```
#include <unistd.h>
pid_t getpid (void);
pid_t getppid (void);
pid_t fork (void);
```

- -1 Echec création processus (seul le père revient)
- 0 On est dans le fils
- > 0 On est dans le père = PID du fils

```
#include <stdio.h>
void perror(const char *s);
#include <stdlib.h>
void exit (int status); //EXIT_SUCCESS, EXIT_FAILURE
void au_revoir (void) { printf ("Au revoir\n"); }
void bye_bye (void) { printf ("Bye bye\n"); }
int main () {
        atexit (au_revoir);
        atexit (bye_bye);
        printf ("Avant exit ...\n");
        exit (0);
        printf ("Après exit\n"); }
```

Exécution : Avant exit … → Bye bye → Au revoir

```
#include <sys/types.h>
#include <sys/wait.h>
pid_t wait (int *stat_infos);
pid_t waitpid (pid_t pid, int *stat_infos, int options);
```

wait attend la fin de n'importe quel fils et renvoie son PID;
waitpid attend la fin du fils ayant le PID pid;
options = 0 : attente bloquante
options = WNOHANG : retour immédiat
Appels bloquants silencieusement repris : act.sa_flags |= SA_RESTART;

```
unsigned int alarm (unsigned int seconds);
```

Afficher la table des processus : ps, pstree, top

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

(si -1: `a tous les processus permis sauf init et lui-même).
Renvoie 0 succès, -1 erreur.
kill (pid, 0) = tester s'il existe (même zombie)

```
#include <unistd.h>
int pipe(int pipefd[2]);
```



```
#include <unistd.h>
int dup (int oldfd);
int dup2 (int oldfd, int newfd);
void FD_ZERO (fd_set *set);
void FD_SET (int fd, fd_set *set);
int FD_ISSET (int fd, fd_set *set);
void FD_CLR (int fd, fd_set *set);
int select (int nfds, fd_set *readfds, fd_set *writefds,
fd_set *exceptfds, struct timeval *timeout);
struct timeval t;
t.tv_sec = 0; t.tv_usec = 200000;
select (0, NULL, NULL, NULL, &t);
int mkfifo(const char *pathname, mode_t mode);
```

SERVEUR UDP : socket, bind recvfrom, sendto, close
CLIENT UDP : socket, bind, sendto, recvfrom, close
SERVEUR TCP : socket, bind, listen, accept, read, write, close
CLIENT TCP : socket, bind, connect, write, read, close

```
uint16_t htons (uint16_t hostshort);
uint16_t ntohs (uint16_t netshort);
uint32_t htonl (uint32_t hostlong);
uint32_t ntohl (uint32_t netlong);
int atoi( const char *str );
```

Attribuer un port libre : struct sockaddr_in adr;
adr.sin_port = htons (0);
L'afficher : **socklen_t addrlen = sizeof (struct sockaddr_in);**

```
if (getsockname (soc, (struct sockaddr*) &addr,
&addrlen) < 0) exit (1);
printf ("Port attribu´e = %d\n", ntohs (adr.sin_port));
```

Créer socket locale : adr.sin_family = AF_INET;
adr.sin_port = htons (port);
adr.sin_addr.s_addr = htonl (INADDR_ANY);
Adresse IPv4 en chaîne de caractère "a.b.c.d" : char *inet_ntoa(struct in_addr in);
GET: demander une ressource
HEAD: demander informations sur ressource
POST: transmettre des informations
PUT: transmettre par URL
PATCH: modification partielle
OPTIONS: obtenir les options de communication
CONNECT: pour utiliser un proxy comme tunnel
TRACE: ´echo de la requête pour diagnostic
DELETE: pour supprimer une ressource

```
char *s = "789 65";
sscanf (s, "%d%n", &x, &p); → x = 789, p = 3
sscanf (s+p, "%d", &y); → y = 65
```

```
#include <stdio.h>      /* printf, fgets */
#include <stdlib.h>     /* exit */
#include <unistd.h>     /* close */
#include <string.h>     /* strlen */
#include <sys/types.h>  /* open, socket, bind, sendto, recvfrom, wait */
#include <signal.h>     /* sigaction */
#include <sys/wait.h>   /* wait */
#include <sys/stat.h>   /* open */
#include <fcntl.h>      /* open */
#include <sys/socket.h> /* socket, bind, sendto, recvfrom, getsockname */
#include <sys/un.h>     /* socket domaine AF_UNIX */
#include <netinet/ip.h> /* socket domaine AF_INET */
#include <arpa/inet.h>  /* inet_ntoa */
#include <netdb.h>      /* gethostbyname */
#include <sys/time.h>   /* gettimeofday */
#include <time.h>       /* time, gettimeofday */
#include <errno.h>      /* errno */

/* Prototypes */
void bor_perror (const char *funcname);
int bor_signal (int sig, void (*h)(int), int options);

ssize_t bor_read (int fd, void *buf, size_t count);
ssize_t bor_read_str (int fd, char *buf, size_t count);
ssize_t bor_write (int fd, const void *buf, size_t count);
ssize_t bor_write_str (int fd, const char *buf);

int bor_listen (int soc, int max_pending);

int bor_create_socket_un (int type, const char *path, struct sockaddr_un *sa);
void bor_set_sockaddr_un (const char *path, struct sockaddr_un *sa);
int bor_bind_un (int soc, struct sockaddr_un *sa);
int bor_connect_un (int soc, const struct sockaddr_un *sa);
int bor_accept_un (int soc, struct sockaddr_un *sa);
ssize_t bor_recvfrom_un (int soc, void *buf, size_t count, struct sockaddr_un *sa);
ssize_t bor_recvfrom_un_str (int soc, char *buf, size_t count, struct sockaddr_un *sa);
ssize_t bor_sendto_un (int soc, const void *buf, size_t count, const struct sockaddr_un *sa);
ssize_t bor_sendto_un_str (int soc, const char *buf, const struct sockaddr_un *sa);

int bor_create_socket_in (int type, int port, struct sockaddr_in *sa);
void bor_set_sockaddr_in (int port, uint32_t ipv4, struct sockaddr_in *sa);
int bor_getsockname_in (int soc, struct sockaddr_in *sa);
char *bor_adrtoa_in (struct sockaddr_in *sa);
int bor_bind_in (int soc, struct sockaddr_in *sa);
int bor_resolve_address_in (const char *host, int port, struct sockaddr_in *sa);
int bor_connect_in (int soc, const struct sockaddr_in *sa);
int bor_accept_in (int soc, struct sockaddr_in *sa);
ssize_t bor_recvfrom_in (int soc, void *buf, size_t count, struct sockaddr_in *sa);
ssize_t bor_recvfrom_in_str (int soc, char *buf, size_t count, struct sockaddr_in *sa);
ssize_t bor_sendto_in (int soc, const void *buf, size_t count, const struct sockaddr_in *sa);
ssize_t bor_sendto_in_str (int soc, const char *buf, const struct sockaddr_in *sa);
```

```c
/* Affiche le message de perror en conservant la valeur de errno.*/
void bor_perror (const char *funcname)
{
   int e = errno; perror (funcname); errno = e;
}
/*  Place le handler de signal void h(int) pour le signal sig avec sigaction().
 *  Le signal est automatiquement masque pendant sa delivrance.
 *  Si options = 0,
 *    - les appels bloquants sont interrompus avec retour -1 et errno = EINTR.
 *    - le handler est rearme automatiquement apres chaque delivrance de signal.
 *  si options est une combinaison bit a bit de
 *    - SA_RESTART : les appels bloquants sont silencieusement repris.
 *    - SA_RESETHAND : le handler n'est pas rearme.
 *  Renvoie le resultat de sigaction. Verbeux.
*/
int bor_signal (int sig, void (*h)(int), int options)
{
   int r;
   struct sigaction s;
   s.sa_handler = h;
   sigemptyset (&s.sa_mask);
   s.sa_flags = options;
   r = sigaction (sig, &s, NULL);
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Lecture d'au plus count octets dans fd et memorisation dans buf.
 * Renvoie le resultat de read(). Verbeux.
*/
ssize_t bor_read (int fd, void *buf, size_t count)
{
   ssize_t r = read (fd, buf, count);
   if (r < 0) bor_perror (__func__);
   else if (r == 0 && count != 0)
      fprintf (stderr, "%s: EOF detected for fd %d\n", __func__, fd);
   return r;
}
```

```c
/* Lecture d'au plus count-1 caracteres dans fd et memorisation dans buf.
 * Ajout d'un '\0' terminal dans buf en cas de succes.
 * Renvoie le resultat de read(). Verbeux.
*/
ssize_t bor_read_str (int fd, char *buf, size_t count)
{
   if (count == 0) {
      fprintf (stderr, "%s: count>0 expected\n", __func__);
      errno = EINVAL;
      return -1;
   }
   ssize_t r = bor_read (fd, buf, count-1);
   if (r >= 0) buf[r] = '\0';
   return r;
}
/* Ecriture d'au plus count octets dans fd provenant de buf.
 * Renvoie le resultat de write(). Verbeux.
*/
ssize_t bor_write (int fd, const void *buf, size_t count)
{
   ssize_t r = write (fd, buf, count);
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Ecriture de la chaine de caracteres buf dans fd.
 * Renvoie le resultat de write(). Verbeux.
*/
ssize_t bor_write_str (int fd, const char *buf)
{
   return bor_write (fd, buf, strlen (buf));
}
/* Transforme la socket ouverte soc de type SOCK_STREAM en socket d'ecoute.
 * max_pending est la taille de la file des connexions en attente.
 * Renvoie le resultat de listen(). Verbeux.
*/
int bor_listen (int soc, int max_pending)
{
   int r = listen (soc, max_pending);
   if (r < 0) bor_perror (__func__);
   return r;
}
```

```
/*---------------------- D O M A I N E   U N I X ---------------------------*/
/* Cree une socket du domaine AF_UNIX selon le type, fabrique une adresse
 * locale sa avec path puis attache la socket a sa.
 * Renvoie la socket attachee >= 0, sinon -1 erreur. Verbeux.

int bor_create_socket_un (int type, const char *path, struct sockaddr_un *sa)
{
   int soc = socket (AF_UNIX, type, 0);
   if (soc < 0) { bor_perror ("socket un"); return -1; }

   bor_set_sockaddr_un (path, sa);
   if (bor_bind_un (soc, sa) < 0) { close (soc); return -1; }

   fprintf (stderr, "%s: socket \"%s\" opened\n", __func__, sa->sun_path);
   return soc;
}
/* Construit une adresse sa du domaine AF_UNIX avec le chemin path.
 * Silencieux.
*/
void bor_set_sockaddr_un (const char *path, struct sockaddr_un *sa)
{
   sa->sun_family = AF_UNIX;
   strncpy (sa->sun_path, path, UNIX_PATH_MAX);
   sa->sun_path[UNIX_PATH_MAX-1] = 0;
}
/* Attachement d'une socket de domaine AF_UNIX a une adresse sockaddr_un.
 * Renvoie le resultat de bind(). Verbeux.
*/
int bor_bind_un (int soc, struct sockaddr_un *sa)
{
   int r = bind (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_un));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Connexion a un serveur TCP/UN.
 * Renvoie le resultat de connect. Verbeux.
*/
```



```
int bor_connect_un (int soc, const struct sockaddr_un *sa)
{
   int r = connect (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_un));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Accepte une connexion en attente.
 * Renvoie le resultat de accept. Verbeux.
 */
int bor_accept_un (int soc, struct sockaddr_un *sa)
{
   socklen_t adrlen = sizeof(struct sockaddr_un);
   int r = accept (soc, (struct sockaddr *) sa, &adrlen);
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Reception d'un datagramme d'au plus count octets.
 * Renvoie le resultat de recvfrom. Verbeux.
*/
ssize_t bor_recvfrom_un (int soc, void *buf, size_t count, struct sockaddr_un *sa)
{
   socklen_t adrlen = sizeof(struct sockaddr_un);
   ssize_t  r = recvfrom (soc, buf, count, 0, (struct sockaddr *) sa, &adrlen);
   if (r < 0) bor_perror (__func__);
   return r;
}
ssize_t bor_recvfrom_un_str (int soc, char *buf, size_t count, struct sockaddr_un *sa)
{
   ssize_t r = bor_recvfrom_un (soc, buf, count-1, sa);
   if (r >= 0) buf[r] = '\0';
   return r;
}
/* Envoi d'un datagramme.
 * Renvoie le resultat de sendto. Verbeux.
*/
ssize_t bor_sendto_un (int soc, const void *buf, size_t count, const struct sockaddr_un *sa)
{
   ssize_t r = sendto (soc, buf, count, 0, (struct sockaddr *) sa,
      sizeof(struct sockaddr_un));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Envoi par datagramme de la chaine de caracteres buf dans soc.
 * Renvoie le resultat de sendto(). Verbeux.
*/
ssize_t bor_sendto_un_str (int soc, const char *buf, const struct sockaddr_un *sa)
{
   return bor_sendto_un (soc, buf, strlen (buf), sa);
}
```

```
/* Reception d'un datagramme d'au plus count-1 caracteres.
 * Ajout d'un '\0' terminal dans buf en cas de succes.
 * Renvoie le resultat de read(). Verbeux.*/
/*-------------------  D O M A I N E   I N T E R N E T ----------------------*/
/* Cree une socket du domaine AF_INET selon le type, fabrique une adresse locale
 * sa avec le port (0 pour obtenir un port libre), puis attache la socket a sa.
 * Renvoie la socket attachee >= 0, sinon -1 erreur. Verbeux.
*/
int bor_create_socket_in (int type, int port, struct sockaddr_in *sa)
{
   int soc = socket (AF_INET, type, 0);
   if (soc < 0) { bor_perror ("socket in"); return -1; }
   bor_set_sockaddr_in (port, INADDR_ANY, sa);
   if (bor_bind_in (soc, sa) < 0) { close (soc); return -1; }
   if (bor_getsockname_in (soc, sa) < 0) { close (soc); return -1; }
   fprintf (stderr, "%s: port %d opened\n", __func__, ntohs(sa->sin_port));
   return soc;
}
/* Construit une adresse sa du domaine AF_INET avec le port et l'adresse ipv4.
 * Donner port = 0 pour obtenir un port libre ; ipv4 = INADDR_ANY pour l'adresse
 * locale. Silencieux.
*/
void bor_set_sockaddr_in (int port, uint32_t ipv4, struct sockaddr_in *sa)
{
   sa->sin_family = AF_INET;
   sa->sin_port = htons (port);
   sa->sin_addr.s_addr = htonl(ipv4);
}
/* Recuperation de l'adresse reelle et du port sous forme Network.
 * Renvoie le resultat de getsockname. Verbeux.
*/
int bor_getsockname_in (int soc, struct sockaddr_in *sa)
{
   socklen_t adrlen = sizeof(struct sockaddr_in);
   int r = getsockname (soc, (struct sockaddr *) sa, &adrlen);
   if (r < 0) bor_perror (__func__);
   return r;
}
```



```
char *bor_adrtoa_in (struct sockaddr_in *sa)
{
   static char s[32];
   sprintf (s, "%s:%d", inet_ntoa (sa->sin_addr), ntohs (sa->sin_port));
   return s;
}
/* Attachement d'une socket de domaine AF_INET a une adresse sockaddr_in.
 * Renvoie le resultat de bind(). Verbeux.
*/
int bor_bind_in (int soc, struct sockaddr_in *sa)
{
   int r = bind (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_in));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Resout une adresse IPV4 sous la forme "nom_local" ou decimale "1.2.3.4"
 * ou FQDN "machine.domaine.tld", puis la stocke avec le port dans sa.
 * Renvoie 0 succes ou -1 erreur. Verbeux.
*/
int bor_resolve_address_in (const char *host, int port, struct sockaddr_in *sa)
{
   sa->sin_family = AF_INET;
   sa->sin_port = htons (port);

   struct hostent *hp;
   if ((hp = gethostbyname (host)) == NULL)
     { herror (__func__); return -1; }
   memcpy (&sa->sin_addr.s_addr, hp->h_addr, hp->h_length);
   return 0;
}
/* Connexion a un serveur TCP/IP.
 * Renvoie le resultat de connect. Verbeux.
*/
int bor_connect_in (int soc, const struct sockaddr_in *sa)
{
   int r = connect (soc, (struct sockaddr *) sa, sizeof(struct sockaddr_in));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Accepte une connexion en attente.
 * Renvoie le resultat de accept. Verbeux.
*/
int bor_accept_in (int soc, struct sockaddr_in *sa)
{
   socklen_t adrlen = sizeof(struct sockaddr_in);
   int r = accept (soc, (struct sockaddr *) sa, &adrlen);
   if (r < 0) bor_perror (__func__);
   return r;
}
```

```c
/* Renvoie l'adresse d'une chaine en memoire statique contenant l'adresse IPv4
 * sous la forme "a.b.c.d:port", de facon a pouvoir l'afficher. Silencieux.
/* Reception d'un datagramme.
 * Renvoie le resultat de recvfrom. Verbeux.
*/
ssize_t bor_recvfrom_in (int soc, void *buf, size_t count, struct sockaddr_in *sa)
{
   socklen_t adrlen = sizeof(struct sockaddr_in);
   ssize_t r = recvfrom (soc, buf, count, 0, (struct sockaddr *) sa, &adrlen);
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Reception d'un datagramme d'au plus count-1 caracteres.
 * Ajout d'un '\0' terminal dans buf en cas de succes.
 * Renvoie le resultat de read(). Verbeux.
*/
ssize_t bor_recvfrom_in_str (int soc, char *buf, size_t count, struct sockaddr_in *sa)
{
   ssize_t r = bor_recvfrom_in (soc, buf, count-1, sa);
   if (r >= 0) buf[r] = '\0';
   return r;
}
/* Envoi d'un datagramme.
 * Renvoie le resultat de sendto. Verbeux.
*/
ssize_t bor_sendto_in (int soc, const void *buf, size_t count, const struct sockaddr_in *sa)
{
   ssize_t r = sendto (soc, buf, count, 0, (struct sockaddr *) sa,
      sizeof(struct sockaddr_in));
   if (r < 0) bor_perror (__func__);
   return r;
}
/* Envoi par datagramme de la chaine de caracteres buf dans soc.
 * Renvoie le resultat de sendto(). Verbeux.
*/
ssize_t bor_sendto_in_str (int soc, const char *buf, const struct sockaddr_in *sa)
{
   return bor_sendto_in (soc, buf, strlen (buf), sa);
}
```

```
/*
   Usage :
   int h1, h2, res;
   h1 = bor_timer_add (delai1, data1);
   h2 = bor_timer_add (delai2, data2);
   res = select (..., bor_timer_delay());
   if (res == 0) {
      handle = bor_timer_handle();
      data = bor_timer_data();
      if (handle == h1) {
         printf ("delai1 ecoule");
      } else if (handle == h2) {
         printf ("delai2 ecoule");
      }
      bor_timer_remove (handle);
   }
*/
typedef struct {
   int handle;
   void *data;
   struct timeval expiration;
} bor_timer_struct;

#define BOR_TIMER_MAX 1000

bor_timer_struct bor_timer_list[BOR_TIMER_MAX];
int bor_timer_nb = 0;
int bor_timer_uniq = 0;
/* Ajoute un timer dont l'echeance sera dans delay millisecondes ;
  data est l'adresse d'une donnee quelconque, que l'on pourra recuperer
  lorsque le timer arrivera a echeance.
  Renvoie un "handle", c'est-a-dire un numero unique de timer, qui
  permettra de reconnaitre quel est le timer arrive a echeance,
  ou encore permettra de supprimer ce timer.

  L'insertion se fait par dichotomie dans une liste triee sur la date
  d'expiration ; le prochain timer est donc toujours en position 0.
  Verbeux.
*/
```

```
int bor_timer_add (unsigned long delay, void *data)
{
   int m1, m2, mid;
   struct timeval t, *mt;
   if (bor_timer_nb >= BOR_TIMER_MAX) {
      fprintf (stderr, "bor_timer_add: erreur, trop de timers\n");
      return -1;
   }
   /* Recupere date courante */
   gettimeofday (&t, NULL);
   /* Calcule date d'expiration */
   t.tv_usec += delay * 1000;   /* delay en millisecondes */
   if (t.tv_usec > 1000000) {
      t.tv_sec += t.tv_usec / 1000000;
      t.tv_usec %= 1000000;
   }
   /* Recherche dichotomique */
   m1 = 0; m2 = bor_timer_nb;
   while (m2-m1 > 0) {
      mid = (m1+m2) / 2;    /* Milieu tq m1 <= mid < m2 */
      mt = &bor_timer_list[mid].expiration;
      if ( mt->tv_sec < t.tv_sec ||
         (mt->tv_sec == t.tv_sec && mt->tv_usec < t.tv_usec) )
          m1 = mid+1;   /* t doit aller dans [mid+1 .. m2] */
      else m2 = mid;     /* t doit aller dans [m1 .. mid]   */
   }
   /* Insere en position m1 */
   if (m1 < bor_timer_nb)
      memmove (bor_timer_list+m1+1,   /* dest */
               bor_timer_list+m1,     /* src */
               (bor_timer_nb-m1)*sizeof(bor_timer_struct));
   bor_timer_nb++;
   bor_timer_list[m1].handle = bor_timer_uniq++;
   bor_timer_list[m1].data = data;
   bor_timer_list[m1].expiration = t;
   return bor_timer_list[m1].handle;
}
```

```
/* Supprime un timer a partir de son handle.
 * On maintient le tableau trie. Silencieux.
*/
void bor_timer_remove (int handle)
{
   int i;
   if (handle < 0) return;
   for (i = 0; i < bor_timer_nb; i++)
      if (bor_timer_list[i].handle == handle) {
         memmove (bor_timer_list+i,    /* dest */
                  bor_timer_list+i+1,  /* src */
                  (bor_timer_nb-i-1)*sizeof(bor_timer_struct));
         bor_timer_nb--;
         break;
      }
}
/* Renvoie le delai entre la date courante et le prochain timer,
   a passer directement a select(). Silencieux.
*/
struct timeval *bor_timer_delay ()
{
   static struct timeval t;
   /* Aucun timer */
   if (bor_timer_nb == 0) return NULL;
   /* Recupere date courante */
   gettimeofday(&t, NULL);
   /* Le prochain timeout est bor_timer_list[0].expiration ;
      on calcule la difference avec la date courante */
   t.tv_sec  = bor_timer_list[0].expiration.tv_sec  - t.tv_sec;
   t.tv_usec = bor_timer_list[0].expiration.tv_usec - t.tv_usec;
   if (t.tv_usec < 0) { t.tv_usec += 1000000; t.tv_sec -= 1;}
   if (t.tv_sec  < 0) t.tv_sec = t.tv_usec = 0;
   /* printf ("Timeout dans %d s  %6d us\n", (int)t.tv_sec, (int)t.tv_usec); */
   /* Renvoie adresse statique du struct */
   return &t;
}
```

```
/* Renvoie le handle du prochain timer, sinon -1. Silencieux.
*/
 int bor_timer_handle ()
{
   if (bor_timer_nb == 0) return -1;
   return bor_timer_list[0].handle;
}
/* Renvoie la data du prochain timer, sinon NULL. Silencieux.
*/
void *bor_timer_data ()
{
   if (bor_timer_nb == 0) return NULL;
   return bor_timer_list[0].data;
}
```

```
int lire_texte (int fd, char *buf, int buf_size)
{
        int k, pos = 0;
        while (1) {
        k = read (fd, buf+pos, buf_size-1-pos);
        if (k < 0) { perror ("read"); return -1; }
        if (k == 0) break;
        pos += k;
        }
        buf[pos] = '\0';
        return pos;
}
```

%d Une donnée entière de type int
%ld Une donnée entière de type long.
%f Une donnée décimale de type float.
%lf Une donnée décimale de type double.
%c Une donnée de type caractère.
%p Une donnée de type adresse en mémoire.
Cette adresse sera présentée sous forme hexadécimale.
%s Une donnée de type chaîne de caractères.
Attention, tout séparateur (blanc, tabulation, retour à la ligne) interrompra la lecture.
%[characters] Un donnée de type chaîne de caractères,
constituée que de caractères parmis ceux spécifiés.
%[^characters] Un donnée de type chaîne de caractères,
constituée de tous caractères sauf les caractères spécifiés.

```
int fscanf( FILE * stream, const char *format, ... );
int scanf( const char *format, ... );
int sscanf( const char * buffer, const char *format, ... );
s[r] = 0; // Rajoute '\0' terminal
write (1, s, strlen(s));
read  (0, s, sizeof(s)-1);
```

```
int ouvrir_tubes_service(Client *c)
{
   printf("ouverture tube_sc\n");
   c->tube_sc = open(c->tube_sc_nom, O_RDONLY);
   if (c->tube_sc < 0) {
      perror("open tube_sc");
      return -1;
   }
   printf("ouverture tube_cs\n");
   c->tube_cs = open(c->tube_cs_nom, O_WRONLY);
   if (c->tube_cs < 0) {
      perror("open tube_cs");
      return -1;
   }
   return 0;
}
```

CLIENT UDP

```
int main(int argc, const char *argv[])
{
   if (argc < 3) {
      fprintf(stderr, "2 arguments requis: addresse du client et du serveur\n");
      exit(1);
   }
   const char* path = argv[1];
   const char* path_serv = argv[2];
   struct sockaddr_un addr;
   struct sockaddr_un addr_serv;
   bor_set_sockaddr_un(path_serv, &addr_serv);
   int soc = bor_create_socket_un(SOCK_DGRAM, path, &addr);
   int co = bor_connect_un(soc, &addr_serv);
   if (co < 0) {
      goto fin1;
   }
   char buf[1000];
   sprintf(buf, "HELLO !!!");
   int sd = bor_sendto_un_str(soc, buf, &addr_serv);
   if (sd < 0) {
      goto fin1;
   }
   int rc = bor_recvfrom_un_str(soc, buf, sizeof(buf), &addr_serv);
   if (rc < 0) {
      goto fin1;
   }
   printf("%s\n", buf);
fin1:
   close(soc);
   unlink(addr.sun_path);
   return 0;
}
```

SERVEUR UDP

```
int dialoguer_avec_un_client(int soc)
{
   struct sockaddr_un adr_client;
   printf("Attente d'un message client\n");
   char buf[100];
```



```
   //reset la liste
   FD_ZERO (&fd_set1);
   //ajout tube
   FD_SET (pipefd[0], &fd_set1);
   //ajout entrée standard
   FD_SET (0, &fd_set1);
   int s = select (pipefd[0] + 1, &fd_set1, NULL, NULL, NULL);
   if (s < 0) {
      if (s == EINTR)
         printf("Signal reçu\n");
      else {
         perror ("select");
         exit (1);
      }
   }
   else if (s == 0)
      printf("Time Out\n");
   if (FD_ISSET (0, &fd_set1)) {
      char buf[1000];
      int r = read (0, buf, 1000);
      if (r < 0) {
         perror ("read");
         exit (1);
      }
      if (r == 0) exit (0);
      printf("Lu %d carac sur fd = %d :\"%s\"\n", r, 0, buf);
   }
```

COMPARER : strcmp(buf, "NUMBER") == 0

```
const char* path = argv[1];
   struct sockaddr_un sa;
   int soc = bor_create_socket_un(SOCK_DGRAM, path, &sa);
   if (soc < 0) {
      exit(1);
   }
   bor_signal(SIGINT, capter_fin, 0);
   int k;
   cptMsgRecu = 0;
   while (boucle_princ) {
      k = dialoguer_avec_un_client(soc);
      if (k < 0) {
         break;
      }
      else
         ++cptMsgRecu;
   }
   printf("Fin du serveur\n");
   close(soc);
   unlink(sa.sun_path);
   exit (k < 0 ? 1 : 0);
```

```
    int k = bor_recvfrom_un_str(soc, buf, sizeof(buf), &adr_client);
    if (k < 0) {
        return -1;
    }
    printf("Reçu %d octets de %s: \"%s\"\n", k, adr_client.sun_path, buf);
    //réponse
    char rep[1000];
    time_t t;
    time(&t);
    sprintf(rep, "DATE : %s", ctime(&t));
    printf("Envoi de la réponse...\n");
    k = bor_sendto_un_str(soc, rep, &adr_client);
    return k;
}
int boucle_princ = 1;
void capter_fin(int sig)
{
    boucle_princ = 0;
    printf("capte_fin %d\n", sig);
}
int main(int argc, const char *argv[])
{
    if (argc < 2)
        fprintf(stderr, "One argument require\n");
    const char* path = argv[1];
    struct sockaddr_un sa;
    int soc = bor_create_socket_un(SOCK_DGRAM, path, &sa);
    if (soc < 0)
        exit(1);
    bor_signal(SIGINT, capter_fin, 0);
    int k;
    while (boucle_princ) {
        k = dialoguer_avec_un_client(soc);
        if (k < 0)
            break;
    }
    printf("Fin du serveur\n");
    close(soc);
    unlink(sa.sun_path);
```



```
    exit (k < 0 ? 1 : 0);
    return 0;
}

bor_signal(SIGPIPE, SIG_IGN, SA_RESTART);
CLIENT TCP :
 const char* path = argv[1];
    const char* path_serv = argv[2];
    struct sockaddr_un addr;
    struct sockaddr_un addr_serv;
    int soc = bor_create_socket_un(SOCK_STREAM, path, &addr);
    if (soc < 0) {
        exit(1);
    }
    bor_set_sockaddr_un(path_serv, &addr_serv);
    printf("Connexion au serveur...\n");
    int co = bor_connect_un(soc, &addr_serv);
    if (co < 0) {
        goto fin1;
    }
    bor_signal(SIGPIPE, SIG_IGN, SA_RESTART);
    bor_signal(SIGINT, capter_fin, 0);
/-------/
typedef struct {
    char *tube_ec_nom, tube_cs_nom[100], tube_sc_nom[100];
    int tube_ec, tube_sc, tube_cs;
} Client;

int creer_tubes_service(Client *c){
    int k;
    printf("crétion des tubes services\n");
    sprintf(c->tube_sc_nom, "tub-sc-%d.tmp", (int) getpid());
    k = mkfifo(c->tube_sc_nom, 0600);
    if (k < 0) {
        perror("mkfifo tube_sc");
        unlink(c->tube_cs_nom);
        return k;
    }
    sprintf(c->tube_cs_nom, "tub-sc-%d.tmp", (int) getpid());
    k = mkfifo(c->tube_cs_nom, 0600);
    if (k < 0) {
        perror("mkfifo tube_cs");
        unlink(c->tube_cs_nom);
        return k;
    }
    return 0;
}
```