si erreur (consulter `errno`)

Les *sockets* locales anonymes

▷ Différence entre SOCK_STREAM et SOCK_DGRAM (1/2)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

int main(int argc,char ** argv)
{
int fd[2],r; (void)argv;
RESTART_SYSCALL(r,socketpair(PF_LOCAL,argc>1 ? SOCK_STREAM : SOCK_DGRAM,0,fd));
if(r==-1) { perror("socketpair"); return EXIT_FAILURE; }
switch(fork())
  {
  case -1: perror("fork"); return EXIT_FAILURE;
```

Les *sockets* locales anonymes

▷ Différence entre SOCK_STREAM et SOCK_DGRAM (2/2)

```
  case 0:
    RESTART_SYSCALL(r,close(fd[1]));
    RESTART_SYSCALL(r,write(fd[0],"first",5));
    RESTART_SYSCALL(r,write(fd[0],"second",6));
    RESTART_SYSCALL(r,close(fd[0]));
    return EXIT_SUCCESS;
  }
RESTART_SYSCALL(r,close(fd[0])); sleep(1);
for(;;)
  {
  char buffer[0x100];
  RESTART_SYSCALL(r,read(fd[1],buffer,0x100));
  if(r==-1) { perror("read"); break; }
  if(r==0) { fprintf(stderr,"EOF\n"); break; }
  buffer[r]='\0'; fprintf(stderr,"[%s]\n",buffer);
  }
RESTART_SYSCALL(r,wait((int *)0));
RESTART_SYSCALL(r,close(fd[1]));
return EXIT_SUCCESS;
}
```

```
$ ./prog X
[firstsecond]
EOF
$
$ ./prog
[first]
[second]
<-- Ctrl-C (pas de EOF !)
$
```

Les *sockets* locales nommées

▷ **Création d'une** *socket* **locale nommée** (`man 2 socket`)

◇ Création par l'appel système `socket()`
`domain=PF_LOCAL`, `type=SOCK_DGRAM` ou `SOCK_STREAM`, `proto=0`
(voir la description de `socket()` page 3)

◇ Doivent être référencées par une adresse

○ `struct sockaddr` : adresse générique (∀ `domain`)
- champ `sa_family` → `AF_LOCAL`, `AF_INET`, `AF_INET6` ...

○ `struct sockaddr_un` : `#include <sys/un.h>`
- champ `sun_family` → `AF_LOCAL`
- champ `sun_path` → chemin dans le système de fichiers

◇ Opération d'attachement → apparition dans le système de fichiers

○ Pas de lecture/écriture par `open()` (≠ `mkfifo()`)

○ `open()` est tout de même possible (pour `fstat()` par exemple)

○ Suppression par `unlink()`

Les *sockets* locales nommées

▷ **Opération d'attachement à une adresse** (`man 2 bind`)

◇ Si `SOCK_DGRAM` → permet de recevoir des messages

◇ Si `SOCK_STREAM` → permet d'accepter des connexions

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

◇ `fd` : la *socket* locale à attacher

◇ `addr` : pointeur sur l'adresse d'attachement préalablement renseignée

  ○ Ici pointeur sur une `struct sockaddr_un` (nécessite un *cast*)

◇ `addrlen` : taille en octets de l'adresse pointée

  ○ Ici `sizeof(struct sockaddr_un)`

◇ Retour : `0` si OK, `-1` si erreur (consulter `errno`)
(`EADDRINUSE` : une *socket* existe déjà à cette adresse → `unlink()`)

Les *sockets* locales nommées

- **Réception de messages en** `SOCK_DGRAM`
  - Créer la *socket* avec `socket()`
  - L'attacher avec `bind()` à une `struct sockaddr_un`
  - Plusieurs solutions pour la réception
    - `ssize_t read(int fd,void * buf,size_t count);`
      (voir le module *SE1*)
    - `ssize_t recv(int fd,void * buf,size_t count,`
      `int flags);`
      Comme `read()` + options (non vues ici → `0`)
    - `ssize_t recvfrom(int fd,void * buf,size_t count,`
      `int flags,struct sockaddr * from,`
      `socklen_t * fromlen);`
      Comme `recv()` + obtention de l'adresse d'origine du message

Les *sockets* locales nommées

- **Émission de messages en** `SOCK_DGRAM`
  - Créer la *socket* avec `socket()`
  - Avec *pseudo*-connexion (`man 2 connect`, voir plus loin)
    - `ssize_t write(int fd,const void * buf,size_t count);`
      (voir le module *SE1*)
    - ```
      ssize_t send(int fd,const void * buf,size_t count,
                   int flags);
      ```
      Comme `write()` + options (non vues ici → `0`)
  - Sans *pseudo*-connexion
    - ```
      ssize_t sendto(int fd,const void * buf,size_t count,
                     int flags,const struct sockaddr * to,
                               socklen_t tolen);
      ```
      Comme `send()` + adresse de destination du message

Les *sockets* locales nommées

▷ Échange de messages en SOCK_DGRAM (1/4)

```
#include <sys/types.h>    /* commun a l'envoi et la reception */
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

#define SOCKET_NAME "myLocalSocket"
```

Les *sockets* locales nommées

▷ Échange de messages en SOCK_DGRAM (2/4)

```
int main(void) /* programme de reception */
{
int fd,r;
struct sockaddr_un addr;
char buffer[0x100];
RESTART_SYSCALL(fd,socket(PF_LOCAL,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL; strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,bind(fd,(const struct sockaddr *)&addr,
                       sizeof(struct sockaddr_un)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }

RESTART_SYSCALL(r,read(fd,buffer,0x100));
if(r==-1) { perror("read"); return EXIT_FAILURE; }
buffer[r]='\0'; fprintf(stderr,"[%s]\n",buffer);
RESTART_SYSCALL(r,close(fd));
RESTART_SYSCALL(r,unlink(SOCKET_NAME)); /* retirer du systeme de fichiers */
return EXIT_SUCCESS;
}
```

Les *sockets* locales nommées

▷ Échange de messages en SOCK_DGRAM (3/4)

```
int main(void) /* programme d'envoi sans pseudo-connexion */
{
int fd,r;
struct sockaddr_un addr;
RESTART_SYSCALL(fd,socket(PF_LOCAL,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }

memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL;
strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,sendto(fd,"Hello",5,0,(const struct sockaddr *)&addr,
                                        sizeof(struct sockaddr_un)));
if(r==-1) { perror("sendto"); return EXIT_FAILURE; }

RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

Les *sockets* locales nommées

▷ Échange de messages en SOCK_DGRAM (4/4)

```
int main(void) /* programme d'envoi avec pseudo-connexion */
{
int fd,r;
struct sockaddr_un addr;
RESTART_SYSCALL(fd,socket(PF_LOCAL,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }

memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL;
strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,connect(fd,(const struct sockaddr *)&addr,
                          sizeof(struct sockaddr_un)));
if(r==-1) { perror("connect"); return EXIT_FAILURE; }

RESTART_SYSCALL(r,write(fd,"Hello",5));
if(r==-1) { perror("write"); return EXIT_FAILURE; }

RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

Les *sockets* locales nommées

- **Principe d'établissement des connexions en `SOCK_STREAM`**
  - Le serveur crée un *socket* d'écoute (`man 2 socket/bind/listen`)
    - Elle ne sert pas à dialoguer !
  - Un client se connecte au serveur (`man 2 socket/connect`)
    - Cette *socket* servira à dialoguer avec le serveur
  - Le serveur accepte la connexion (`man 2 accept`)
    - Apparition d'un *socket* de dialogue (un nouveau descripteur)
    - Elle permet de dialoguer avec le client
  - La *socket* d'écoute permet d'accepter d'autres connexions
    - Le serveur peut dialoguer simultanément avec plusieurs clients
    - Différentes solutions possibles pour ces dialogues *"simultanés"*

Les *sockets* locales nommées

- **Écoute des connexions en** `SOCK_STREAM` (`man 2 listen`)
  - `#include <sys/socket.h>`
    `int listen(int fd,int backlog);`
  - `fd` : la *socket* d'écoute (de type `SOCK_STREAM`)
    - Doit être préalablement attachée à une `struct sockaddr_un`
  - `backlog` : max. de connexions établies mais pas encore acceptées
    - Limite atteinte → échec des nouvelles tentatives de connexion
  - Retour : `0` si OK, `-1` si erreur (consulter `errno`)

Les *sockets* locales nommées

- **Acceptation des connexions en** `SOCK_STREAM` (`man 2 accept`)
  - ```
    #include <sys/types.h>
    #include <sys/socket.h>
    int accept(int fd,struct sockaddr * addr,
                        socklen_t * addrlen);
    ```
  - `fd` : la *socket* d'écoute (de type `SOCK_STREAM`, `listen()`)
  - `addr` : pointeur pour obtenir l'adresse du client (si pointeur non nul)
    - Ici pointeur sur une `struct sockaddr_un` (nécessite un *cast*)
  - `addrlen` : pointeur sur la taille en octets de l'adresse pointée
    - Paramètre *in/out*, ici `sizeof(struct sockaddr_un)`
  - Retour : Descripteur de la nouvelle *socket* de dialogue
    - *Socket* `SOCK_STREAM` dont l'autre extrémité est le client
    - `-1` si erreur (consulter `errno`)
    ```

Les *sockets* locales nommées

- **Établissement d'une connexion en** `SOCK_STREAM` (`man 2 connect`)
  - ```
    #include <sys/types.h>
    #include <sys/socket.h>
    int connect(int fd,const struct sockaddr * addr,
                        socklen_t addrlen);
    ```
  - `fd` : la *socket* de connexion (de type `SOCK_STREAM`)
  - `addr` : pointeur sur l'adresse du serveur préalablement renseignée
    - Ici pointeur sur une `struct sockaddr_un` (nécessite un *cast*)
  - `addrlen` : taille en octets de l'adresse pointée
    - Ici `sizeof(struct sockaddr_un)`
  - Retour : `0` si OK, `-1` si erreur (consulter `errno`)
    (`ECONNREFUSED` : aucune écoute à cette adresse)
    ```

Les *sockets* locales nommées

- **Échange de données en** `SOCK_STREAM`
  - Après l'établissement d'une connexion, $\forall$ extrémité
  - Pour la réception
    - `read()` ou `recv()` (voir la page 37)
    - `recvfrom()` est inutile $\rightarrow$ nécessairement l'autre extrémité !
  - Pour l'émission
    - `write()` ou `send()` (voir la page 38)
    - `sendto()` provoque `EISCONN` $\rightarrow$ nécessairement l'autre extrémité !
  - L'utilisation de `read()` et `write()` permet d'utiliser indifféremment
    - Une *socket* de type `SOCK_STREAM`
    - Un fichier, un tube de communication, un terminal ...

Les *sockets* locales nommées

- **Fermeture d'une connexion** `SOCK_STREAM`
  - L'opération usuelle `close()` met fin à la connexion
    - Fermeture dans le deux directions
    - Le descripteur de fichier est libéré
  - Fermeture asymétrique de la connexion (`man 2 shutdown`)
    - `#include <sys/socket.h>`
      `int shutdown(int fd,int how);`
    - `fd` : descripteur de la connexion à fermer
    - `how` : fermeture en lecture ou écriture
      - `SHUT_RD` (vaut `0`), `SHUT_WR` (vaut `1`) ou `SHUT_RDWR` (vaut `2`)
    - Le descripteur de fichier n'est pas libéré ! (`close()` reste nécessaire)
    - Retour : `0` si OK, `-1` si erreur (consulter `errno`)

Les *sockets* locales nommées

▷ Échange de données en SOCK_STREAM (1/4)

```
#include <sys/types.h>    /* commun au serveur et au client */
#include <sys/socket.h>
#include <sys/un.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

#define SOCKET_NAME "myLocalSocket"
```

Les *sockets* locales nommées

▷ Échange de données en SOCK_STREAM (2/4)

```
int main(void) /* programme serveur */
{
int listenFd,fd,r;
struct sockaddr_un addr;
RESTART_SYSCALL(listenFd,socket(PF_LOCAL,SOCK_STREAM,0));
if(listenFd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL;
strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,bind(listenFd,(const struct sockaddr *)&addr,
                                  sizeof(struct sockaddr_un)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,listen(listenFd,5));
if(r==-1) { perror("listen"); return EXIT_FAILURE; }
```

Les *sockets* locales nommées

▷ Échange de données en SOCK_STREAM (3/4)

```
for(;;)
  {
  char buffer[0x100];
  socklen_t len=sizeof(struct sockaddr_un);
  RESTART_SYSCALL(fd,accept(listenFd,(struct sockaddr *)&addr,&len));
  if(fd==-1) { perror("accept"); return EXIT_FAILURE; }
  RESTART_SYSCALL(r,read(fd,buffer,0x100));
  if(r==-1) { perror("read"); return EXIT_FAILURE; }
  buffer[r]='\0';
  RESTART_SYSCALL(r,close(fd));
  fprintf(stderr,"[%s]\n",buffer);
  if(!strcmp(buffer,"stop")) { break; }
  }
RESTART_SYSCALL(r,close(listenFd));
RESTART_SYSCALL(r,unlink(SOCKET_NAME));
return EXIT_SUCCESS;
}
```

Les *sockets* locales nommées

▷ Échange de données en SOCK_STREAM (4/4)

```
int main(int argc,char ** argv) /* programme client */
{
int fd,r;
struct sockaddr_un addr;
RESTART_SYSCALL(fd,socket(PF_LOCAL,SOCK_STREAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL;
strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,connect(fd,(const struct sockaddr *)&addr,
                          sizeof(struct sockaddr_un)));
if(r==-1) { perror("connect"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,write(fd,argv[1],strlen(argv[1])));
if(r==-1) { perror("write"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

```
$ ./server   $
[abcd]       $ ./client abcd
[1234]       $ ./client 1234
[stop]       $ ./client stop
$            $
```

Scrutation des flots

- **Surveiller plusieurs flots simultanément**
  - Application communiquante → multiples flots (*sockets* ou autres ...)
    - Opération bloquante sur l'un → impossibilité de réagir aux autres
  - **Mauvaise** solution : boucle d'attente active → à éviter absolument !
    - Mettre les flots en mode non-bloquant
    - Si `EAGAIN` sur l'un on passe à un autre
    - Consomme inutilement du temps de calcul et de l'énergie !
  - Bonne solution : scrutation passive (`man 2 select`)
    - Préciser au système les flots à surveiller et bloquer le processus
    - Réveil lorsqu'un des flots surveillés a évolué ou après un délais
    - Systèmes conçus pour suspendre les processus en attente de ressources
      → Gestion de files d'attentes
      → Puissance de calcul pour les traitements qui en ont besoin

Scrutation des flots

▷ **Mécanisme de scrutation passive** (`man 2 select`)

◇
```
#include <sys/select.h>
int select(int n,fd_set * rdfds,fd_set * wrfds,
           fd_set * exfds,struct timeval * timeout);
```

◇ Ensemble de descripteurs `fd_set` et macros associées
`FD_ZERO(fd_set * set)` : vider l'ensemble
`FD_SET(int fd,fd_set * set)` : ajouter `fd`
`FD_CLR(int fd,fd_set * set)` : retirer `fd`
`FD_ISSET(int fd,fd_set * set)` : tester la présence de `fd`

◇ `n` : descripteur maxi `+ 1` (dimensionnement de masques internes)

◇ `rdfds` : descripteurs à surveiller en lecture

◇ `wrfds` : descripteurs à surveiller en écriture

◇ `exfds` : descripteurs à surveiller pour circonstances exceptionnelles

Scrutation des flots

- **Mécanisme de scrutation passive** (`man 2 select`)
  - `timeout` : délai d'attente maximal avant le retour (∞ si pointeur nul)
    - champ `tv_sec` : nombre de secondes
    - champ `tv_usec` : nombre de μ-secondes dans la dernière seconde
    - Test et retour immédiat si durée nulle
  - Retour :
    - `-1` si erreur (consulter `errno`)
    - Nombre de descripteurs dans les conditions attendues
    - `0` si `timeout` écoulé et rien détecté
    - `rdfds`, `wrfds` et `exfds` sont modifiés (paramètres *in/out*)
      → Ne contiennent plus que les descripteurs qui sont dans l'état attendu
      → La prochaine opération sur l'un d'eux ne sera pas bloquante (succès immédiat ou échec)

Scrutation des flots

- **Mise en œuvre de la scrutation passive**
  - ◇ Initialiser les ensembles de descripteurs avec `FD_ZERO`
  - ◇ Insérer les descripteurs avec `FD_SET` selon l'opération souhaitée
    - ○ Lecture : `read()`, `recv()`, `recvfrom()` ou `accept()`
    - ○ Écriture : `write()` ou `send()` (régulation producteur/consommateur)
    - ○ Circonstances exceptionnelles : peu utilisé
    - ○ Faire un calcul de max. pour déterminer le plus grand descripteur
  - ◇ Préparer éventuellement un `timeout`
  - ◇ Effectuer l'appel à `select`
  - ◇ Pour chaque descripteur précédemment inséré
    - ○ Tester avec `FD_ISSET` s'il est toujours présent dans son ensemble
    - ○ Si oui → faire l'opération souhaitée (correspondant à l'ensemble)
      → Elle ne provoquera aucune attente (seulement **une** opération !)

Scrutation des flots

▷ Utilitaire pour insérer un descripteur en calculant le maximum

```
#define FD_SET_MAX(fd,fdSet,maxFd) \
        { FD_SET((fd),(fdSet)); if((fd)>(maxFd)) { (maxFd)=(fd); } }
```

▷ Utilitaire pour initialiser le délai d'attente

```
#define SET_TIMEVAL(tv,seconds) \
        { (tv).tv_sec=(int)(seconds); \
          (tv).tv_usec=(int)((seconds-(tv).tv_sec)*1e6); }
```

▷ Utilitaire pour déterminer l'instant courant

```
#include <sys/time.h>
double /* current time in seconds */
getTime(void)
{
struct timeval tv;
gettimeofday(&tv,(struct timezone *)0); /* man 2 gettimeofday */
return tv.tv_sec+tv.tv_usec*1e-6;
}
```

Scrutation des flots

▷ Entrée standard + *socket* SOCK_DGRAM + émission régulière (1/3)

```
int main(void) /* precede de #include, #define ... */
{
int fd,r; struct sockaddr_un addr; char buffer[0x100]; double nextTime;
RESTART_SYSCALL(fd,socket(PF_LOCAL,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL; strcpy(addr.sun_path,"SocketOne");
RESTART_SYSCALL(r,bind(fd,(const struct sockaddr *)&addr,
                       sizeof(struct sockaddr_un)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
nextTime=getTime();
for(;;)
  {
  struct timeval tv; fd_set rdSet; int maxFd=-1; double remaining=nextTime-getTime();
  if(remaining<0.0) remaining=0.0;
  SET_TIMEVAL(tv,remaining); FD_ZERO(&rdSet);
  FD_SET_MAX(STDIN_FILENO,&rdSet,maxFd);
  FD_SET_MAX(fd,&rdSet,maxFd);
  RESTART_SYSCALL(r,select(maxFd+1,&rdSet,(fd_set *)0,(fd_set *)0,&tv));
  if(r==-1) { perror("select"); break; }
```

Scrutation des flots

▷ Entrée standard + *socket* SOCK_DGRAM + émission régulière (2/3)

```
if(FD_ISSET(STDIN_FILENO,&rdSet))
  {
  RESTART_SYSCALL(r,read(STDIN_FILENO,buffer,0x100));
  if(r==-1) { perror("read stdin"); return EXIT_FAILURE; }
  buffer[r]='\0';
  fprintf(stderr,"<%s>\n",buffer);
  if(!strcmp(buffer,"stop")) break;
  }
if(FD_ISSET(fd,&rdSet))
  {
  RESTART_SYSCALL(r,read(fd,buffer,0x100));
  if(r==-1) { perror("read socket"); return EXIT_FAILURE; }
  buffer[r]='\0';
  fprintf(stderr,"[%s]\n",buffer);
  if(!strcmp(buffer,"stop")) break;
  }
```

Scrutation des flots

▷ Entrée standard + *socket* SOCK_DGRAM + émission régulière (3/3)

```
 if(getTime()>=nextTime)
   {
   memset(&addr,0,sizeof(struct sockaddr_un));
   addr.sun_family=AF_LOCAL;
   strcpy(addr.sun_path,"myLocalSocket");
   RESTART_SYSCALL(r,sendto(fd,"TicTac",6,0,(const struct sockaddr *)&addr,
                                           sizeof(struct sockaddr_un)));
   if(r==-1) perror("sendto");
   nextTime+=2.0;
   }
 }
RESTART_SYSCALL(r,close(fd));
RESTART_SYSCALL(r,unlink("SocketOne"));
return EXIT_SUCCESS;
}
```

Scrutation des flots

▷ *Sockets* d'écoute et de dialogue SOCK_STREAM (1/2)

```
int main(void) /* precede de #include, #define ... */
{
int listenFd,r,i,stop=0,nbFd=0; int fd[0x20]; char buffer[0x100];
struct sockaddr_un addr;
RESTART_SYSCALL(listenFd,socket(PF_LOCAL,SOCK_STREAM,0));
if(listenFd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_un));
addr.sun_family=AF_LOCAL; strcpy(addr.sun_path,SOCKET_NAME);
RESTART_SYSCALL(r,bind(listenFd,(const struct sockaddr *)&addr,
                                sizeof(struct sockaddr_un)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
if(listen(listenFd,5)==-1) { perror("listen"); return EXIT_FAILURE; }
while(!stop)
  {
  fd_set rdSet; int maxFd=-1;
  FD_ZERO(&rdSet); FD_SET_MAX(listenFd,&rdSet,maxFd);
  for(i=0;i<nbFd;i++) FD_SET_MAX(fd[i],&rdSet,maxFd);
  RESTART_SYSCALL(r,select(maxFd+1,&rdSet,(fd_set *)0,(fd_set *)0,
                           (struct timeval *)0));
  if(r==-1) { perror("select"); break; }
```

Scrutation des flots

▷ *Sockets* d'écoute et de dialogue SOCK_STREAM (2/2)

```
 if(FD_ISSET(listenFd,&rdSet))
   {
   int sock; socklen_t len=sizeof(struct sockaddr_un);
   RESTART_SYSCALL(sock,accept(listenFd,(struct sockaddr *)&addr,&len));
   if(sock==-1) perror("accept"); else fd[nbFd++]=sock;
   }
 for(i=0;i<nbFd;i++)
   if(FD_ISSET(fd[i],&rdSet))
     {
     RESTART_SYSCALL(r,read(fd[i],buffer,0x100));
     if(r<=0) { RESTART_SYSCALL(r,close(fd[i])); fd[i]=fd[--nbFd]; }
     else { buffer[r]='\0'; fprintf(stderr,"[%s]\n",buffer);
            if(!strcmp(buffer,"stop")) stop=1; }
     }
 }
while(nbFd--) RESTART_SYSCALL(r,close(fd[nbFd]));
RESTART_SYSCALL(r,close(listenFd)); RESTART_SYSCALL(r,unlink(SOCKET_NAME));
return EXIT_SUCCESS;
}
```

Scrutation des flots

- **Problème des entrées avec tampon de réception**
  - Par exemple un `FILE *` pour lire un descripteur de fichier (`fdopen()`)
  - `select()` sur le descripteur → attente de données disponibles
  - Lecture avec le `FILE *` (`fread()`, `fgets()`, `fscanf()` ...)
    - En interne, lecture d'un gros bloc sur le descripteur avec `read()` (tentative de remplissage du tampon de reception)
    - Plus de données dans le tampon que celles immédiatement attendues (ex : plusieurs lignes reçues d'un coup mais une seule pour `fgets()`)
  - Doit-on refaire une scrutation passive pour obtenir la suite ?!?
    - Oui si le tampon de réception est vide
    - Non si le tampon de réception est n'est pas vide (les données utiles sont déjà dans le tampon)
      → `select()` bloqué en attente de nouvelles données
      → Elles n'arriveront pas ! (déjà reçues)

Scrutation des flots

▷ **Problème des entrées avec tampon de réception**

◇ Il est donc nécessaire de consulter l'état du tampon de réception

◇ Pour *SSL* il y a `SSL_pending()` (voir le cours *SSL*)

◇ Pour les `FILE *`, il n'y a pas de solution standard `:-(`

○ Dans le cas de la *GNU lib C* on peut faire

```
#define INPUT_STREAM_PENDING(stream) \
    ((stream)->_IO_read_end-(stream)->_IO_read_ptr)
```

◇ Si le tampon de réception est vide → démarche usuelle (`FD_SET()` → `select()` → si `FD_ISSET()`, lecture)

◇ Si le tampon de réception n'est pas vide

○ Ne pas insérer le descripteur correspondant dans le `fd_set`

○ Effectuer un `select()` avec un `timeout` de durée nulle (les données sont déjà dans le tampon → pas d'attente !)

○ Procéder à la lecture sans tester `FD_ISSET()`

Scrutation des flots

▷ Problème des entrées avec tampon de réception (1/3)

```
/* #include, #define ... */
int main(void) /* sans consultation du tampon de reception */
{
for(;;) /* nb: il faudrait plusieurs descripteurs pour que select() ait un interet */
  {
  int r,maxFd=-1;
  fd_set rdSet;
  FD_ZERO(&rdSet);
  FD_SET_MAX(fileno(stdin),&rdSet,maxFd);
  RESTART_SYSCALL(r,select(maxFd+1,&rdSet,(fd_set *)0,(fd_set *)0,
                           (struct timeval *)0));
  if(r==-1) { perror("select"); break; }
  if(FD_ISSET(fileno(stdin),&rdSet))
    {
    int c=fgetc(stdin);
    if(c==EOF) break; else fprintf(stderr,"--> %d [%c]\n",c,(char)c);
    }
  }
return 0;
}
```

Scrutation des flots

▷ Problème des entrées avec tampon de réception (2/3)

```
/* #include, #define ... */
#define INPUT_STREAM_PENDING(stream) ((stream)->_IO_read_end-(stream)->_IO_read_ptr)
int main(void) /* avec consultation du tampon de reception */
{
for(;;) /* nb: il faudrait plusieurs descripteurs pour que select() ait un interet */
  {
  int r,maxFd=-1; fd_set rdSet;
  struct timeval tv0={0,0}; struct timeval * ptv=(struct timeval *)0;
  FD_ZERO(&rdSet);
  if(INPUT_STREAM_PENDING(stdin)) ptv=&tv0;
  else FD_SET_MAX(fileno(stdin),&rdSet,maxFd);
  RESTART_SYSCALL(r,select(maxFd+1,&rdSet,(fd_set *)0,(fd_set *)0,ptv));
  if(r==-1) { perror("select"); break; }
  if(INPUT_STREAM_PENDING(stdin)||FD_ISSET(fileno(stdin),&rdSet))
    { int c=fgetc(stdin);
      if(c==EOF) break; else fprintf(stderr,"--> %d [%c]\n",c,(char)c); }
  }
return 0;
}
```

Scrutation des flots

▷ Problème des entrées avec tampon de réception (3/3)

```
$ ./prog1
abcd          ("abcd\n" arrivent ensemble)
--> 97 [a]    ("bcd\n" restent dans le tampon)
1234          (DEBLOCAGE PAR NOUVELLE SAISIE)
--> 98 [b]    ( fgetc() consomme le tampon,
--> 99 [c]      pas de nouvelle lecture
--> 100 [d]     depuis le descripteur
--> 10 [        ...)
]             (le tampon est vide --> lecture)
--> 49 [1]    ("234\n" restent dans le tampon)
   <-- Ctrl-C
```

```
$ ./prog2
abcd          ("abcd\n" arrivent ensemble)
--> 97 [a]    (tampon non vide donc pas
--> 98 [b]     d'attente dans select(),
--> 99 [c]     fgetc() consomme le tampon
--> 100 [d]    ...
--> 10 [       ...)
]             (le tampon est vide)
1234          (DEBLOCAGE PAR NOUVELLE SAISIE)
--> 49 [1]
--> 50 [2]
--> 51 [3]
--> 52 [4]
--> 10 [
]
   <-- Ctrl-C
```

Scrutation des flots

▷ Scrutation en écriture

- ◇ La scrutation en lecture est une démarche naturelle
  - ○ On ne maîtrise pas les données qui nous parviennent
  - ○ On guette alors leur arrivée pour les traiter
- ◇ En écriture c'est plus surprenant
  - ○ On connaît exactement les données à transmettre
  - ○ Toutes les données doivent *"passer"* avant de poursuivre (généralement il n'y a rien à attendre donc pas de blocage)
- ◇ Le système utilise en interne des tampons de taille limitée
  - ○ Si le consommateur n'extrait pas les données → remplissage → blocage lors de l'écriture !!!
  - ○ Il faut attendre qu'il y ait à nouveau de la place pour écrire la suite
- ◇ `select()` permet d'être tenu au courant de cette possibilité

Scrutation des flots

▷ Scrutation en écriture

- ◇ La plupart du temps aucune opérations de lecture n'est possible
  - ○ De temps en temps des données arrivent
    → la lecture devient possible
- ◇ La plupart du temps les opérations d'écritures sont possibles
  - ○ De temps en temps le tampon interne du système est plein
    → l'écriture devient impossible
- ◇ Démarche à adopter pour des écritures sans risque de blocage :
  - ○ L'application place les données à émettre dans une file
  - ○ Si la file n'est pas vide → scruter en écriture
    (vidage d'une portion de la file dès que ça devient possible)
  - ○ Si la file devient vide → ne plus scruter en écriture
- ◇ Rappel : `write()` ne bloque que si aucun octet ne peut être écrit
  (sinon écriture immédiate de tout ce qui peut passer, 1 octet ou plus)

Scrutation des flots

▷ Scrutation en écriture

```
char buffer[0x100]; /* tampon qu'on remplit et vide regulierement */
int remaining=0;    /* quantite restant a transmettre */
char * ptr=buffer;  /* prochaines donnees a transmettre */
for(;;)
  {
  fd_set rdSet,wrSet; int maxFd=-1;
  FD_ZERO(&rdSet); FD_ZERO(&wrSet);
  if(remaining) FD_SET_MAX(fd,&wrSet,maxFd);
  /* ... inserer d'autres descripteurs ... */
  RESTART_SYSCALL(r,select(maxFd+1,&rdSet,&wrSet,(fd_set *)0,(struct timeval *)0));
  if(r==-1) { perror("select"); break; }
  if(FD_ISSET(fd,&wrSet))
    {
    RESTART_SYSCALL(r,write(fd,ptr,remaining));
    if(r<=0) ptr=(char *)0;       /* plus d'ecriture possible ... */
    else if(remaining-=r) ptr+=r; /* il en reste encore */
    else ptr=buffer;              /* tout est passe */
    }
  /* ... tester les autres descripteurs ... */
  }
```

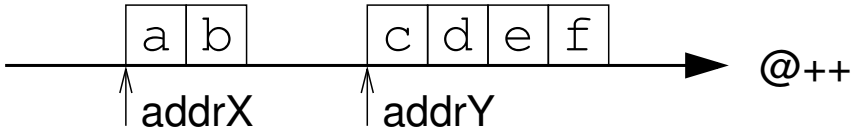
Les *sockets Internet*

- **Communication entre les processus de machines distantes**
  - Famille de protocoles (ou domaine) `PF_INET` (repose sur *IPv4*)
    - Il en existe d'autres (`PF_INET6`, `PF_APPLETALK` ...)
    - Abstraction bien plus élevée que l'accès aux interfaces réseau
  - Utilisation très similaire aux *sockets* locales
    - `SOCK_DGRAM` repose sur *UDP*
    - `SOCK_STREAM` repose sur *TCP*
    - Opérations d'attachement, d'écoute, de connection ...
    - Pas de `socketpair()` (n'a de sens qu'en local)
  - Utilisation d'un nouveau type d'adresse : `struct sockaddr_in`
    - Famille `AF_INET`
    - Une adresse *IP*
    - Un numéro de port

Les *sockets Internet*

▷ **Données au format** *hôte* **et au format** *réseau*

◇ La représentation des entiers varie d'une machine à l'autre



◇ Sur une machine *big-endian*

- ○ La valeur de 16 bits en `addrX` vaut `(a<<8)+b`
- ○ La valeur de 32 bits en `addrY` vaut `(c<<24)+(d<<16)+(e<<8)+f`

◇ Sur une machine *little-endian*

- ○ La valeur de 16 bits en `addrX` vaut `(b<<8)+a`
- ○ La valeur de 32 bits en `addrY` vaut `(f<<24)+(e<<16)+(d<<8)+c`

◇ Pour communiquer il faut s'accorder sur l'ordre des octets

- ○ Dans les entêtes des protocoles *IP*, *UDP*, *TCP* ...
- ○ Dans les adresses des *sockets* (adresses *IP* + port)
- ○ Dans les données applicatives

Les *sockets Internet*

- **Données au format** *hôte* **et au format** *réseau*
  - Ordre réseau standard : *big-endian*
    - Les machines *little-endian* doivent donc s'adapter
    - Inverser l'ordre des octets des valeurs de plus de 8 bits
  - Utilisation de macros pour rendre le code portable
    - `#include <arpa/inet.h>`
    - `host16value=ntohs(net16value);` *network-to-host-short*
    - `net16value=htons(host16value);` *host-to-network-short*
    - `host32value=ntohl(net32value);` *network-to-host-long*
    - `net32value=htonl(host32value);` *host-to-network-long*
    - N'inversent les octets que si nécessaire sur la machine

Les adresses des *sockets Internet*

- **Effectuer une résolution** *DNS* (`man 3 gethostbyname`)
  - `#include <netdb.h>`
    `struct hostent * gethostbyname(const char * name);`
  - Recherche l'adresse de la machine ayant le nom de domaine `name`
  - Renvoie un pointeur nul si la résolution est impossible
    - Consulter la variable spécifique `h_errno`
  - En cas de succès, consulter le champ `h_addr` du résultat
    - *Cast* en `unsigned long *` → adresse *IP* en ordre *réseau*
    - Ne pas libérer/modifier le résultat (on le consulte juste)
    - Fonction non-réentrante → synchro. nécessaire en *multi-threads* !

Les adresses des *sockets Internet*

- **Déterminer le nom de la machine courante** (`man 2 gethostname`)
  - `#include <unistd.h>`
    `int gethostname(char * name,size_t len);`
  - Le nom de la machine (au sens *DNS*) est placé dans `name`
    - Doit être préalablement alloué à `len` octets
  - Retour : `0` si OK, `-1` si erreur (consulter `errno`)

```
#include <stdio.h>
#include <unistd.h>
#include <netdb.h>
int main(void)
{
char name[0x100]; unsigned long addr; struct hostent * host;
if(gethostname(name,0x100)==-1) { perror("gethostname"); return 1; }
host=gethostbyname(name);
addr=host ? ntohl(*((unsigned long *)(host->h_addr))) : 0;
printf("I am ‘%s’ (%ld.%ld.%ld.%ld)\n",name,(addr>>24)&0x00FF,(addr>>16)&0x00FF,
                                          (addr>>8)&0x00FF,addr&0x00FF);
return 0;
}
```

```
$ ./prog
I am ‘nowin’ (172.16.1.8)
$
```

Les adresses des *sockets Internet*

- **Attachement explicite** (`man 2 bind`)
  - Avant une écoute en *TCP* ou une attente de message en *UDP*
  - Utilisation d'un nouveau type d'adresse : `struct sockaddr_in`
    - `#include <netinet/in.h.h>`
    - Champ `sin_family` : `AF_INET`
    - Champ `sin_port` : un numéro de port (16 bits, ordre *réseau*)
    - Champ `sin_addr.s_addr` : une adresse *IP* (32 bits, ordre *réseau*)
  - Voir la description de `bind()` page 36
    - `addr` : pointeur sur une `struct sockaddr_in` (nécessite un *cast*)
    - `addrlen` : `sizeof(struct sockaddr_in)`
  - Si le port vaut `0` → choix arbitraire d'un port libre
  - L'adresse *IP* doit être celle d'une interface de la machine
    - Si `INADDR_ANY`, l'interface n'importe pas

Les adresses des *sockets Internet*

- **Attachement implicite** (`man 2 bind`)
  - Attachement explicite pas toujours nécessaire (programmes *clients*)
    - Lors de l'établissement d'une connexion *TCP* (`man 2 connect`)
    - Lors de l'envoi d'un message *UDP* (`man 2 sendto`)
    - Besoin d'une adresse *IP source* et d'un port *source*
  - Équivalent à un attachement à `INADDR_ANY` sur le port `0`
    - Choix arbitraire d'un port libre
    - Adresse *IP* source : celle de l'interface de sortie
- **Adresses utilisées dans les primitives de communication**
  - Dans `connect()`, `accept()`, `sendto()`, `recvfrom()` (exactement la même démarche qu'avec les *sockets* locales)
  - L'adresse d'une `struct sockaddr_in` *castée* en `struct sockaddr *`
  - La taille est `sizeof(struct sockaddr_in)`

Communication en *UDP*

- **Les** *sockets UDP* (`man 2 socket`)
  - Création par l'appel système `socket()` (voir la description page 3)
    `domain=PF_INET`, `type=SOCK_DGRAM`, `proto=0`
  - Usage très similaire aux *sockets* locales de type `SOCK_DGRAM`
  - Possibilité de recevoir des messages diffusés
    - Autoriser avec l'option `SO_BROADCAST` (`man 2 setsockopt`)
      (pour l'envoi **et** pour la reception)
    - Un adresse *IP* de diffusion s'utilise comme l'adresse *IP* d'une machine
      (à la réception, un ensemble de machine est susceptible d'être concerné)
  - Plusieurs *sockets UDP* peuvent être attachées à la même adresse !
    - Autoriser avec l'option `SO_REUSEADDR` (`man 2 setsockopt`)
    - Potentiellement ambigu mais très utile en cas de diffusion
  - Communication *"non-fiable"* (délivrance ? ordre ? ...)

Communication en *UDP*

▷ Échange de messages en *UDP* (1/4)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

int main(void) /* programme de reception */
{
int fd,r,on; struct sockaddr_in addr; socklen_t len;
unsigned long ipAddr; char buffer[0x100];
RESTART_SYSCALL(fd,socket(PF_INET,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
```

Communication en *UDP*

▷ Échange de messages en *UDP* (2/4)

```
on=1; /* autoriser l'option SO_BROADCAST */
RESTART_SYSCALL(r,setsockopt(fd,SOL_SOCKET,SO_BROADCAST,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_BROADCAST"); return EXIT_FAILURE; }
on=1; /* autoriser l'option SO_REUSEADDR */
RESTART_SYSCALL(r,setsockopt(fd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_REUSEADDR"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876); addr.sin_addr.s_addr=htonl(INADDR_ANY);
RESTART_SYSCALL(r,bind(fd,(const struct sockaddr *)&addr,sizeof(struct sockaddr_in)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
len=sizeof(struct sockaddr_in);
RESTART_SYSCALL(r,recvfrom(fd,buffer,0x100,0,(struct sockaddr *)&addr,&len));
if(r==-1) { perror("recvfrom"); return EXIT_FAILURE; }
buffer[r]='\0'; ipAddr=ntohl(addr.sin_addr.s_addr);
fprintf(stderr,"[%s] from %ld.%ld.%ld.%ld:%d\n",buffer,(ipAddr>>24)&0x00FF,
        (ipAddr>>16)&0x00FF,(ipAddr>>8)&0x00FF,ipAddr&0x00FF,ntohs(addr.sin_port));
RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

Communication en *UDP*

▷ Échange de messages en *UDP* (3/4)

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/in.h>
#include <netdb.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

int main(int argc,char ** argv) /* programme d'envoi */
{
int fd,r,on; struct hostent * host; struct sockaddr_in addr; unsigned long ipAddr;
host=gethostbyname(argc>1 ? argv[1] : "localhost");
if(!host) { fprintf(stderr,"unknown host\n"); return EXIT_FAILURE; }
ipAddr=ntohl(*((unsigned long *)(host->h_addr)));
```

Communication en *UDP*

▷ Échange de messages en *UDP* (4/4)

```
RESTART_SYSCALL(fd,socket(PF_INET,SOCK_DGRAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
on=1; /* autoriser l'option SO_BROADCAST */
RESTART_SYSCALL(r,setsockopt(fd,SOL_SOCKET,SO_BROADCAST,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_BROADCAST"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876);
addr.sin_addr.s_addr=htonl(ipAddr);
RESTART_SYSCALL(r,sendto(fd,"Hello",5,0,(const struct sockaddr *)&addr,
                                        sizeof(struct sockaddr_in)));
if(r==-1) { perror("sendto"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

```
$ ./sendUdp nowin

$ ./sendUdp 192.168.40.255

$ ./sendUdp 255.255.255.255
```

```
$ ./recvUdp
[Hello] from 192.168.40.29:32784
$ ./recvUdp
[Hello] from 192.168.40.29:32784
$ ./recvUdp
[Hello] from 192.168.40.29:32784
```

Communication en *TCP*

- **Les** *sockets TCP* (`man 2 socket`)
  - Création par l'appel système `socket()` (voir la description page 3)
    `domain=PF_INET`, `type=SOCK_STREAM`, `proto=0`
  - Usage très similaire aux *sockets* locales de type `SOCK_STREAM`
  - Éviter le problème de l'état `TIME_WAIT` des connexions *TCP*
    - L'adresse (*IP*+port) est indisponible pour un certain temps (attendre avant qu'un nouvel attachement devienne possible)
    - Autoriser malgré tout avec l'option `SO_REUSEADDR`
    - Permet juste d'éviter d'attendre avant de relancer un serveur (Ne permet pas d'avoir plusieurs serveurs sur le même port !)
  - Communication *"fiable"* (délivrance, ordre ...)
    - Toutefois, les limites des échanges sont soumises aux aléas du transport (ex : N octets en E envois $\rightarrow$ N octets en R réceptions, E $\neq$ R)
      $\rightarrow$ Il faut utiliser attentivement le résultat de `read()`/`write()`

Communication en *TCP*

▷ Échange de données en *TCP* (1/4)

```
#include <sys/types.h>    /* commun au serveur et au client */
#include <sys/socket.h>
#include <netinet/in.h>
#include <netdb.h>        $ ./server                              $
#include <unistd.h>       [abcd] from 192.168.40.29:34823         $ ./client abcd nowin
#include <stdio.h>        [1234] from 192.168.40.29:34824         $ ./client 1234 nowin
#include <stdlib.h>       [stop] from 192.168.40.29:34825         $ ./client stop nowin
#include <string.h>       $                                       $
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));
```

Communication en *TCP*

▷ Échange de données en *TCP* (2/4)

```
int main(void) /* programme serveur */
{
int listenFd,fd,r,on;
struct sockaddr_in addr;
RESTART_SYSCALL(listenFd,socket(PF_INET,SOCK_STREAM,0));
if(listenFd==-1) { perror("socket"); return EXIT_FAILURE; }
on=1; /* autoriser l'option SO_REUSEADDR */
RESTART_SYSCALL(r,setsockopt(listenFd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_REUSEADDR"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876);
addr.sin_addr.s_addr=htonl(INADDR_ANY);
RESTART_SYSCALL(r,bind(listenFd,(const struct sockaddr *)&addr,
                                sizeof(struct sockaddr_in)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
if(listen(listenFd,5)==-1) { perror("listen"); return EXIT_FAILURE; }
```

Communication en *TCP*

▷ Échange de données en *TCP* (3/4)

```
for(;;)
  {
  char buffer[0x100];
  unsigned long ipAddr;
  socklen_t len=sizeof(struct sockaddr_in);
  RESTART_SYSCALL(fd,accept(listenFd,(struct sockaddr *)&addr,&len));
  if(fd==-1) { perror("accept"); return EXIT_FAILURE; }
  RESTART_SYSCALL(r,read(fd,buffer,0x100));
  if(r==-1) { perror("read"); return EXIT_FAILURE; }
  buffer[r]='\0';
  ipAddr=ntohl(addr.sin_addr.s_addr);
  fprintf(stderr,"[%s] from %ld.%ld.%ld.%ld:%d\n",buffer,(ipAddr>>24)&0x00FF,
          (ipAddr>>16)&0x00FF,(ipAddr>>8)&0x00FF,ipAddr&0x00FF,ntohs(addr.sin_port));
  RESTART_SYSCALL(r,close(fd));
  if(!strcmp(buffer,"stop")) { break; }
  }
RESTART_SYSCALL(r,close(listenFd));
return EXIT_SUCCESS;
}
```

Communication en *TCP*

▷ Échange de données en *TCP* (4/4)

```
int main(int argc,char ** argv) /* programme client */
{
int fd,r; struct hostent * host; struct sockaddr_in addr; unsigned long ipAddr;
host=gethostbyname(argc>2 ? argv[2] : "localhost");
if(!host) { fprintf(stderr,"unknown host\n"); return EXIT_FAILURE; }
ipAddr=ntohl(*((unsigned long *)(host->h_addr)));
RESTART_SYSCALL(fd,socket(PF_INET,SOCK_STREAM,0));
if(fd==-1) { perror("socket"); return EXIT_FAILURE; }
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876);
addr.sin_addr.s_addr=htonl(ipAddr);
RESTART_SYSCALL(r,connect(fd,(const struct sockaddr *)&addr,
                          sizeof(struct sockaddr_in)));
if(r==-1) { perror("connect"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,write(fd,argv[1],strlen(argv[1])));
if(r==-1) { perror("write"); return EXIT_FAILURE; }
RESTART_SYSCALL(r,close(fd));
return EXIT_SUCCESS;
}
```

Communication en *TCP*

- **Identifier une connexion** *TCP* (`man 2 getsockname/getpeername`)
  - Adresse *IP* et port de chaque extrémité d'une connexion déjà établie
    - Obtenue par héritage, loin dans le code ...
  - `#include <sys/socket.h>`
    `int getsockname(int fd,` (extrémité locale)
    `struct sockaddr * addr,socklen_t * len);`
    `int getpeername(int fd,` (extrémité distante)
    `struct sockaddr * addr,socklen_t * len);`
  - `addr` et `len` : désignent une `struct sockaddr_in` ici
  - Retour : `0` si OK, `-1` si erreur (consulter `errno`)
  - Ces informations sont souvent déjà connues
    - Port passé à `bind()` (sauf si `0`) = port obtenu par `getsockname()`
    - Adresse obtenue par `accept()` = adresse obtenue par `getpeername()`
    - Adresse passée à `connect()` = adresse obtenue par `getpeername()`

Communication en *TCP*

▷ Identifier une connexion *TCP*

```
int fd= ... /* une connexion deja etablie */
int r; unsigned long ipAddr;                          this end 192.168.40.29:49809
struct sockaddr_in addr;                              other end 192.168.40.236:9876
socklen_t len;

len=sizeof(struct sockaddr_in);
RESTART_SYSCALL(r,getsockname(fd,(struct sockaddr *)&addr,&len));
if(r==-1) { perror("getsockname"); return -1; }
ipAddr=ntohl(addr.sin_addr.s_addr);
fprintf(stderr,"this end %ld.%ld.%ld.%ld:%d\n",(ipAddr>>24)&0x00FF,
        (ipAddr>>16)&0x00FF,(ipAddr>>8)&0x00FF,ipAddr&0x00FF,ntohs(addr.sin_port));

len=sizeof(struct sockaddr_in);
RESTART_SYSCALL(r,getpeername(fd,(struct sockaddr *)&addr,&len));
if(r==-1) { perror("getpeername"); return -1; }
ipAddr=ntohl(addr.sin_addr.s_addr);
fprintf(stderr,"other end %ld.%ld.%ld.%ld:%d\n",(ipAddr>>24)&0x00FF,
        (ipAddr>>16)&0x00FF,(ipAddr>>8)&0x00FF,ipAddr&0x00FF,ntohs(addr.sin_port));
```

Bilan intermédiaire sur les sockets

- ▷ *Sockets* **locales**
  - ◇ Communication entre les processus d'une même machine
  - ◇ Plus de nuances et de fonctionnalités que les tubes (`pipe`)
  - ◇ Opérations compatibles avec les autres types de descripteurs de fichier
- ▷ *Sockets Internet*
  - ◇ Communication à travers les réseaux en *UDP* et *TCP*
  - ◇ Utilisation et démarche similaires aux *socket* locales
    - ○ Écoute, connexion, envoi, réception, scrutation ...
  - ◇ Quelques opérations supplémentaires à considérer
    - ○ Gestion des noms de domaines, des adresses *IP* et des ports
    - ○ Précautions à prendre quant à l'ordre des octets (*hôte*/*réseau*)
      → Imposé pour ce qui est des adresses *IP* et des ports
      → À gérer attentivement pour les données applicatives

L'architecture client/serveur *TCP*

- **Expression du besoin**
  - Un serveur écoute sur une port donné
  - Un nombre quelconque de clients doivent pouvoir s'y connecter
  - Le serveur doit pouvoir dialoguer simultanément avec ses clients
    - Pour rendre des services plus ou moins longs
    - Sans interférence d'un client sur un autre
- **Situation actuelle : scrutation passive**
  - Répond **efficacement** au besoin `:-)`
    - Peu gourmand en ressources et très réactif
  - Difficile à écrire en cas de longs dialogues avec les clients `:-(`
    - Blocage d'un dialogue → blocage de tous les dialogues et de l'écoute
    - Repasser souvent dans la boucle de scrutation et avancer par étape
    - Décrire l'algorithme comme une machine à états !

L'architecture client/serveur *TCP*

- **Traitement parallèle des requêtes**
  - À chaque nouvelle connexion une nouvelle tâche en parallèle
    - Une seule tâche d'écoute et de multiples tâches de dialogue
    - Solution *multi-processus* (historique) ou *multi-threads* (moderne)
  - Beaucoup plus gourmand en ressources `:-(`
    - Ordonnancement, espaces d'adressages, piles d'exécution ...
  - Algorithme beaucoup plus facile à écrire `:-)`
    - En cas de traitements longs, bloquants ...
    - Pas de risque de bloquer les autres dialogues ou l'écoute
  - Intéressant uniquement si le service n'est pas rendu immédiatement
    - Créer une tâche pour la détruire immédiatement !?!
    - Dans ce cas on continue à utiliser la scrutation passive

Serveur *TCP multi-processus*

- **Principe : 1 connexion → 1 processus**
  - Le programme principal effectue une boucle d'écoute (`accept()`)
    - Un processus enfant est créé (`fork()`) à chaque nouvelle connexion
    - Le processus enfant hérite des descripteurs de fichier du parent
  - Le processus parent ferme immédiatement la *socket* de dialogue
    - Elle est utilisée par le processus enfant
    - Le parent peut attendre immédiatement une nouvelle connexion
  - Le processus enfant ferme immédiatement la *socket* d'écoute
    - Elle est utilisée par le processus parent
    - L'enfant exploite librement la *socket* de dialogue
  - C'est la solution *"historique"*
    - Cloisonnement des dialogues (un plantage reste localisé)
    - Duplique l'espace d'adressage ! (sauf si recouvrement, voir plus loin)

Serveur *TCP multi-processus*

▷ Exemple de mise en œuvre

```
#include <sys/types.h>          $ ./server              $
#include <sys/wait.h>           [abcd] (pid=13371)      $ ./client abcd nowin
#include <sys/socket.h>         [1234] (pid=13378)      $ ./client 1234 nowin
#include <netinet/in.h>         [hello] (pid=13388)     $ ./client hello nowin
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

void waitChildren(int signum) /* eliminer automatiquement les enfants termines */
{
int r;
(void)signum;
do { r=waitpid(-1,(int *)0,WNOHANG); } while((r!=-1)||(errno==EINTR));
}
```

Serveur *TCP multi-processus*

```
void service(int fd)
{
char buffer[0x100]; int r;
RESTART_SYSCALL(r,read(fd,buffer,0x100));
if(r==-1) { perror("read"); return; }
buffer[r]='\0'; fprintf(stderr,"[%s] (pid=%d)\n",buffer,getpid());
RESTART_SYSCALL(r,close(fd));
}

int main(void)
{
struct sigaction act;
int listenFd,fd,r,on;
struct sockaddr_in addr;
memset(&act,0,sizeof(struct sigaction));
act.sa_handler=&waitChildren;
if(sigaction(SIGCHLD,&act,(struct sigaction *)0)==-1)
  { perror("sigaction"); return -1; }
RESTART_SYSCALL(listenFd,socket(PF_INET,SOCK_STREAM,0));
if(listenFd==-1) { perror("socket"); return EXIT_FAILURE; }
on=1; /* autoriser l'option SO_REUSEADDR */
RESTART_SYSCALL(r,setsockopt(listenFd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_REUSEADDR"); return EXIT_FAILURE; }
```

Serveur *TCP multi-processus*

```
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876);
addr.sin_addr.s_addr=htonl(INADDR_ANY);
RESTART_SYSCALL(r,bind(listenFd,(const struct sockaddr *)&addr,
                                sizeof(struct sockaddr_in)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
if(listen(listenFd,5)==-1) { perror("listen"); return EXIT_FAILURE; }
for(;;)
  {
  RESTART_SYSCALL(fd,accept(listenFd,(struct sockaddr *)0,(socklen_t *)0));
  if(fd==-1) { perror("accept"); return EXIT_FAILURE; }
  switch(fork())
    {
    case -1: perror("fork"); break;
    case 0: RESTART_SYSCALL(r,close(listenFd)); service(fd); exit(0);
    default: RESTART_SYSCALL(r,close(fd));
    }
  }
RESTART_SYSCALL(r,close(listenFd));
return EXIT_SUCCESS;
}
```

Serveur *TCP multi-processus*

▷ **Recouvrement de processus** (`man 2 execve`)

◇ C'est le moyen d'exécuter un nouveau programme

◇ L'espace d'adressage du processus est complètement réinitialisé

- Remplacer le code et les données par un nouveau programme
- Les descripteurs de fichiers sont conservés (toujours utilisables)

◇
```
#include <unistd.h>
int execve(const char * file,char * const argv[],
                             char * const envp[]);
```

◇ `file` : chemin vers le programme (fichier exécutable)

◇ `argv` : ligne de commande (normalement `argv[0]=file`)

◇ `envp` : variables d'environnement (de la forme `"nom=valeur"`)

◇ Le dernier élément de `argv` et `envp` est `(char *)0`

◇ **Pas de retour** si OK, `-1` si erreur (consulter `errno`)

Serveur *TCP multi-processus*

- **Recouvrement de processus**
  - Il existe des variantes (facilités) qui appellent toutes `execve()` (`man 3 execl/execlp/execle/execv/execvp`)
    - `p` : recherche de `file` dans le `PATH` (si pas de `/`)
    - `v` : ligne de commande passée dans un tableau (`[]`)
    - `l` : ligne de commande énumérée (`...`) terminée par `(char *)0`
    - `e` : environnement fourni, sinon le courant est réutilisé
  - ex : `int execlp(const char * file,const char * arg,...);`
- **Recouvrement de processus et redirection d'entrée/sortie**
  - `dup2(fd,STDIN_FILENO); dup2(fd,STDOUT_FILENO); close(fd);` (voir le module *SE1*)
  - `execlp("prog","prog","arg1","arg2",(char *)0);`
  - Les entrée/sortie de `prog` utilisent ce qui était désigné par `fd`

Serveur *TCP multi-processus*

▷ Serveur *TCP* avec recouvrement de processus

```
void service(int fd) /* a utiliser dans le programme precedent */
{
int r;
RESTART_SYSCALL(r,dup2(fd,STDIN_FILENO)); if(r==-1) { perror("dup2 in"); return; }
RESTART_SYSCALL(r,dup2(fd,STDOUT_FILENO)); if(r==-1) { perror("dup2 out"); return; }
RESTART_SYSCALL(r,dup2(fd,STDERR_FILENO)); if(r==-1) { perror("dup2 err"); return; }
RESTART_SYSCALL(r,close(fd));
execlp("sh","sh","-i",(const char *)0); /* !! offrir un shell est tres dangereux !! */
perror("execlp");
}
```

```
{user@nowin}$ ./server
```

```
{keke@nowhere}$ nc nowin 9876
sh: no job control in this shell
{user@nowin}$ ls
server
server.c
server.o
{user@nowin}$ exit
exit
{keke@nowhere}$
```

Serveur *TCP multi-processus*

- **Recouvrement de processus et tampons de réception**
  - Imaginons le comportement suivant pour un serveur *TCP* :
    - Lire une ligne de texte depuis la connexion
    - Analyser cette ligne pour déterminer la commande de recouvrement
    - Rediriger les entrées/sorties
    - Effectuer le recouvrement qui exploitera la connexion
  - Si par confort on veut utiliser `fgets()` pour la première ligne
    - On crée un `FILE *` avec `fdopen()` (voir le module *SE1*)
    - On obtient la première ligne avec `fgets()`
      → Le tampon de reception du `FILE *` peut contenir la suite des données
    - Le contenu du tampon sera perdu lors du recouvrement !!!
    - **Il ne faut extraire de la connexion que la quantité utile**

Serveur *TCP multi-processus*

▷ Utilitaire pour extraire une ligne de texte

```
#define MAKE_READLINE_FUNCTION(fnctName,readFnct,inputType)        \
        int /* >0: nb read   0: EOF   -1: error */                 \
        fnctName(inputType input,                                  \
                 void * buffer,                                    \
                 unsigned int size)                                \
        {                                                          \
        int r;                                                     \
        unsigned int len=0;                                        \
        char * ptr=(char *)buffer;                                 \
        --size;                                                    \
        while(len<size)                                            \
          {                                                        \
          RESTART_SYSCALL(r,readFnct(input,ptr+len,sizeof(char))); \
          if(r<0) return -1;                                       \
          if(!r||(ptr[len++]=='\n')) break;                        \
          }                                                        \
        ptr[len]='\0';                                             \
        return len;                                                \
        }
```

Serveur *TCP multi-processus*

▷ Utilitaire pour extraire une quantité exacte

```
#define MAKE_READFULLY_FUNCTION(fnctName,readFnct,inputType) \
        int /* >0: nb read   0: EOF   -1: error */           \
        fnctName(inputType input,                            \
                 void * buffer,                              \
                 unsigned int size)                          \
        {                                                    \
        int r;                                               \
        unsigned int remaining=size;                         \
        char * ptr=(char *)buffer;                           \
        while(remaining)                                     \
          {                                                  \
          RESTART_SYSCALL(r,readFnct(input,ptr,remaining));  \
          if(r<0) return -1;                                 \
          if(!r) break;                                      \
          ptr+=r;                                            \
          remaining-=r;                                      \
          }                                                  \
        return size-remaining;                               \
        }
```

Serveur *TCP multi-processus*

▷ Utilitaire pour expédier une quantité exacte

```
#define MAKE_WRITEFULLY_FUNCTION(fnctName,writeFnct,outputType) \
        int /* >=0: nb written   -1: error */                   \
        fnctName(outputType output,                              \
                 const void * buffer,                            \
                 unsigned int size)                              \
        {                                                        \
        int r;                                                   \
        unsigned int remaining=size;                             \
        const char * ptr=(const char *)buffer;                   \
        while(remaining)                                         \
          {                                                      \
          RESTART_SYSCALL(r,writeFnct(output,ptr,remaining));    \
          if(r<0) return -1;                                     \
          ptr+=r;                                                \
          remaining-=r;                                          \
          }                                                      \
        return size-remaining;                                   \
        }
```

Serveur *TCP multi-threads*

- **Principe : 1 connexion → 1 thread**
  - Le programme principal effectue une boucle d'écoute (`accept()`)
    - Un *thread* est créé à chaque nouvelle connexion
    - Le *thread* partage les ressources du programme principal
  - On ne ferme ni la *socket* d'écoute ni la *socket* de dialogue
    - Le *thread* exploite librement la *socket* de dialogue
    - Le programme peut attendre immédiatement une nouvelle connexion
  - C'est l'alternative *"moderne"* au `fork()`
    - Espace d'adressage non dupliqué
    - Dialogues non cloisonnés (un plantage arrête le serveur)
    - Un *thread* peut se bloquer sans bloquer le serveur

Généralités sur les threads

- **Donner plusieurs activités à un même processus**
  - Mener plusieurs traitements bloquants
  - Gagner du temps sur une machine multi-processeurs
- **Plus efficace que plusieurs processus**
  - Commutation plus efficace (*processus légers*)
  - Espace d'adressage commun (communication simplifiée)
- **Informations propres à chaque thread**
  - Pile d'exécution, valeurs des registres
- **Informations partagées au sein du processus**
  - Code, données, descripteurs de fichiers
- **Partage dépendant de l'implémentation**
  - Signaux, propriétés ...

Généralités sur les threads

- **Exécutions indépendantes**
  - Nécessité d'une synchronisation explicite
- **Espace d'adressage commun**
  - Accès concurrents à des ressources communes
    - Mécanismes d'exclusion mutuelle
  - Écriture de traitements réentrants
    - Pas de données statiques ou globales
    - Arguments supplémentaires
  - Choix de primitives système réentrantes
    - Fonction ayant l'extension `_r` (*Posix.1c*)
    - Ex : `asctime()` et `asctime_r()`

Généralités sur les threads

▷ **Exemple de fonction non réentrante**

◇ Contenu de `buffer` indéterminé si invocations simultanées

```
const char * writeHexa(int i)
{
static char buffer[0x10];
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

▷ **Version réentrante de cette fonction**

◇ Chaque invocation utilise des données différentes (transmises)

```
char * writeHexa_r(int i,char * buffer)
{
sprintf(buffer,"0x%.8x",i);
return(buffer);
}
```

Développer avec les *Pthreads*

- **Les Pthreads : norme *Posix.1c*, la référence !**
  - Fonctionnalités de base portables
  - Influence sur les fonctionnalités habituelles (mono-tâche)
- **Compilation avec la macro `_REENTRANT`**
  - `$ cc -c -D_REENTRANT prog.c`
    - Influence sur les `.h` standards
    - Implémentation différente de certains services
- **Édition de liens avec la bibliothèque `pthread`**
  - `$ cc -o prog prog.o -lpthread`
- **Signalement des erreurs**
  - Pour la majorité des fonctions `pthread` :
    - Retour valant `0` → ok
    - Retour non nul → code d'erreur interprété comme `errno`

Conventions de nommage des fonctionnalités *Pthreads*

- `#include <pthread.h>`
- **Les types :** `pthread[_`*`objet`*`]_t`
  - *`objet`* :
    - `attr`, `mutex` ou `cond`
    - *thread* si omis
- **Les fonctions :** `pthread[_`*`objet`*`]_`*`operation`*`[_np]`
  - *`objet`* : (voir les types)
  - *`operation`* : traitement concernant le type désigné
    - `init`, `destroy` ...
    - `create`, `exit`, `join` ...
    - `lock`, `unlock`, `signal`, `broadcast` ...
  - L'extension `_np` signale un traitement non portable (spécifique à l'implémentation courante)

Identification d'un thread

- **Le processus dans sa globalité**
  - `#include <sys/types.h>`
    `#include <unistd.h>`
    `pid_t getpid(void);`
  - 1 PID par thread pour les threads en mode noyau !
- **Le thread courant**
  - `pthread_t pthread_self(void);`
- **Le test d'égalité**
  - `pthread_t` est un type opaque → pas de comparaison directe !
  - `int pthread_equal(pthread_t t1,pthread_t t2);`
  - Résultat non nul si `t1` ≡ `t2`, `0` sinon

Création d'un thread

- **La fonction** `pthread_create()` (`man 3 pthread_create`)
  - `int pthread_create(pthread_t * id,`
    `pthread_attr_t * attr,`
    `void * (*fct)(void *),`
    `void * fctArg);`
  - Crée un *thread* exécutant `fct` avec l'argument `fctArg`
  - Stocke son identifiant dans `id`
  - Le *thread* a les propriétés décrites par `attr` (propriétés par défaut si pointeur nul)
  - Retour : `0` si ok, non nul si erreur (`EAGAIN`)
  - Causes d'erreur :
    - `PTHREAD_THREADS_MAX` atteint
    - Plus assez de ressources système

Terminaison d'un thread

- **Fin de sa fonction**
  - Résultat transmis par la valeur de retour
- **La fonction** `pthread_exit()` (`man 3 pthread_exit`)
  - `void pthread_exit(void * result);`
  - Termine le *thread* courant en retournant `result`
  - Résultat lisible par `pthread_join()` (voir plus loin)
- **Fin du programme principal**
  - Fin de `main()` (ou `exit()`) ou recouvrement (par `exec()`)
    → destruction de tous les threads (il reste une activité)
  - `pthread_exit()` dans `main()` → attente de tous les *threads*

Attente d'un thread

- **La fonction** `pthread_join()` (`man 3 pthread_join`)
  - `int pthread_join(pthread_t th,void ** result);`
  - Attendre la terminaison de `th` et obtenir son résultat dans `result`
  - Les ressources de `th` sont libérées (principe similaire à `waitpid()`)
  - Retour : `0` si ok, non nul si erreur
    (Un seul `pthread_join()` sur un *thread* donné, le *thread* attendu ne doit pas être détaché, un *thread* ne peut s'attendre lui même)
- **Détachement d'un** *thread*
  - `int pthread_detach(pthread_t th);`
  - Le *thread* désigné (souvent `pthread_self()`) passe dans l'état détaché
    - `pthread_join` devient impossible sur `th`
    - Libération des ressources dès sa terminaison
  - Retour : `0` si ok, non nul si erreur (inconnu ou déjà détaché)

Créer et attendre un thread

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

void * task(void * data)
{
int i; int * nb=(int *)data; /* (void *) --> (int *) */
fprintf(stderr,"start 1\n");
for(i=0;i<*nb;i++) { /* ! busy wait ! */ } fprintf(stderr,"end 1\n");
return (void *)0;
}

int main(void)
{
int i; pthread_t th; void * result;
int nb=100000000;
if(pthread_create(&th,(pthread_attr_t *)0,&task,&nb)) /* (int *) --> (void *) */
  { fprintf(stderr,"Pb create()\n"); return 1; }
fprintf(stderr,"start 2\n");
for(i=nb/2;i<nb;i++) { /* ! busy wait ! */ } fprintf(stderr,"end 2\n");
pthread_join(th,&result);
fprintf(stderr,"quit\n");
return 0;
}
```

```
$ ./prog
start 2
start 1
end 2
end 1
quit
$
```

Créer et détacher des threads

```
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
void * task(void * data)
{
int * nb=(int *)data; /* (void *) --> (int *) */
pthread_detach(pthread_self());
sleep(1); (*nb)--; fprintf(stderr,"stop\n");
return (void *)0;
}
int main(void)
{
volatile int i; pthread_t th;
for(i=0;i<5;i++)
  {
  if(pthread_create(&th,(pthread_attr_t *)0,&task,&i)) /* (int *) --> (void *) */
    { fprintf(stderr,"Pb create()\n"); return 1; }
  }
while(i) { /* ! busy wait ! */ }
fprintf(stderr,"quit\n");
return 0;
}
```

```
$ ./prog
stop
stop
stop
stop
stop
quit
$
```

Influencer l'ordonnancement avec `sched_yield()`

```
#include <pthread.h>
#include <sched.h>
#include <stdio.h>
void * task(void * data)
{
int i;
for(i=0;i<1000000;i++)
  {
  fprintf(stderr,"%d",*((int *)data));
  /* sched_yield(); */
  }
return (void *)0;
}
int main(void)
{
int n1=1,n2=2; pthread_t t1,t2; void * result;
if(pthread_create(&t1,(pthread_attr_t *)0,&task,&n1))
  { fprintf(stderr,"Pb thread 1\n"); return 1; }
if(pthread_create(&t2,(pthread_attr_t *)0,&task,&n2))
  { fprintf(stderr,"Pb thread 2\n"); return 1; }
pthread_join(t1,&result); pthread_join(t2,&result);
return 0;
}
```

```
$ ./prog # sans sched_yield()
11111111111111111111111111111111111111111
1111 ... 11122222222222222222222222222222
222222222222222222222 ... 11111111111111
11111111111111111111111111111111 ... 11
1111122222222 ... 22222222222222222222222
2222
$
$ ./prog # avec sched_yield()
12121212121212121212121212121212121212121
212121212121212121212 ... 2121212121212
1212
$
```

Serveur *TCP multi-threads*

```
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <pthread.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

#define RESTART_SYSCALL(result,syscall) \
        do { (result)=(syscall); } while(((result)<0)&&(errno==EINTR));

typedef struct
  {
  int fd;
  /* eventuellement d'autres champs ... */
  } ConnectInfo;
```

Serveur *TCP multi-threads*

```
void * service(void * data)
{
ConnectInfo * info=(ConnectInfo *)data; /* (void *) --> (ConnectInfo *) */
char buffer[0x100]; int r;
pthread_detach(pthread_self()); /* pas besoin de pthread_join() */
RESTART_SYSCALL(r,read(info->fd,buffer,0x100));
if(r==-1) { perror("read"); }
else { buffer[r]='\0'; fprintf(stderr,"[%s]\n",buffer); }
RESTART_SYSCALL(r,close(info->fd));
free(info); /* alouee dans le programme principal */
return (void *)0;
}

int main(void)
{
int listenFd,r,on;
struct sockaddr_in addr;
RESTART_SYSCALL(listenFd,socket(PF_INET,SOCK_STREAM,0));
if(listenFd==-1) { perror("socket"); return EXIT_FAILURE; }
on=1; /* autoriser l'option SO_REUSEADDR */
RESTART_SYSCALL(r,setsockopt(listenFd,SOL_SOCKET,SO_REUSEADDR,&on,sizeof(int)));
if(r==-1) { perror("setsockopt SO_REUSEADDR"); return EXIT_FAILURE; }
```

Serveur *TCP multi-threads*

```
memset(&addr,0,sizeof(struct sockaddr_in));
addr.sin_family=AF_INET;
addr.sin_port=htons(9876);
addr.sin_addr.s_addr=htonl(INADDR_ANY);
RESTART_SYSCALL(r,bind(listenFd,(const struct sockaddr *)&addr,
                                sizeof(struct sockaddr_in)));
if(r==-1) { perror("bind"); return EXIT_FAILURE; }
if(listen(listenFd,5)==-1) { perror("listen"); return EXIT_FAILURE; }
for(;;)
  {
  pthread_t th;
  ConnectInfo * info=(ConnectInfo *)malloc(sizeof(ConnectInfo)); /* pour le thread */
  RESTART_SYSCALL(info->fd,accept(listenFd,(struct sockaddr *)0,(socklen_t *)0));
  if(info->fd==-1) { perror("accept"); break; }
  if(pthread_create(&th,(pthread_attr_t *)0,
                    &service,info)) /* (ConnectInfo *) --> (void *) */
    { fprintf(stderr,"pb thread\n"); break; }
  }
RESTART_SYSCALL(r,close(listenFd));
pthread_exit((void *)0); /* attendre que tous les threads soient termines */
return EXIT_SUCCESS;
}
```

Synchronisation *multi-threads*

- **Problématique**
  - Plusieurs *threads* peuvent utiliser les mêmes données simultanément
    - Le opérations ne sont généralement pas atomiques
    - Elles peuvent être dans états incohérents vis-à-vis de chaque traitement
  - Il faut garantir l'accès exclusif aux ressources manipulées
- **Sémaphores d'exclusion mutuelle** (verrous)
  - Limiter l'accès à une donnée
  - Un seul accès à la fois
- **Variables conditions**
  - Attendre qu'une condition logique soit vérifiée
    - Les autres *threads* font évoluer cette condition
  - Blocage du traitement en attendant

Synchronisation *multi-threads*

▷ Exemple : gestion d'un tableau sans synchronisation

```
Machin * machins[MAX_NB_MACHINS];
int nbMachins=0;

void ajouterMachin(Machin * m)
{
if(nbMachins<MAX_NB_MACHINS)
  {
  machins[nbMachins++]=m;
  }
}

void retirerMachin(Machin * m)
{
int i;
for(i=0;i<nbMachins;i++)
  {
  if(machins[i]==m)
    { machins[i]=machins[--nbMachins]; break; }
  }
}
```

Les sémaphores d'exclusion mutuelle

- **Création d'un verrou** (`man 3 pthread_mutex_init`)
  - Type : `pthread_mutex_t`
  - Initialisation statique
    - `pthread_mutex_t myMutex=PTHREAD_MUTEX_INITIALIZER;`
  - Initialisation dynamique
    - ```
      int pthread_mutex_init(pthread_mutex_t * mtx,
                             pthread_mutexattr_t * attr);
      ```
    - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'un verrou** (`man 3 pthread_mutex_destroy`)
  - `int pthread_mutex_destroy(pthread_mutex_t * mtx);`
  - Le verrou n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si le verrou est monopolisé → erreur `EBUSY`
      ```

Les sémaphores d'exclusion mutuelle

- **Opération de verrouillage** (`man 3 pthread_mutex_lock`)
  - `int pthread_mutex_lock(pthread_mutex_t * mtx);`
  - Si le verrou est libre
    - Il est monopolisé par le *thread* courant
    - Le *thread* courant peut utiliser la ressource protégée (il devra libérer le verrou après)
  - Si le verrou n'est pas libre
    - Le *thread* courant est bloqué jusqu'à la libération du verrou
- **Opération de déverrouillage** (`man 3 pthread_mutex_unlock`)
  - `int pthread_mutex_unlock(pthread_mutex_t * mtx);`
  - Libère le verrou
  - Si des *threads* sont bloqués en attente sur ce verrou
    - L'un d'eux (lequel ?) est débloqué et monopolise le verrou

Les sémaphores d'exclusion mutuelle

- **Tentative de verrouillage** (`man 3 pthread_mutex_trylock`)
  - `int pthread_mutex_trylock(pthread_mutex_t * mtx);`
  - Si le verrou est libre
    - Il est monopolisé par cet appel qui retourne `0`
    - Le *thread* courant peut utiliser la ressource protégée (il devra libérer le verrou après)
  - Si le verrou n'est pas libre
    - Cet appel retourne immédiatement un résultat non nul
    - Il ne faut pas utiliser les ressources protégées par le verrou
  - Attention : l'interpretation du résultat n'est pas intuitive (`if(...)`)
    - `0` : OK $\rightarrow$ verrouillage effectué
    - `!=0` : verrouillage impossible

Les sémaphores d'exclusion mutuelle

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

void * task(void * data)
{
unsigned long i,n;
pthread_mutex_t * mtx=(pthread_mutex_t *)data;
fprintf(stderr,"begin task() in %u\n",
        (unsigned int)pthread_self()); /* ugly cast ! */
for(n=0;n<2;n++)
  {
  pthread_mutex_lock(mtx); /* begin critical section */
  fprintf(stderr,"  thread %u -->\n",
          (unsigned int)pthread_self()); /* ugly cast ! */
  for(i=0;i<50*1000*1000;i++) {}
  fprintf(stderr,"  thread %u <--\n",
          (unsigned int)pthread_self()); /* ugly cast ! */
  pthread_mutex_unlock(mtx); /* end critical section */
  }
fprintf(stderr,"end task() in %u\n",
        (unsigned int)pthread_self()); /* ugly cast ! */
return (void *)0;
}
```

```
$ ./prog
begin task() in 3085118352
  thread 3085118352 -->
begin task() in 3076729744
begin task() in 3068341136
  thread 3085118352 <--
  thread 3076729744 -->
  thread 3076729744 <--
  thread 3068341136 -->
  thread 3068341136 <--
  thread 3076729744 -->
  thread 3076729744 <--
  thread 3068341136 -->
end task() in 3076729744
  thread 3068341136 <--
end task() in 3068341136
  thread 3085118352 -->
  thread 3085118352 <--
end task() in 3085118352
$
```

Les sémaphores d'exclusion mutuelle

```
int main(void)
{
pthread_mutex_t mtx;
pthread_t th[3];
int i;
void * result;
pthread_mutex_init(&mtx,(pthread_mutexattr_t *)0);
for(i=0;i<3;i++)
  {
  if(pthread_create(&th[i],(pthread_attr_t *)0,task,&mtx))
    { fprintf(stderr,"Pb create\n"); return 1; }
  }
for(i=0;i<3;i++)
  {
  if(pthread_join(th[i],&result))
    { fprintf(stderr,"Pb join\n"); return 1; }
  }
if(pthread_mutex_destroy(&mtx))
  { fprintf(stderr,"Pb mutex destroy\n"); return 1; }
return 0;
}
```

Les variables conditions

- **Création d'une condition** (`man 3 pthread_cond_init`)
  - Type : `pthread_cond_t` (doit être associé à un `pthread_mutex_t`)
  - Initialisation statique
    - `pthread_cond_t myCond=PTHREAD_COND_INITIALIZER;`
  - Initialisation dynamique
    - `int pthread_cond_init(pthread_cond_t * cond, pthread_condattr_t * attr);`
    - Généralement `attr` est nul (initialisation par défaut)
- **Destruction d'une condition** (`man 3 pthread_cond_destroy`)
  - `int pthread_cond_destroy(pthread_cond_t * cond);`
  - La condition n'est plus utilisable
  - Retour : `0` si ok, non nul si erreur
  - Erreur si la condition est utilisée → erreur `EBUSY`

Les variables conditions

▷ **Attente d'une condition** (`man 3 pthread_cond_wait`)

◇ `int pthread_cond_wait(pthread_cond_t * cond, pthread_mutex_t * mtx);`

◇ Bloque le *thread* courant
→ débloqué quand une modification est signalée sur `cond`
→ évite l'attente active

◇ `cond` n'a aucune valeur logique ! (sert à la synchronisation)

◇ `mtx` doit être monopolisé avant et libéré après

◇ Interruptible par les signaux → relance

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
while(!conditionIsSatisfied())
  pthread_cond_wait(&cond,&mtx);
pthread_mutex_unlock(&mtx);
```

Les variables conditions

▷ **Attente temporisée d'une condition**

(`man 3 pthread_cond_timedwait`)

◇
```
int pthread_cond_timedwait(pthread_cond_t * cond,
                           pthread_mutex_t * mtx,
                           const struct timespec * date);
```

◇ Même principe que `pthread_cond_wait()`

◇ Renvoie `ETIMEDOUT` si `date` est dépassée

◇ `date` est une limite, pas un délai !

- Prendre la date courante (`time()`, `gettimeofday()`)
- Ajouter un délai (`struct timespec`, `man 3 nanosleep`)
  champ `tv_sec` : nombre de secondes
  champ `tv_nsec` : nombre de nano-secondes dans la dernière seconde
- Basé sur le temps universel (pas de fuseau horaire)

Les variables conditions

▷ **Signaler une condition** (`man 3 pthread_cond_broadcast`)

◇ `int pthread_cond_broadcast(pthread_cond_t * cond);`

◇ Débloque tous les *threads* attendant la condition `cond`

◇ Le verrou associé à `cond` doit être monopolisé avant et libéré après

◇ Démarche usuelle :

```
pthread_mutex_lock(&mtx);
/* make changes on sensible values */
pthread_cond_broadcast(&cond);
pthread_mutex_unlock(&mtx);
```

◇ `int pthread_cond_signal(pthread_cond_t * cond);`

○ Ne débloque qu'un *thread* parmi tous ceux qui attendent `cond`

○ Un peu plus efficace que `pthread_cond_broadcast`

○ Bien moins général (incohérence si plusieurs *threads* en attente !)

Synchronisation *multi-threads*

▷ Exemple : gestion d'un tableau avec synchronisation

```
Machin * machins[MAX_NB_MACHINS];
int nbMachins=0;
pthread_mutex_t machinMtx;
pthread_cond_t machinCond;
...
pthread_mutex_init(&machinMtx,(pthread_mutexattr_t *)0);
pthread_cond_init(&machinCond,(pthread_condattr_t *)0);

...
pthread_mutex_lock(&machinMtx);
while(!nbMachins) /* attendre qu'il y ait quelque chose */
  pthread_cond_wait(&machinCond,&machinMtx);
pthread_mutex_unlock(&machinMtx);
/* On peut utiliser le contenu du tableau */
/* en prenant tout de meme des precautions */
...
```

Synchronisation *multi-threads*

```
void ajouterMachin(Machin * m)
{
pthread_mutex_lock(&machinMtx);
if(nbMachins<MAX_NB_MACHINS)
  {
  machins[nbMachins++]=m;
  }
pthread_cond_broadcast(&machinCond);
pthread_mutex_unlock(&machinMtx);
}

void retirerMachin(Machin * m)
{
int i;
pthread_mutex_lock(&machinMtx);
for(i=0;i<nbMachins;i++)
  {
  if(machins[i]==m)
    { machins[i]=machins[--nbMachins]; break; }
  }
pthread_cond_broadcast(&machinCond);
pthread_mutex_unlock(&machinMtx);
}
```

Bilan sur les architectures client/serveur *TCP*

- **Scrutation passive**
  - Solution économe et réactive
  - Bien adaptée aux services rendus *"instantanément"*
  - Risque de bloquage général en cas de maladresse de programmation
- **Serveur** *multi-processus*
  - Solution assez lourde en ressources
  - Bien adaptée pour le cloisonnement des services
    - Peu de risque de bloquage général, les plantages restent localisés
  - Bien adaptée au recouvrement par un autre programme
- **Serveur** *multi-threads*
  - Solution assez lourde en ressources (moins que les processus)
  - Peu de risques de bloquage général
  - Bien adaptée aux relations inter-clients mais synchronisation stricte !