Programmation Système et Réseau sous Unix

M. Billaud <billaud@info.iut.u-bordeaux1.fr> 15 juin 2004

Ce document est un support de cours pour les enseignements de Système et de Réseau. Il présente quelques appels système Unix nécessaires à la réalisation d'applications communicantes. Une première partie rappelle les notions de base indispensables à la programmation en C : `printf`, `scanf`, `exit`, communication avec l'environnement, allocation dynamique, gestion des erreurs. Ensuite on présente de façon plus détaillées les entrées-sorties générales d'UNIX : fichiers, tuyaux, répertoires etc., ainsi que la communication inter-processus par le mécanisme des sockets locaux par flots et datagrammes. Viennent ensuite les processus et les signaux. Les mécanismes associés aux *threads Posix* sont détaillés : sémaphores, verrous, conditions. Une autre partie décrit les IPC, que l'on trouve plus couramment sur les divers UNIX : segments partagés sémaphores et files de messages. La dernière partie aborde la communication réseau par l'interface des *sockets*, et montre des exemples d'applications client-serveur avec TCP et UDP.

Table des matières

1 Avant-Propos

1.1 Objectifs

Ce document présente quelques appels système utiles à la réalisation d'application communicantes sous UNIX.

Pour écrire de telles applications il faut savoir faire communiquer de processus entre eux, que ce soit sur la même machine ou sur des machines reliées en réseau (Internet par exemple).

Pour cela, on passe par des *appels systèmes* pour demander au système d'exploitation d'effecter des actions : ouvrir des voies de communication, expédier des données, créer des processus etc. On parle de *programmation système* lorsqu'on utilise explicitement ces appels sans passer par des bibliothèques ou des modules de haut niveau qui les encapsulent pour en cacher la complexité (supposée).[1] .

Les principaux appels systèmes sont présentés ici, avec des exemples d'utilisation.[2]

1.2 Copyright

[1] Sur le marché, on trouve par exemple des composants « serveur-TCP » tout faits pour tel ou tel langage de programmation.

[2] Attention, ce sont des illustrations des appels, pas des recommandations sur la bonne manière de les employer. En particulier, les contrôles de sécurité sur les données sont très sommaires.

– cette note de copyright et de permission doit être préservée dans toutes copies partielles ou totales
– toutes traductions ou travaux dérivés doivent être approuvés par l'auteur en le prévenant avant leur distribution
– si vous distribuez une partie de ce travail, des instructions pour obtenir la version complète doivent également être fournies
– de courts extraits peuvent être reproduits sans ces notes de permissions.

L'auteur décline tout responsabilité vis-à-vis des dommages résultant de l'utilisation des informations et des programmes qui figurent dans ce document,

1.3 Obtenir la dernière version

La dernière version de ce document peut être obtenue depuis la page Web `http ://www.labri.fr/~billaud`

2 Rappels divers

2.1 printf()/scanf()

```
int printf(const char *format, ...);
int scanf(const char *format, ...);
```

Ces instructions font des écritures et des lectures *formattées* sur les flots de sortie et d'entrée standard. Les spécifications de format sont décrites dans la page de manuel `printf(3)`.

2.2 sprintf()/sscanf()

```
int sprintf(char *str, const char *format, ...);
int sscanf(const char *str, const char *format, ...);
```

Similaires aux précédentes, mais les opérations lisent ou écrivent dans le tampon `str`.

Remarque : la fonction `sprintf()` ne connait pas la taille du tampon `str` ; il y a donc un risque de débordement. Il faut prévoir des tampons assez larges, ou (mieux) utiliser la fonction

```
#include <stdio.h>
int snprintf(char *str, size_t size, const char *format, ...);
```

de la bibliothèque GNU, qui permet d'indiquer une taille à ne pas dépasser.

2.3 system()

```
#include <stdlib.h>
int system (const char * string);
```

permet de lancer une ligne de commande (*shell*) depuis un programme. L'entier retourné par `system()` est le *code de retour* fourni en paramètre à `exit()` par la commande.

Exemple :

```
1  /*
2   * Divers/envoifichier.c
3   *
4   * Illustration de system();
5   */
```

```
 6  #include <stdio.h>
 7  #include <stdlib.h>

 8  #define TAILLE_MAX_COMMANDE      100
 9  #define TAILLE_MAX_ADRESSE       100
10  #define TAILLE_MAX_NOMFICHIER    100

11  int main(void)
12  {
13      char destinataire[TAILLE_MAX_ADRESSE];
14      char nom_fichier[TAILLE_MAX_NOMFICHIER];
15      char commande[TAILLE_MAX_COMMANDE];
16      int reponse;

17      printf("e-mail destinataire ?\n");
18      scanf("%s", destinataire);  /* dangereux */
19      printf("nom du fichier à envoyer ?\n");
20      scanf("%s", nom_fichier);   /* dangereux */

21      snprintf(commande, TAILLE_MAX_COMMANDE,
22               "uuencode %s %s | mail %s",
23               nom_fichier, nom_fichier, destinataire);

24      reponse = system(commande);
25      if (reponse == EXIT_SUCCESS) {
26          printf("OK\n");
27      } else {
28          printf("La commande retourne : %d\n", reponse);
29      }
30      return EXIT_SUCCESS;
31  }
```

3 Communication avec l'environnement

3.1 Paramètres de `main(...)`

Le lancement d'un programme C provoque l'appel de sa fonction principale `main(...)` qui a 2 paramètres optionnels :

`argc` : nombre de paramètres sur la ligne de commande (y compris le nom de l'exécutable lui-même) ;

`argv` : tableau de chaînes contenant les paramètres de la ligne de commande.

Remarque : les noms `argc`, `argv` sont purement conventionnels.

Exemple :

```
 1  /*
 2   * Divers§env.c
 3   */

 4  #include <stdio.h>
 5  #include <stdlib.h>

 6  int main(int argc, char *argv[])
 7  {
 8      int k;
```

```
 9          printf("Appel avec %d paramètres\n", argc);

10          for (k = 0; k < argc; k++)
11              printf("  %d: %s\n", k, argv[k]);

12          return EXIT_SUCCESS;
13      }
```

3.2 `getopt()` : analyse de paramètres de la ligne de commande

L'analyse des options d'une ligne de commande est facilité par l'emploi de `getopt()`.

```
#include <unistd.h>

int getopt(int argc, char *const argv[], const char *optstring);

extern char *optarg;
extern int optind, opterr, optopt;
```

`getopt()` analyse le tableau de paramètres `argv` à `argc` éléments, en se basant sur la *chaîne de spécification d'options* `optstring`. Par exemple la chaîne `"vhx :y :"` déclare 4 options, les deux dernières étant suivies d'un paramètre.

A chaque étape, `optarg` retourne le nom de l'option (ou un point d'interrogation pour une option non reconnue), et fournit éventuellement dans `optarg` la valeur du paramètre associé.

A la fin `getopt` retourne `-1`, et le tableau `argv` a été réarrangé pour que les paramètres supplémentaires soient stockés à partir de l'indice `optind`.

```
 1      /* Divers/essai-getopt.c */

 2      #include <stdio.h>
 3      #include <stdlib.h>
 4      #include <unistd.h>

 5      int main(int argc, char *argv[])
 6      {
 7          char *opt_a = NULL;
 8          int opt_x = 0;
 9          int index;
10          char c;

11          while ((c = getopt(argc, argv, "hxa:")) != -1) {
12              switch (c) {
13              case 'h':
14                  printf("Help:  %s [options...] parametres ... \n\n",
15                         argv[0]);
16                  printf("Options:\n" "-h\tCe message d'aide\n"
17                         "-x\toption x\n" "-a nom\t paramètre optionnel\n");
18                  return EXIT_SUCCESS;
19              case 'x':
20                  opt_x = 1;
21                  break;
22              case 'a':
23                  opt_a = optarg;
24                  break;
```

```
25              case '?':
26                  fprintf(stderr, "Option inconnue `-%c'.\n", optopt);
27                  return EXIT_FAILURE;
28              default:
29                  return EXIT_FAILURE;
30              }
31          }
32          printf("= option `-x' %s\n", (opt_x ? "activée" : "désactivée"));

33          if (opt_a == NULL)
34              printf("= paramètre `-a' absent\n");
35          else
36              printf("= paramètre `-a' présent = %s\n", opt_a);

37          printf("%d paramètres supplémentaires\n", argc - optind);
38          for (index = optind; index < argc; index++)
39              printf("    ->  %s\n", argv[index]);

40          return EXIT_SUCCESS;
41      }
```

Exemple.

```
% essai-getopt -a un deux trois -x quatre
= option `-x' activée
= paramètre `-a' présent = un
3 paramètres supplémentaires
    ->  deux
    ->  trois
    ->  quatre
```

3.3 Variables d'environnement

La fonction `getenv()` permet de consulter les variables d'environnement :

```
#include <stdlib.h>
char *getenv(const char *name);
```

Exemple :

```
1   /* Divers/getterm.c */

2   #include <stdlib.h>
3   #include <stdio.h>

4   int main(void)
5   {
6       char *terminal = getenv("TERM");
7       printf("Le terminal est ");
8       if (terminal == NULL)
9           printf("inconnu\n");
10      else
11          printf("un %s\n", terminal);
12      return EXIT_SUCCESS;
13  }
```

Exercice : Ecrire un programme `exoenv.c` qui affiche les valeurs des variables d'environnement indiquées. Exemple d'exécution :

```
% exoenv TERM LOGNAME PWD
TERM=xterm
LOGNAME=billaud
PWD=/net/profs/billaud/essais
%
```

Voir aussi les fonctions

```
int putenv(const char *string);
int setenv(const char *name, const char *value, int overwrite);
void unsetenv(const char *name);
```

qui permettent de modifier les variables d'environnement du processus courant et de ses fils. Il n'y a pas de moyen de modifier les variables du processus père.

Exercice. Écrire un petit programme pour mettre en évidence le fait que le troisième paramètre de `main(...)` indique les valeurs des variables d'environnement *au moment de l'appel*, et non pas les valeurs courantes.

3.4 `exit()` : Fin de programme

L'exécution d'un programme se termine
– soit par un `return` dans la fonction `main()`
– soit par un appel à la fonction `exit()`

```
#include <stdlib.h>
void exit(int status);
```

Le paramètre `status` est le *code de retour* du processus. On utilisera de préférence les deux constantes `EXIT_SUCCESS` et `EXIT_FAILURE` définies dans `stdlib.h`.

4 Erreurs

4.1 `errno`, `perror()`

La plupart des fonctions du système peuvent échouer pour diverses raisons. On peut alors examiner la variable `errno` pour déterminer plus précisément la cause de l'échec, et agir en conséquence.

```
#include <errno.h>
extern int errno;
```

La fonction `perror()` imprime sur la sortie d'erreur standard un message décrivant la dernière erreur qui s'est produite, précédé par la chaîne `s`.

```
#include <stdio.h>
void perror(const char *s);
```

Enfin, la fonction `strerror()` retourne le texte (en anglais) du message d'erreur correspondant à un numéro.

```
#include <string.h>
char *strerror(int errnum);
```

Exemple : programme qui change les droits d'accès à des fichiers grâce à la commande système *chmod(2)*.

```c
/* Divers/droits.c */

/*
 * met les droits 0600 sur un ou plusieurs fichiers
 * (illustration de chmod() et errno)
*/

#include <stdlib.h>
#include <stdio.h>
#include <sys/stat.h>
#include <errno.h>
#include <string.h>

void ecrire_message(int numero)
{
    switch (numero) {
    case EACCES:
        printf("impossible de consulter un des "
               "répertoires du chemin");
        break;
    case ELOOP:
        printf("trop de liens symboliques (boucles ?)");
        break;
    case ENAMETOOLONG:
        printf("le nom est trop long");
        break;
    case ENOENT:
        printf("le fichier n'existe pas");
        break;
    case EPERM:
        printf("permission refusée");
        break;
    default:
        printf("erreur %s", strerror(numero));
        break;
    }
}

int main(int argc, char *argv[])
{
    int k;
    for (k = 1; k < argc; k++) {
        printf("%s: ", argv[k]);
        if (chmod(argv[k], S_IREAD | S_IWRITE) == 0) {
            printf("fichier protégé");
        } else {
            ecrire_message(errno);
        }
        printf("\n");
    }
    return EXIT_SUCCESS;
}
```

4.2 Traitement des erreurs, branchements non locaux

```
#include <setjmp.h>

int setjmp(jmp_buf env);
void longjmp(jmp_buf env, int val);
```

Ces deux fonctions permettent de réaliser un *branchement* d'une fonction à une autre (la première doit avoir été appelée, au moins indirectement, par la seconde). C'est un moyen primitif de réaliser un traitement d'erreurs par *exceptions*.

La fonction `setjmp()` sauve l'environnement (contexte d'exécution) dans la variable tampon `env`, et retourne 0 si elle a été appelée directement.

La fonction `longjmp()` rétablit le dernier environnement qui a été sauvé dans `env` par un appel à `setjmp()`. Le programme continue à l'endroit du `setjmp()` comme si celui-ci avait retourné la valeur `val`. (Si on met `val` à 0 la valeur retournée est 1).

Exemple.

```
 1  /* Divers/jump.c */

 2  #include <setjmp.h>
 3  #include <stdio.h>
 4  #include <stdlib.h>

 5  #define EXCEPTION_PAS_DE_SOLUTION   1
 6  #define EXCEPTION_EQUATION_TRIVIALE 2

 7  jmp_buf erreur;

 8  float resolution(int a, int b)
 9  {
10      if (a == 0) {
11          if (b == 0)
12              longjmp(erreur, EXCEPTION_EQUATION_TRIVIALE);
13          else
14              longjmp(erreur, EXCEPTION_PAS_DE_SOLUTION);
15      };
16      return (-b / (float) a);
17  }

18  int main(void)
19  {
20      int a, b;
21      float x;
22      int ret;

23      printf("Coefficients de ax+b=0\n");
24      scanf("%d %d", &a, &b);

25      ret = setjmp(erreur);

26      if (ret == 0) {
27          x = resolution(a, b);
28          printf("Solution = %f\n", x);
29          return EXIT_SUCCESS;
```

```
30          }

31          switch (ret) {
32          case EXCEPTION_PAS_DE_SOLUTION:
33              printf("Cette équation n'a pas de solution\n");
34              break;
35          case EXCEPTION_EQUATION_TRIVIALE:
36              printf("Cette équation est toujours vraie.\n");
37              break;
38          }
39          return EXIT_FAILURE;
40      }
```

5 Allocation dynamique

```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
void *realloc(void *ptr, size_t size);
```

`malloc()` demande au système d'exploitation l'allocation d'un espace mémoire de taille supérieure ou égale à `size` octets. La valeur retournée est un pointeur sur cet espace (`NULL` en cas d'échec).

`free()` restitue cet espace au système. `realloc()` permet d'agrandir la zone allouée.

Il est très fréquent de devoir allouer une zone mémoire pour y loger une copie d'une chaîne de caractères. On utilise pour cela la fonction `strdup()`

```
#include <string.h>
char *strdup(const char *s);
```

L'instruction "`nouveau = strdup(ancien)`" est approximativement équivalente à :

```
nouveau = strcpy(malloc(1+strlen(ancien)), ancien);
```

Exemple : la fonction `lireligne()` ci-dessous lit une ligne de l'entrée standard et retourne cette ligne dans un tampon d'une taille suffisante. Elle renvoie le pointeur `NULL` si il n'y a plus de place en mémoire.

```
 1  /* lireligne.c */

 2  #include <stdlib.h>
 3  #include <stdio.h>

 4  #define TAILLE_INITIALE 16

 5  char *lire_ligne(void)
 6  {
 7      /*
 8         Renvoie un tampon de texte contenant une ligne
 9         lue sur l'entrée standard.
10         Un nouveau tampon est créé à chaque invocation.
11         Renvoie NULL en cas de problème d'allocation
12       */

13      char *chaine;
14      int taille_allouee, taille_courante;
15      int c;
```

```
16          chaine = malloc(TAILLE_INITIALE);
17          if (chaine == NULL)
18              return NULL;
19          taille_courante = 0;
20          taille_allouee = TAILLE_INITIALE;

21          while (1) {
22              c = getchar();
23              if ((c == '\n') || (c == EOF))
24                  break;
25              chaine[taille_courante++] = c;
26              if (taille_courante == taille_allouee) {
27                  char *tmp;
28                  taille_allouee *= 2;
29                  tmp = realloc(chaine, taille_allouee);
30                  if (tmp == NULL) {
31                      free(chaine);
32                      return NULL;
33                  }
34                  chaine = tmp;
35              }
36          };
37          chaine[taille_courante] = '\0';
38          return chaine;
39      }

40      int main(void)
41      {
42          char *chaine;
43          printf("tapez une grande chaîne\n");
44          chaine = lire_ligne();
45          if (chaine == NULL) {
46              fprintf(stderr, "Plus de place en mémoire\n");
47              return EXIT_FAILURE;
48          }
49          printf("%s\n", chaine);
50          free(chaine);
51          return EXIT_SUCCESS;
52      }
```

Attention : dans l'exemple ci-dessus la fonction `lirechaine()` alloue un nouveau tampon à chaque invocation. Il est donc de la responsabilité du programmeur de libérer ce tampon après usage pour éviter les *fuites mémoire*.

6 Manipulation des fichiers, opérations de haut niveau

Ici on utilise le terme « fichier » dans son sens Unixien le plus large, ce qui inclue les périphériques (comme `/dev/tty`, `/dev/audio`,...) aussi bien que les « pseudo-fichiers » comme `/proc/cpuinfo`, `/proc/sound`, etc.

6.1 Flots standards

Il y a trois flots prédéclarés, qui correspondent à l'entrée et la sortie standards, et à la sortie d'erreur :

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

```
FILE *stdin;
FILE *stdout;
FILE *stderr;
```

Quelques fonctions agissent implicitement sur ces flots :

```
int  printf(const char *format, ...);
int  scanf(const char *format, ...);
int  getchar(void);
char *gets(char *s);
```

`printf(...)` écrit sur `stdout` la valeur d'expressions selon un format donné. `scanf(...)` lit sur `stdin` la valeur de variables.

```
 1  /* facture.c */

 2  /* exemple printf(), scanf() */

 3  #include <stdlib.h>
 4  #include <stdio.h>

 5  int main(void)
 6  {
 7      char article[10];
 8      float prix;
 9      int quantite;
10      int n;

11      printf("article, prix unitaire, quantité ?\n");
12      n = scanf("%s %f %d", article, &prix, &quantite);

13      if (n != 3) {               /* on n'a pas réussi à lire 3 choses ? */
14          printf("Erreur dans les données");
15          return EXIT_FAILURE;
16      }

17      printf("%d %s à %10.2f =  %10.2f\n",
18             quantite, article, prix, prix * quantite);

19      return EXIT_SUCCESS;
20  }
```

`scanf()` renvoie le nombre d'objets qui ont pu être effectivement lus sans erreur.

`gets()` lit des caractères jusqu'à une marque de fin de ligne ou de fin de fichier, et les place (marque non comprise) dans le tampon donné en paramètre, suivis par un caractère NUL. `gets()` renvoie finalement l'adresse du tampon. Attention, prévoir un tampon assez grand, ou (beaucoup mieux) utilisez `fgets()`.

`getchar()` lit un caractère sur l'entrée standard et retourne sa valeur sous forme d'entier positif, ou la constante `EOF` (-1) en fin de fichier.

6.2 Opérations sur les flots

```
#include <stdio.h>
FILE *fopen(char *path, char *mode);
int fclose(FILE *stream);
int fprintf(FILE *stream, const char *format, ...);
```

```
int fscanf(FILE *stream, const char *format, ...);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

`fopen()` tente d'ouvrir le fichier désigné par la chaîne `path` selon le mode indiqué, qui peut être `"r"` (lecture seulement), `"r+"` (lecture et écriture), `"w"` (écriture seulement), `"w+"` (lecture et écriture, effacement si le fichier existe déjà), `"a"` (écriture à partir de la fin du fichier si il existe déjà), `"a+"` (lecture et écriture, positionnement à la fin du fichier si il existe déjà).

Si l'ouverture échoue, `fopen()` retourne le pointeur `NULL`.

```
int fprintf(FILE *stream, const char *format, ...);
int fscanf(FILE *stream, const char *format, ...);
int fgetc(FILE *stream);
char *fgets(char *s, int size, FILE *stream);
```

Les trois premières fonctions ne diffèrent de `printf()`, `scanf()` et `getchar()` que par le premier paramètre, qui précise sur quel flot porte l'opération.

Comme `gets()`, `fgets()` lit une ligne de caractères pour la placer dans un tampon, mais

1. la longueur de ce tampon est précisée (ceci empêche les débordements)
2. le caractère de fin de ligne (*newline*) est effectivement stocké dans le tampon.

Exemple : comptage du nombre de caractères et de lignes d'un fichier. Affichage du résultat sur la sortie standard.

```
 1  /* Diver/comptage.c */

 2  #include <stdio.h>
 3  #include <stdlib.h>

 4  void abandon(char raison[])
 5  {
 6      perror(raison);
 7      exit(EXIT_FAILURE);
 8  }

 9  void compter(char nom_fichier[], FILE * f)
10  {
11      int c;
12      int nb_caracteres = 0, nb_lignes = 0;

13      while ((c = fgetc(f)) != EOF) {
14          nb_caracteres++;
15          if (c == '\n')
16              nb_lignes++;
17      };
18      printf("%s: %d cars, %d lignes\n",
19             nom_fichier, nb_caracteres, nb_lignes);
20  }

21  int main(int argc, char *argv[])
22  {
23      char *nom_fichier;
24      char *nom_programme = argv[0];
25      FILE *f;
```

```
26          switch (argc) {
27          case 1:
28              compter("entrée standard", stdin);
29              break;
30          case 2:
31              nom_fichier = argv[1];
32              f = fopen(nom_fichier, "r");
33              if (f == NULL) {
34                  fprintf(stderr,
35                          "%s: fichier %s absent ou illisible\n",
36                          nom_programme, nom_fichier);
37                  exit(EXIT_FAILURE);
38              };
39              compter(nom_fichier, f);
40              fclose(f);
41              break;
42          default:
43              fprintf(stderr, "Usage: %s [fichier]\n", nom_programme);
44              exit(EXIT_SUCCESS);
45          };
46          return (EXIT_SUCCESS);
47      }
```

6.3 Positionnement

```
int feof(FILE *stream);
long ftell(FILE *stream);
int fseek(FILE *stream, long offset, int whence);
```

`feof()` indique si la fin de fichier est atteinte. `ftell()` indique la *position courante* dans le fichier (0 = début). `fseek()` déplace la position courante : si `whence` est `SEEK_SET` la position est donnée par rapport au début du fichier, si c'est `SEEK_CUR` : déplacement par rapport à la position courante, `SEEK_END` : déplacement par rapport à la fin.

6.4 Divers

```
int ferror(FILE *stream);
void clearerr(FILE *stream);
int fileno(FILE *stream);
FILE *fdopen(int fildes, char *mode);
```

La fonction`ferror()` indique si une erreur a eu lieu sur un flot, `clearerr()` efface l'indicateur d'erreur et `fileno()` renvoie le numéro de fichier de bas niveau (*descripteur de fichier*) correspondant à un flot. Inversement, `fdopen()` ouvre un fichier de haut niveau à partir d'un fichier de bas niveau déjà ouvert.

7 Manipulation des fichiers, opérations de bas niveau

7.1 Ouverture, fermeture, lecture, écriture

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags, mode_t mode);
```

Ouverture d'un fichier nommé `pathname`. Les `flags` peuvent prendre l'une des valeurs suivantes : `O_RDONLY` (lecture seulement), `O_WRONLY` (écriture seulement), `O_RDWR` (lecture et écriture). Cette valeur peut être combinée éventuellement (par un "ou logique") avec des options : `O_CREAT` (création du fichier si il n'existe pas déjà), `O_TRUNC` (si le fichier existe il sera tronqué), `O_APPEND` (chaque écriture se fera à la fin du fichier), etc.

En cas de création d'un nouveau fichier, le `mode` sert à préciser les droits d'accès. Lorsqu'un nouveau fichier est créé, `mode` est combiné avec le `umask` du processus pour former les droits d'accès du fichier. Les permissions effectives sont alors `(mode & ~umask)`.

Le paramètre `mode` doit être présent quand les `flags` contiennent `O_CREAT`.

L'entier retourné par `open()` est un *descripteur de fichier* (-1 en cas d'erreur), qui sert à référencer le fichier par la suite.

Les descripteurs 0, 1 et 2 correspondent aux fichiers standards `stdin`, `stdout` et `stderr` qui sont normalement déjà ouverts (0=entrée, 1=sortie, 2=erreurs).

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

int close(int fd);
int read(int fd, char *buf, size_t count);
size_t write(int fd, const char *buf, size_t count);
```

`close()` ferme le fichier indiqué par le descripteur `fd`. Retourne 0 en cas de succès, -1 en cas d'échec. `read()` demande à lire *au plus* `count` octets sur `fd`, à placer dans le tampon `buf`. Retourne le nombre d'octets qui ont été effectivement lus, qui peut être inférieur à la limite donnée pour cause de non-disponibilité (-1 en cas d'erreur, 0 en fin de fichier). `write()` tente d'écrire sur le fichier les `count` premiers octets du tampon `buf`. Retourne le nombre d'octets qui ont été effectivement écrits, -1 en cas d'erreur.

Exemple :

```
 1  /* copie.c */

 2  /*
 3     Entrées-sorties de bas niveau
 4     Usages:
 5     1: copie              (entrée-standard -> sortie standard)
 6     2: copie fichier      (fichier -> sortie standard)
 7     3: copie source dest  (source -> dest)
 8  */

 9  #include <stdlib.h>
10  #include <stdio.h>
11  #include <sys/types.h>
12  #include <sys/stat.h>
13  #include <fcntl.h>
14  #include <unistd.h>

15  #define ENTREE_STANDARD 0
16  #define SORTIE_STANDARD 1
17  #define TAILLE_TAMPON   4096

18  void abandon(char message[])
19  {
20      fprintf(stderr, "Erreur fatale %s\n", message);
```

```
21          exit(EXIT_FAILURE);
22      }

23      void transfert(int entree, int sortie)
24      {
25          int nb_lus, nb_ecrits;
26          char tampon[TAILLE_TAMPON];

27          if (entree == -1) {
28              abandon("Fichier d'entrée introuvable");
29          }
30          if (sortie == -1) {
31              abandon("Ne peut pas ouvrir le fichier de sortie");
32          }

33          for (;;) {
34              nb_lus = read(entree, tampon, TAILLE_TAMPON);
35              if (nb_lus <= 0)
36                  break;
37              nb_ecrits = write(sortie, tampon, nb_lus);
38              if (nb_ecrits != nb_lus)
39                  abandon("Probleme d'ecriture");
40          }
41      }

42      int main(int argc, char *argv[])
43      {
44          int entree, sortie;

45          switch (argc) {
46          case 1:
47              transfert(ENTREE_STANDARD, SORTIE_STANDARD);
48              break;
49          case 2:
50              entree = open(argv[1], O_RDONLY);
51              transfert(entree, SORTIE_STANDARD);
52              close(entree);
53              break;
54          case 3:
55              entree = open(argv[1], O_RDONLY);
56              sortie = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0666);
57              transfert(entree, sortie);
58              close(entree);
59              close(sortie);
60              break;
61          default:
62              abandon("Usage: copie [src [dest]]");
63          }
64          return EXIT_SUCCESS;
65      }
```

Problème. Montrez que la taille du tampon influe sur les performances des opérations d'entrée-sortie. Pour cela, modifiez le programme précédent pour qu'il accepte 3 paramètres : les noms des fichiers source et destination, et la taille du tampon (ce tampon sera alloué dynamiquement).

7.2 Suppression

```
#include <stdio.h>
int remove(const char *pathname);
```

Cette fonction supprime le fichier `pathname`, et retourne 0 en cas de succès (-1 sinon).

Exercice : écrire un substitut pour la commande *rm*.

7.3 Positionnement

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

off_t lseek(int fildes, off_t offset, int whence);
```

`lseek()` repositionne le pointeur de lecture. Similaire à `fseek()`. Pour connaître la position courante faire un appel à `stat()`.

Exercice. Écrire un programme pour manipuler un fichier relatif d'enregistrements de taille fixe.

7.4 Verrouillage

```
#include <sys/file.h>
int flock(int fd, int operation)
```

Lorsque `operation` est `LOCK_EX`, il y a verrouillage du fichier désigné par le descripteur `fd`. Le fichier est déverrouillé par l'option `LOCK_UN`.

Problème. Ecrire une fonction `mutex()` qui permettra de délimiter une section critique dans un programme C. Exemple d'utilisation :

```
#include "mutex.h"
...
mutex("/tmp/foobar",MUTEX_BEGIN);
...
mutex("/tmp/foobar",MUTEX_END);
```

Le premier paramètre indique le nom du fichier utilisé comme verrou. Le second précise si il s'agit de verrouiller ou déverrouiller. Faut-il prévoir des options `MUTEX_CREATE`, `MUTEX_DELETE` ? Qu'arrive-t'il si un programme se termine en "oubliant" de fermer des sections critiques ?

Fournir le fichier d'interface `mutex.h`, l'implémentation `mutex.c`, et des programmes illustrant l'utilisation de cette fonction.

7.5 Informations sur les fichiers/répertoires/...

```
#include <sys/stat.h>
#include <unistd.h>
int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

Ces fonctions permettent de connaître diverses informations sur un fichier désigné par un chemin d'accès (`stat()`) ou par un descripteur (`fstat()`).

Exemple :

```
 1  /* Divers/taille.c */
 2  /*
 3          indique la taille et la nature
 4          des "fichiers" cités en paramètres
 5  */

 6  #include <stdlib.h>
 7  #include <stdio.h>
 8  #include <sys/stat.h>
 9  #include <errno.h>
10  #include <string.h>

11  char *nom_mode(mode_t mode)
12  {
13      if (S_ISREG(mode))
14          return "fichier";
15      if (S_ISLNK(mode))
16          return "lien symbolique";
17      if (S_ISDIR(mode))
18          return "répertoire";
19      if (S_ISCHR(mode))
20          return "périphérique mode caractère";
21      if (S_ISBLK(mode))
22          return "péripherique mode bloc";
23      if (S_ISFIFO(mode))
24          return "fifo";
25      if (S_ISSOCK(mode))
26          return "socket";
27      return "inconnu";
28  }

29  int main(int argc, char *argv[])
30  {
31      int k;
32      struct stat s;

33      if (argc < 2) {
34          fprintf(stderr, "Usage: %s fichier ... \n", argv[0]);
35          exit(EXIT_FAILURE);
36      };
37      for (k = 1; k < argc; k++) {
38          printf("%20s:\t", argv[k]);
39          if (stat(argv[k], &s) == 0) {
40              printf("taille %8ld, type %s",
41                     s.st_size, nom_mode(s.st_mode));
42          } else {
43              switch (errno) {
44              case ENOENT:
45                  printf("le fichier n'existe pas");
46                  break;
47              default:
48                  printf("erreur %s", strerror(errno));
49                  break;
50              }
```

```
51              }
52              printf("\n");
53          }
54          return EXIT_SUCCESS;
55      }
```

7.6 Parcours de répertoires

Le parcours d'un répertoire, pour obtenir la liste des fichiers et répertoires qu'il contient, se fait grâce aux fonctions :

```
#include <sys/types.h>
#include <dirent.h>
DIR *opendir(const char *name);
int closedir(DIR *dir);
void rewinddir(DIR *dir);
void seekdir(DIR *dir, off_t offset);
off_t telldir(DIR *dir);
```

Voir la documentation pour des exemples.

Exercice : écrire une version simplifiée de la commande `ls`.

Exercice : écrire une commande qui fasse apparaître la structure d'une arborescence. Exemple d'affichage :

```
C++
| CompiSep
| Fichiers
Systeme
| Semaphores
| Pipes
| Poly
  | SVGD
  | Essais
| Fifos
```

Conseil : écrire une fonction à deux paramètres : le chemin d'accès du répertoire et le niveau de récursion.

7.7 Duplication de descripteurs

```
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

Ces deux fonctions créent une copie du descripteur `oldfd`. `dup()` utilise le plus petit numéro de descripteur libre. `dup2()` réutilise le descripteur `newfd`, en fermant éventuellement le fichier qui lui était antérieurement associé.

La valeur retournée est celle du descripteur, ou -1 en cas d'erreur.

Exemple

```
1   /* Divers/redirection.c */

2   /*
3     Le fichier cité en paramètre est passé à travers
4     la commande wc.
5   */
```

```
 6  #include <sys/types.h>
 7  #include <fcntl.h>
 8  #include <stdio.h>
 9  #include <stdlib.h>
10  #include <unistd.h>
11  #include <errno.h>

12  #define COMMANDE "wc"

13  void abandon(char message[])
14  {
15      perror(message);
16      exit(EXIT_FAILURE);
17  }

18  int main(int argc, char *argv[])
19  {
20      int fd;

21      if (argc != 2) {
22          fprintf(stderr, "Usage: %s fichier\n", argv[0]);
23          abandon("mauvais nombre de paramètres");
24      }

25      fd = open(argv[1], O_RDONLY);
26      if (fd == -1)
27          abandon("open");

28      /* transfert du descripteur dans celui de l'entrée standard */
29      if (dup2(fd, 0) < 0)
30          abandon("dup2");

31      close(fd);

32      system(COMMANDE);
33      if (errno != 0)
34          abandon("system");

35      return EXIT_SUCCESS;
36  }
```

Exercice : que se produit-il si on essaie de rediriger la *sortie* standard d'une commande à la manière de l'exemple précédent ? (essayer avec "`ls`", "`ls -l`").

7.8 `mmap()` : fichiers "mappés" en mémoire

Un fichier "mappé en mémoire" apparaît comme un tableau d'octets, ce qui permet de le parcourir en tous sens plus commodément qu'avec des `seek()`, `read()` et `write()`.

C'est technique beaucoup plus économique que de copier le fichier dans une zone allouée en mémoire : c'est le système de mémoire virtuelle qui s'occupe de lire et écrire physiquement les pages du fichier au moment où on tente d'y accéder, et gère tous les tampons.

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

```
void  *  mmap(void  *start,  size_t length,
              int prot , int flags,
              int fd, off_t offset);
int munmap(void *start, size_t length);
```

La fonction `mmap()` "mappe" en mémoire un morceau (de longueur `length`, en partant du `offset`-ième octet) du fichier désigné par le descripteur `fd` , et retourne un pointeur sur la zone de mémoire correspondante.

On peut définir quelques options (protection en lecture seule, partage, etc) grâce à `prot` et `flags`. Voir pages de manuel. `munmap()` "libère" la mémoire.

```
 1  /*
 2    Divers/inverse.c

 3    Affichage des lignes d'un fichier en partant de la fin
 4    Exemple d'utilisation de mmap

 5    M. Billaud, Octobre 2002
 6  */

 7  #include <unistd.h>
 8  #include <stdio.h>
 9  #include <stdlib.h>
10  #include <sys/types.h>
11  #include <sys/stat.h>
12  #include <sys/mman.h>
13  #include <fcntl.h>

14  void abandon(char message[])
15  {
16      perror(message);
17      exit(EXIT_FAILURE);
18  }

19  /* affichage à l'envers du contenu d'un tampon de texte */

20  void afficher(char tampon[], int taille)
21  {
22      char *ptr;
23      int nb = 1;                /* longueur ligne courante, y compris '\n'  */

24      for (ptr = tampon + taille - 1; ptr >= tampon; ptr--) {
25          if (*ptr == '\n') {
26              write(1, ptr + 1, nb);
27              nb = 0;
28          }
29          nb++;
30      }
31      write(1, ptr + 1, nb);
32  }

33  int main(int argc, char *argv[])
34  {
35      int fd;
36      struct stat s;
```

```
37          char *t;

38          if (argc != 2) {
39              fprintf(stderr, "Usage: %s fichier\n", argv[0]);
40              return EXIT_FAILURE;
41          }
42          if (stat(argv[1], &s) != 0)
43              abandon("Contrôle du type");

44          if (!S_ISREG(s.st_mode)) {
45              fprintf(stderr, "%s n'est pas un fichier\n", argv[1]);
46              return EXIT_FAILURE;
47          }
48          fd = open(argv[1], O_RDONLY);
49          if (fd < 0)
50              abandon("Ouverture");

51          t = mmap(NULL, s.st_size, PROT_READ, MAP_PRIVATE, fd, 0);
52          if (t == MAP_FAILED)
53              abandon("Mise en mémoire");

54          afficher(t, s.st_size);

55          if (munmap(t, s.st_size) < 0)
56              abandon("Détachement");

57          close(fd);
58          return EXIT_SUCCESS;
59      }
```

8 Pipes et FIFOs

Les *pipes* et les *FIFOs* permettent de faire communiquer des processus d'une même machine : ce qu'un (ou plusieurs) processus écrit peut être lu par un autre.

8.1 FIFOs

Les FIFOs sont également appelés « tuyaux nommés ». Ce sont des objets visibles dans l'arborescence des fichiers et répertoires. Ce qu'un processus écrit dans une FIFO peut être lu immédiatement par d'autres processus.

```
#include <stdio.h>
int mkfifo (const char *path, mode_t mode);
```

La fonction `mkfifo()` crée un FIFO ayant le chemin indiqué par `path` et les droits d'accès donnés par `mode`. Si la création réussit, la fonction renvoie 0, sinon -1. Exemple :

```
if(mkfifo("/tmp/fifo.courrier",0644) == -1)
   perror("mkfifo");
```

Les FIFOS sont ensuite utilisables par `open()`, `read()`, `write()`, `close()`, `fdopen()` etc.

Exercice. Regarder ce qui se passe quand
– plusieurs processus écrivent dans une même FIFO (faire une boucle `sleep-write`).
– plusieurs processus lisent la même FIFO.

Exercice. Ecrire une commande *mutex* qui permettra de délimiter une section critique dans des shell-scripts. Exemple d'utilisation :

```
mutex -b /tmp/foobar
...
mutex -e /tmp/foobar
```

Le premier paramètre indique si il s'agit de verrouiller (`-b` = begin) ou de déverrouiller (`-e` = end). Le second paramètre est le nom du verrou.

Conseil : la première option peut s'obtenir en tentant de lire un *jeton* dans une FIFO. C'est la seconde option qui dépose le jeton. Prévoir une option pour créer une FIFO ?

8.2 Pipes

On utilise un *pipe* (tuyau) pour faire communiquer un processus et un de ses descendants[3].

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

L'appel `pipe()` fabrique un tuyau de communication et renvoie dans un tableau une paire de descripteurs. On lit à un bout du tuyau (sur le descripteur de sortie `fildes[0]`) ce qu'on a écrit dans l'autre (`filedes[1]`).

Voir exemple dans 11.1 (fork).

Les *pipes* ne sont pas visibles dans l'arborescence des fichiers et répertoires, par contre ils sont hérités lors de la création d'un processus.

La fonction `socketpair()` (voir 10.5 (socketpair)) généralise la fonction `pipe`.

8.3 Pipes depuis/vers une commande

```
#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

`popen()` lance la commande décrite par la chaîne `command` et retourne un flot.

Si `type` est `"r"` le flot retourné est celui de la sortie standard de la commande (on peut y lire). Si `type` est `"w"` c'est son entrée standard.

`pclose()` referme ce flot.

Exemple : envoi d'un « `ls -l` » par courrier

```
1   /* Divers/avis.c */

2   /* Illustration de popen() */

3   #include <sys/types.h>
4   #include <fcntl.h>
5   #include <stdlib.h>
6   #include <unistd.h>
7   #include <stdio.h>

8   #define TAILLE_MAX_COMMANDE 100

9   void abandon(char raison[])
```

[3] ou des descendants - au sens large - du processus qui ont crée le tuyau

```
10      {
11          perror(raison);
12          exit(EXIT_FAILURE);
13      }

14      int main(int argc, char *argv[])
15      {
16          char cmdmail[TAILLE_MAX_COMMANDE];
17          FILE *fmail, *fls;
18          int c;

19          if (argc != 2) {
20              fprintf(stderr, "Usage: %s destinataire\n", argv[0]);
21              exit(EXIT_FAILURE);
22          };

23          snprintf(cmdmail, TAILLE_MAX_COMMANDE, "sendmail %s", argv[1]);
24          fmail = popen(cmdmail, "w");
25          if (fmail == NULL)
26              abandon("popen(sendmail)");

27          fprintf(fmail, "Subject: ls -l\n\n");
28          fprintf(fmail, "Cher %s,\nvoici mon répertoire:\n", argv[1]);

29          fls = popen("ls -l", "r");
30          if (fls == NULL)
31              abandon("popen(ls)");

32          while ((c = fgetc(fls)) != EOF) {
33              fputc(c, fmail);
34          }
35          pclose(fls);

36          fprintf(fmail, "---\nLe Robot\n");
37          pclose(fmail);
38          return EXIT_SUCCESS;
39      }
```

9 select() : attente de données

Il est assez courant de devoir attendre des données en provenance de plusieurs sources. On utilise pour cela la fonction `select()` qui permet de surveiller plusieurs descripteurs simultanément.

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

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

FD_CLR(int fd, fd_set *set);
FD_ISSET(int fd, fd_set *set);
```

```
FD_SET(int fd, fd_set *set);
FD_ZERO(fd_set *set);
```

Cette fonction attend que des données soient prêtes à être lues sur un des descripteurs de l'ensemble `readfs`, ou que l'un des descripteurs de `writefds` soit prêt à recevoir des écritures, que des exceptions se produisent (`exceptfds`), ou encore que le temps d'attente `timeout` soit épuisé.

Lorsque `select()` se termine, `readfds`, `writefds` et `exceptfds` contiennent les descripteurs qui ont changé d'état. `select()` retourne le nombre de descripteurs qui ont changé d'état, ou -1 en cas de problème.

L'entier `n` doit être supérieur (strictement) au plus grand des descripteurs contenus dans les 3 ensembles (c'est en fait le nombre de bits significatifs du masque binaire qui représente les ensembles). On peut utiliser la constante `FD_SETSIZE`.

Les pointeurs sur les ensembles (ou le délai) peuvent être `NULL`, ils représentent alors des ensembles vides (ou une absence de limite de temps).

Les macros `FD_CLR, FD_ISSET, FD_SET, FD_ZERO` permettent de manipuler les ensembles de descripteurs.

9.1 Attente de données provenant de plusieurs sources

Exemple :

```
1   /* Divers/mix.c */

2   /*
3      affiche les données qui proviennent de 2 fifos
4      usage:  mix f1 f2
5   */

6   #include <sys/time.h>
7   #include <sys/types.h>
8   #include <sys/stat.h>
9   #include <unistd.h>
10  #include <stdio.h>
11  #include <stdlib.h>
12  #include <fcntl.h>

13  #define TAILLE_TAMPON 128

14  void abandon(char message[])
15  {
16      perror(message);
17      exit(EXIT_FAILURE);
18  }

19  /************************************************
20     Mixe les données en provenance de deux
21     descripteurs
22  ************************************************/

23  void mixer(int fd1, int fd2, int sortie)
24  {
25      fd_set ouverts;             /* les descripteurs ouverts */
26      int nb_ouverts;
```

```
27          FD_ZERO(&ouverts);
28          FD_SET(fd1, &ouverts);
29          FD_SET(fd2, &ouverts);
30          nb_ouverts = 2;

31          while (nb_ouverts > 0) { /* tant qu'il reste des
32                                      descripteurs ouverts.... */
33              fd_set prets;
34              char tampon[TAILLE_TAMPON];

35              /* on attend qu'un descripteur soit prêt ... */
36              prets = ouverts;
37              if (select(FD_SETSIZE, &prets, NULL, NULL, NULL) < 0)
38                  abandon("select");

39              if (FD_ISSET(fd1, &prets)) {
40                  /* si fd1 est prêt... */
41                  int nb_lus = read(fd1, tampon, TAILLE_TAMPON);
42                  if (nb_lus >= 0) {       /* on copie ce qu'on a lu */
43                      write(sortie, tampon, nb_lus);
44                  } else {            /* fin de fd1 : on l'enlève */
45                      close(fd1);
46                      nb_ouverts--;
47                      FD_CLR(fd1, &ouverts);
48                  }
49              }

50              if (FD_ISSET(fd2, &prets)) {
51                  /* si fd2 est prêt... */
52                  int nb_lus = read(fd2, tampon, TAILLE_TAMPON);
53                  if (nb_lus >= 0) {
54                      write(sortie, tampon, nb_lus);
55                  } else {
56                      close(fd2);
57                      nb_ouverts--;
58                      FD_CLR(fd2, &ouverts);
59                  }
60              }
61          }
62      }

63      int main(int argc, char *argv[])
64      {
65          int fd1, fd2;

66          if (argc != 3) {
67              fprintf(stderr, "Usage : %s f1 f2\n", argv[0]);
68              return EXIT_FAILURE;
69          }

70          fd1 = open(argv[1], O_RDONLY);
71          if (fd1 < 0)
72              abandon("Ouverture fichier 1 refusée");

73          fd2 = open(argv[2], O_RDONLY);
```

```
74          if (fd2 < 0)
75              abandon("Ouverture fichier 2 refusée");

76          mixer(fd1, fd2, 1);

77          return EXIT_SUCCESS;
78      }
```

9.2 Attente de données avec délai maximum

L'exemple suivant montre comment utiliser la limite de temps dans le cas (fréquent) d'attente sur un seul descripteur.

Une fonction utilitaire :

```
 1      /*
 2        SelectFifo/AttenteDonnees.c

 3        attente de données provenant d'un descripteur ouvert,
 4        avec délai maximum
 5      */

 6      #include <sys/time.h>
 7      #include <sys/types.h>
 8      #include <sys/stat.h>
 9      #include <unistd.h>
10      #include <stdio.h>
11      #include <stdlib.h>
12      #include <fcntl.h>

13      /*
14        fonction attendre_donnees

15        paramètres:
16        - un descripteur ouvert
17        - une durée d'attente en millisecondes

18        rôle: attend que des données arrivent sur le descripteur pendant
19        un certain temps.

20        retourne:
21        1  si des données sont arrivées
22        0  si le délai est dépassé
23        -1 en cas d'erreur. Voir variable errno.
24      */

25      int attendre_donnees(int fd, int millisecondes)
26      {
27          fd_set set;
28          struct timeval delai;

29          FD_ZERO(&set);
30          FD_SET(fd, &set);
31          delai.tv_sec = millisecondes / 1000;
32          delai.tv_usec = (millisecondes % 1000) * 1000;
```

```
33          return select(FD_SETSIZE, &set, NULL, NULL, &delai);
34      }
```

Le programme principal :

```
1       /*
2         SelectFifo/lecteur.c

3         Exemple de lecture avec délai (timeout).
4         M. Billaud, Septembre 2002

5         Ce programme attend des lignes de texte provenant d'une fifo, et les
6         affiche.
7         En attendant de recevoir les lignes, il affiche une petite étoile
8         tournante (par affichage successif des symboles  - \ | et /).

9         Exemple d'utilisation:
10        - créer une fifo :  mkfifo /tmp/mafifo
11        - dans une fenêtre, lancer : lecteur /tmp/mafifo
12        le programme se met en attente
13        - dans une autre fenêtre, faire
14        cat > /tmp/mafifo
15        puis taper quelques lignes de texte.

16      */

17      #include <sys/time.h>
18      #include <sys/types.h>
19      #include <sys/stat.h>
20      #include <unistd.h>
21      #include <stdio.h>
22      #include <stdlib.h>
23      #include <fcntl.h>

24      #define TAILLE_TAMPON 100
25      #define DELAI         500       /* millisecondes */

26      extern int attendre_donnees(int fd, int millisecondes);

27      int main(int argc, char *argv[])
28      {
29          int fd;
30          char symbole[] = "-\\|-";
31          int n = 0;
32          int numero_ligne = 1;

33          if (argc != 2) {
34              fprintf(stderr, " Usage: %s fifo\n", argv[0]);
35              exit(EXIT_FAILURE);
36          }
37          printf("> Ouverture fifo %s ...\n", argv[1]);
38          fd = open(argv[1], O_RDONLY);
39          if (fd == -1) {
40              fprintf(stderr, "Ouverture refusée\n");
41              exit(EXIT_FAILURE);
42          };
```

```
43          printf("OK\n");

44          for (;;) {
45              int r = attendre_donnees(fd, DELAI);
46              if (r == 0) {
47                  /* délai dépassé, on affiche un caractère suivi par backspace */
48                  printf("%c\b", symbole[n++ % 4]);
49                  fflush(stdout);
50              } else {
51                  /* on a reçu quelque chose */
52                  char tampon[TAILLE_TAMPON];
53                  int nb_lus;
54                  nb_lus = read(fd, tampon, TAILLE_TAMPON - 1);
55                  if (nb_lus <= 0)
56                      break;
57                  tampon[nb_lus] = '\0';
58                  printf("%4d %s", numero_ligne++, tampon);
59              }
60          }
61          close(fd);
62          exit(EXIT_SUCCESS);
63      }
```

10 Communication interprocessus par sockets locaux

Les *sockets* (*prises*) sont un moyen générique de faire communiquer des processus entre eux. C'est le moyen standard utilisé pour la communication sur le réseau Internet, mais on peut l'employer également pour la communication locale entre processus tournant sur une même machine (commes les FIFOs et les pipes, et les files de messages IPC que nous verrons plus loin).

10.1 Création d'un socket

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

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

La fonction `socket()` crée une nouvelle prise et retourne un descripteur qui servira ensuite aux lectures et écritures. Le paramètre `domain` indique le « domaine de communication » utilisé, qui est `PF_LOCAL` ou (synonyme) `PF_UNIX` pour les communications locales.[4]

Le *type* indique le style de communication désiré entre les deux participants. Les deux styles principaux sont
- `SOCK_DGRAM` : communication par messages (blocs contenant des octets) appelés *datagrammes*
- `SOCK_STREAM` : la communication se fait par un flot (bidirectionnel) d'octets une fois que la connection est établie.

Fiabilité : la fiabilité des communication par datagrammes est garantie pour le domaine local, mais ce n'est pas le cas pour les domaines « réseau » que nous verrons plus loin : les datagrammes peuvent être perdus, dupliqués, arriver dans le désordre etc. et c'est au programmeur d'application d'en tenir compte. Par contre la fiabilité des « streams » est assurée par les couches basses du système de communication, évidemment au prix d'un surcoût (numérotation des paquets, accusés de réception, temporisations, retranmissions, etc).

[4]Le domaine définit une famille de protocoles (protocol family) utilisables. Autres familles disponibles : `PF_INET` protocoles internet IPv4, `PF_INET6` protocoles internet IPv6, `PF_IPX` protocoles Novel IPX, `PF_X25` protocoles X25 (ITU-T X.25 / ISO-8208), `PF_APPLETALK`Appletalk, etc.

Enfin, le paramètre `protocol` indique le protocole sélectionné. La valeur `0` correspond au protocole par défaut pour le domaine et le type indiqué.

10.2 Adresses

La fonction `socket()` crée un socket anonyme. Pour qu'un autre processus puisse le désigner, il faut lui associer un *nom* par l'intermédiaire d'une *adresse* contenue dans une structure `sockaddr_un` :

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

struct sockaddr_un {
        sa_family_t  sun_family;              /* AF_UNIX */
        char         sun_path[UNIX_PATH_MAX]; /* pathname */
};
```

Ces adresses sont des chemins d'accès dans l'arborescence des fichiers et répertoires.

Exemple :

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

socklen_t longueur_adresse;

struct sockaddr_un  adresse;

adresse.sun_family      = AF_LOCAL;
strcpy(adresse.sun_path,"/tmp/xyz");
longueur_adresse = SUN_LEN (&dresse);
```

L'association d'une adresse à un socket se fait par `bind()` (voir exemples plus loin).

10.3 Communication par datagrammes

Dans l'exemple développé ici, un serveur affiche les datagrammes émis par les clients.

10.3.1 La réception de datagrammes

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

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

int  recvfrom(int  s,  void  *buf,  size_t len, int flags,
              struct sockaddr *from, socklen_t *fromlen);
```

La fonction `bind()` permet de nommer le socket de réception. La fonction `recvfrom()` attend l'arrivée d'un datagramme qui est stocké dans les `len` premiers octets du tampon `buff`. Si `from` n'est pas `NULL`, l'adresse du socket émetteur[5] est placée dans la structure pointée par `from`, dont la longueur maximale est contenue dans l'entier pointé par `fromlen`.

[5] qui peut servir à expédier une réponsee

Si la lecture a réussi, la fonction retourne le nombre d'octets du message lu, et la longueur de l'adresse est mise à jour.

Le paramètre `flags` permet de préciser des options.

```
1   /* serveur-dgram-local.c */

2   /*
3      Usage:  serveur-dgram-local chemin

4      Reçoit des datagrammes par un socket du domain local,
5      et les affiche. Le paramètre indique le nom du socket.
6      S'arrête quand la donnée est "stop".
7   */

8   #include <stdio.h>
9   #include <unistd.h>
10  #include <stdlib.h>
11  #include <sys/types.h>
12  #include <sys/socket.h>
13  #include <sys/un.h>

14  #define TAILLE_MAX_DONNEE 1024

15  void abandon(char message[])
16  {
17      perror(message);
18      exit(EXIT_FAILURE);
19  }

20  int main(int argc, char *argv[])
21  {
22      socklen_t longueur_adresse;
23      struct sockaddr_un adresse;
24      int fd;

25      if (argc != 2) {
26          fprintf(stderr, "usage: %s chemin\n", argv[0]);
27          abandon("mauvais nombre de parametres");
28      }

29      adresse.sun_family = AF_LOCAL;
30      strcpy(adresse.sun_path, argv[1]);
31      longueur_adresse = SUN_LEN(&adresse);

32      fd = socket(PF_LOCAL, SOCK_DGRAM, 0);
33      if (fd < 0)
34          abandon("Création du socket serveur");

35      if (bind(fd, (struct sockaddr *) &adresse, longueur_adresse) < 0)
36          abandon("Nommage du socket serveur");

37      printf("> Serveur démarré sur socket local \"%s\"\n", argv[1]);
38      while (1) {
39          int lg;
40          /* un caractère supplémentaire permet d'ajouter le
```

```
41                terminateur de chaîne, qui n'est pas transmis */
42            char tampon[TAILLE_MAX_DONNEE + 1];
43            lg = recvfrom(fd, tampon, TAILLE_MAX_DONNEE, 0, NULL, NULL);
44            if (lg <= 0)
45                abandon("Réception datagramme");
46            tampon[lg] = '\0';      /* ajout terminateur */
47            printf("Reçu : %s\n", tampon);
48            if (strcmp(tampon, "stop") == 0) {
49                printf("> arrêt demandé\n");
50                break;
51            }
52        }
53        close(fd);
54        unlink(argv[1]);
55        return EXIT_SUCCESS;
56    }
```

10.3.2 Emission de datagrammes

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

int  sendto(int s, const void *msg, size_t len, int flags,
            const struct sockaddr *to, socklen_t tolen);
```

`sendto` utilise le descripteur de socket `s` pour envoyer le message formé des `len` premiers octets de `msg` à l'adresse de longueur `tolen` pointée par `to`.

Le même descripteur peut être utilisé pour des envois à des adresses différentes.

```
1     /* client-dgram-local.c */

2     /*
3        Usage:  client-dgram-local chemin messages

4        Envoie des datagrammes à un socket du domain local,
5        et les affiche. Le premier paramètre indique le nom du socket,
6        les autres des chaînes de caractères.

7        Exemple : client-dgram-local un deux "trente et un" stop
8     */

9     #include <stdio.h>
10    #include <unistd.h>
11    #include <stdlib.h>
12    #include <sys/types.h>
13    #include <sys/socket.h>
14    #include <sys/un.h>

15    #define TAILLE_MAX_DONNEE 1024

16    void abandon(char message[])
17    {
18        perror(message);
19        exit(EXIT_FAILURE);
20    }
```

```
21  int main(int argc, char *argv[])
22  {
23      socklen_t longueur_adresse;
24      struct sockaddr_un adresse;
25      int fd;
26      int k;

27      if (argc <= 2) {
28          fprintf(stderr, "usage: %s chemin message\n", argv[0]);
29          abandon("mauvais nombre de paramètres");
30      }

31      adresse.sun_family = AF_LOCAL;
32      strcpy(adresse.sun_path, argv[1]);
33      longueur_adresse = SUN_LEN(&adresse);

34      fd = socket(PF_LOCAL, SOCK_DGRAM, 0);
35      if (fd < 0)
36          abandon("Création du socket client");

37      for (k = 2; k < argc; k++) {
38          int lg;
39          lg = strlen(argv[k]);
40          if (lg > TAILLE_MAX_DONNEE)
41              lg = TAILLE_MAX_DONNEE;

42          /* le message est envoyé sans le terminateur '\0' */
43          if (sendto(fd, argv[k], lg, 0,
44                     (struct sockaddr *) &adresse,
45                     longueur_adresse) < 0)
46              abandon("Expédition du message");

47          printf(".");
48          fflush(stdout);
49          sleep(1);
50      }

51      printf("OK\n");
52      close(fd);
53      return EXIT_SUCCESS;
54  }
```

Exercice : modifier les programmes précédents pour que le serveur envoie une réponse qui sera affichée par le client[6].

10.3.3 Emission et réception en mode connecté

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

int  connect(int sockfd,
             const struct sockaddr *serv_addr,
```

[6]Penser à attribuer un nom au socket du client pour que le serveur puisse lui répondre, par exemple avec l'aide de la fonction `tempnam()`.

```
              socklen_t addrlen);

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

Un émetteur qui va envoyer une série de messages au même destinataire par le même socket peut faire préalablement un `connect()` pour indiquer une destination par défaut, et employer ensuite `send()` à la place de `sendto()`.

Le récepteur qui ne s'intéresse pas à l'adresse de l'envoyeur peut utiliser `recv()`.

Exercice : modifier les programmes précédents pour utiliser `recv()` et `send()`.

10.4 Communication par flots

Dans ce type de communication, c'est une suite d'octets qui est transmises (et non pas une suite de messages comme dans la communication par datagrammes).

Les sockets locaux de ce type sont crées par

```
int fd = socket(PF_LOCAL,SOCK_STREAM,0);
```

10.4.1 Architecture client-serveur

La plupart des applications communicantes sont conçues selon une structure dissymétrique : l'architecture client-serveur, dans laquelle un processus serveur est contacté par plusieurs clients.

Le client crée un socket (`socket()`), qu'il met en relation (par `connect()`) avec celui du serveur. Les données sont échangées par `read()`, `write()` ... et le socket est fermé par `close()`.

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

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

Du côté serveur : un socket est crée et une adresse lui est associée (`socket()` + `bind()`). Un `listen()` prévient le système que ce socket recevra des demandes de connexion, et précise le nombre de connexions que l'on peut mettre en file d'attente.

Le serveur attend les demandes de connexion par la fonction `accept()` qui renvoit un descripteur, lequel permet la communication avec le client.

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

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

int  listen(int s, int backlog);

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

Remarque : il est possible ne fermer qu'une "moitié" de socket : `shutdown(1)` met fin aux émissions (causant une "fin de fichier" chez le correspondant), `shutdown(0)` met fin aux réceptions.

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

int shutdown(int s, int how);
```

Le client :

```
1   /*
2     client-stream

3     Envoi/réception de données par un socket local (mode connecté)

4     Exemple de client qui
5     - ouvre un socket
6     - envoie sur ce socket du texte lu sur l'entree standard
7     - attend et affiche une réponse
8     */

9   #include <unistd.h>
10  #include <sys/types.h>
11  #include <sys/socket.h>
12  #include <sys/un.h>
13  #include <signal.h>
14  #include <stdio.h>
15  #include <string.h>
16  #include <stdlib.h>

17  #define TAILLE_TAMPON 1000

18  void abandon(char message[])
19  {
20      perror(message);
21      exit(EXIT_FAILURE);
22  }

23  int main(int argc, char *argv[])
24  {
25      char *chemin;               /* chemin d'accès du socket serveur */
26      socklen_t longueur_adresse;
27      struct sockaddr_un adresse;
28      int fd;

29      /* 1. réception des paramètres de la ligne de commande */
30      if (argc != 2) {
31          printf("Usage: %s chemin\n", argv[0]);
32          abandon("Mauvais nombre de paramètres");
33      }
34      chemin = argv[1];

35      /* 2. Initialisation du socket */

36      /* 2.1 création du socket   */
37      fd = socket(PF_LOCAL, SOCK_STREAM, 0);
```

```
38          if (fd < 0)
39              abandon("Création du socket client");

40          /* 2.2 Remplissage adresse serveur */
41          adresse.sun_family = AF_LOCAL;
42          strcpy(adresse.sun_path, chemin);
43          longueur_adresse = SUN_LEN(&adresse);

44          /* 2.3 connexion au serveur */
45          if (connect(fd, (struct sockaddr *) &adresse, longueur_adresse)
46              < 0)
47              abandon("connect");

48          printf("CLIENT> Connexion établie\n");

49          /* 3. Lecture et envoi des données */
50          for (;;) {
51              char tampon[TAILLE_TAMPON];
52              int nb_lus, nb_envoyes;

53              nb_lus = read(0, tampon, TAILLE_TAMPON);
54              if (nb_lus <= 0)
55                  break;
56              nb_envoyes = write(fd, tampon, nb_lus);
57              if (nb_envoyes != nb_lus)
58                  abandon("envoi données");
59          }
60          /* 4. Fin de l'envoi */
61          shutdown(fd, 1);
62          printf("CLIENT> Fin envoi, attente de la réponse.\n");

63          /* 5. Réception et affichage de la réponse */
64          for (;;) {
65              char tampon[TAILLE_TAMPON];
66              int nb_lus;
67              nb_lus = read(fd, tampon, TAILLE_TAMPON - 1);
68              if (nb_lus <= 0)
69                  break;
70              tampon[nb_lus] = '\0';  /* ajout d'un terminateur de chaîne */
71              printf("%s", tampon);
72          }
73          /* et fin */
74          close(fd);
75          printf("CLIENT> Fin.\n");
76          return EXIT_SUCCESS;
77      }
```

Le serveur est programmé ici de façon atypique, puisqu'il traite qu'il traîte une seule communication à la fois. Si le client fait traîner les choses, les autres clients en attente resteront bloqués longtemps.

```
1   /*
2     serveur-stream

3     Envoi/réception de données par un socket local (mode connecté)

4     Exemple de serveur qui
5     - attend une connexion
```

```
 6       - lit du texte
 7       - envoie une réponse

 8       Remarques
 9       - ce serveur ne traite qu'une connexion à la fois.
10       - Il ne s'arrête jamais.
11      */

12      #include <unistd.h>
13      #include <sys/types.h>
14      #include <sys/socket.h>
15      #include <sys/un.h>
16      #include <signal.h>
17      #include <stdio.h>
18      #include <stdlib.h>
19      #include <ctype.h>

20      #define TAILLE_TAMPON             1000
21      #define MAX_CONNEXIONS_EN_ATTENTE 4

22      void abandon(char message[])
23      {
24          perror(message);
25          exit(EXIT_FAILURE);
26      }

27      int main(int argc, char *argv[])
28      {
29          int fd_serveur;
30          struct sockaddr_un adresse;
31          size_t taille_adresse;
32          char *chemin;
33          char tampon[TAILLE_TAMPON];
34          int nb_car;
35          int numero_connexion = 0;

36          /* 1. réception des paramêtres de la ligne de commande */

37          if (argc != 2) {
38              printf("usage: %s chemin\n", argv[0]);
39              abandon("Mauvais nombrede paramêtres");
40          }
41          chemin = argv[1];

42          /* 3. Initialisation du socket de réception */
43          /* 3.1 création du socket   */
44          fd_serveur = socket(PF_LOCAL, SOCK_STREAM, 0);
45          if (fd_serveur < 0)
46              abandon("Création du socket serveur");

47          /* 3.2 Remplissage adresse serveur */
48          adresse.sun_family = AF_LOCAL;
49          strcpy(adresse.sun_path, chemin);
50          taille_adresse = SUN_LEN(&adresse);
```

```
51          /* 3.3 Association de l'adresse au socket */
52          taille_adresse = sizeof adresse;
53          if (bind(fd_serveur, (struct sockaddr *) &adresse, taille_adresse)
54              < 0)
55              abandon("bind");

56          /* 3.4 Ce socket attend des connexions mises en file d'attente */
57          listen(fd_serveur, MAX_CONNEXIONS_EN_ATTENTE);

58          printf("SERVEUR> Le serveur écoute le socket %s\n", chemin);

59          /* 4. boucle du serveur */
60          for (;;) {
61              int fd_client;
62              int compteur = 0;

63              /* 4.1 attente d'une connexion */
64              printf("SERVEUR> Attente d'une connexion.\n");
65              fd_client = accept(fd_serveur, NULL, NULL);
66              if (fd_client < 0)
67                  abandon("accept");

68              numero_connexion++;
69              printf("SERVEUR> Connexion #%d établie.\n", numero_connexion);

70              /* 4.2 lecture et affichage des données envoyées par le client */
71              for (;;) {
72                  int nb_lus;
73                  nb_lus = read(fd_client, tampon, TAILLE_TAMPON - 1);
74                  if (nb_lus <= 0)
75                      break;
76                  compteur += nb_lus;
77                  tampon[nb_lus] = '\0';
78                  printf("%s", tampon);
79              }

80              /* 4.4 plus de données, on envoit une réponse */
81              printf("SERVEUR> envoi de la réponse.\n");
82              nb_car = sprintf(tampon, "*** Fin de la connexion #%d\n",
83                               numero_connexion);
84              write(fd_client, tampon, nb_car);
85              nb_car =
86                  sprintf(tampon, "*** Vous m'avez envoyé %d caractères\n",
87                          compteur);
88              write(fd_client, tampon, nb_car);
89              nb_car = sprintf(tampon, "*** Merci et à bientot.\n");
90              write(fd_client, tampon, nb_car);

91              /* 4.4 fermeture de la connexion */
92              close(fd_client);
93              printf("SERVEUR> fin de connexion.\n");
94          }

95          /* on ne passe jamais ici */
96          return EXIT_SUCCESS;
```

```
97      }
```

Dans une programmation plus classique, le serveur lance un processus (par `fork()`, voir plus loin) dès qu'une connexion est établie, et délègue le traitement de la connexion à ce processus.

Une autre technique est envisageable pour traiter plusieurs connexions par un processus unique : le serveur maintient une liste de descripteurs ouverts, et fait une boucle autour d'un `select()`, en attente de données venant

- soit du descripteur "principal" ouvert par le serveur. Dans ce cas il effectue ensuite un `accept(...)` qui permettra d'ajouter un nouveau client à la liste.
- soit d'un des descripteurs des clients, et il traite alors les données venant de ce client (il l'enlève de la liste en fin de communication).

Cette technique conduit à des performances nettement supérieures aux serveurs multiprocessus ou multithreads (pas de temps perdu à lancer des processus), au prix d'une programmation qui oblige le programmeur à gérer lui-même le "contexte de déroulement" de chaque processus.

```
1       /*
2         serveur-stream-monotache

3         Envoi/réception de données par un socket local (mode connecté)

4         Exemple de serveur monotâche qui gère plusieurs connexions

5         - attend une connexion
6         - lit du texte
7         - envoie une réponse
8        */

9       #include <unistd.h>
10      #include <sys/types.h>
11      #include <sys/socket.h>
12      #include <sys/un.h>
13      #include <signal.h>
14      #include <stdio.h>
15      #include <stdlib.h>
16      #include <ctype.h>
17      #include <assert.h>

18      #define TAILLE_TAMPON             1000
19      #define MAX_CONNEXIONS_EN_ATTENTE 4
20      #define MAX_CLIENTS               10

21      /* les données propres à chaque client */

22      #define INACTIF -1
23      struct {
24          int fd;
25          int numero_connexion;
26          int compteur;
27      } client[MAX_CLIENTS];

28      void abandon(char message[])
29      {
30          perror(message);
31          exit(EXIT_FAILURE);
32      }
```

```
33  int main(int argc, char *argv[])
34  {
35      int fd_serveur;
36      struct sockaddr_un adresse;
37      size_t taille_adresse;
38      char *chemin;
39      int nb_connexions = 0;
40      int i;
41      int nbfd;

42      /* 1. réception des paramètres de la ligne de commande */

43      if (argc != 2) {
44          printf("usage: %s chemin\n", argv[0]);
45          abandon("mauvais nombre de paramètres");
46      }
47      chemin = argv[1];

48      /* 3. Initialisation du socket de réception */
49      /* 3.1 création du socket   */
50      fd_serveur = socket(PF_LOCAL, SOCK_STREAM, 0);
51      if (fd_serveur < 0)
52          abandon("Création du socket serveur");

53      /* 3.2 Remplissage adresse serveur */
54      adresse.sun_family = AF_LOCAL;
55      strcpy(adresse.sun_path, chemin);
56      taille_adresse = SUN_LEN(&adresse);

57      /* 3.3 Association de l'adresse au socket */
58      taille_adresse = sizeof adresse;
59      if (bind(fd_serveur, (struct sockaddr *) &adresse, taille_adresse)
60          < 0)
61          abandon("bind");

62      /* 3.4 Ce socket attend des connexions mises en file d'attente */
63      listen(fd_serveur, MAX_CONNEXIONS_EN_ATTENTE);

64      printf("SERVEUR> Le serveur écoute le socket %s\n", chemin);

65      /* 3.5 initialisation du tableau des clients */
66      for (i = 0; i < MAX_CLIENTS; i++) {
67          client[i].fd = INACTIF;
68      }

69      /* 4. boucle du serveur */

70      for (;;) {
71          fd_set lectures;

72          /* 4.1 remplissage des masques du select */
73          FD_ZERO(&lectures);
74          FD_SET(fd_serveur, &lectures);
75          for (i = 0; i < MAX_CLIENTS; i++) {
```

```
76                  if (client[i].fd != INACTIF)
77                      FD_SET(client[i].fd, &lectures);
78              }

79              /* 4.2 attente d'un événement (ou plusieurs) */
80              nbfd = select(FD_SETSIZE, &lectures, NULL, NULL, NULL);
81              assert(nbfd >= 0);

82              /* 4.3 en cas de nouvelle connexion : */
83              if (FD_ISSET(fd_serveur, &lectures)) {
84                  /* si il y a de la place dans la table des clients,
85                     on y ajoute la nouvelle connexion */
86                  nbfd--;
87                  for (i = 0; i < MAX_CLIENTS; i++) {
88                      if (client[i].fd == INACTIF) {
89                          int fd_client;
90                          fd_client = accept(fd_serveur, NULL, NULL);
91                          if (fd_client < 0)
92                              abandon("accept");

93                          client[i].fd = fd_client;
94                          client[i].numero_connexion = ++nb_connexions;
95                          client[i].compteur = 0;
96                          printf
97                              ("SERVEUR> arrivée de connexion #%d (fd %d)\n",
98                               client[i].numero_connexion, fd_client);
99                          break;
100                     }
101                     if (i >= MAX_CLIENTS) {
102                         printf("SERVEUR> trop de connexions !\n");
103                     }
104                 }
105             };

106             /* 4.4 traitement des clients actifs qui ont reçu des données */
107             for (i = 0; (i < MAX_CLIENTS) && (nbfd > 0); i++) {
108                 if ((client[i].fd != INACTIF) &&
109                     FD_ISSET(client[i].fd, &lectures)) {
110                     int nb_lus;
111                     char tampon[TAILLE_TAMPON];
112                     nbfd--;
113                     nb_lus =
114                         read(client[i].fd, tampon, TAILLE_TAMPON - 1);
115                     printf("SERVEUR> données reçues de #%d (%d octets)\n",
116                            client[i].numero_connexion, nb_lus);
117                     if (nb_lus > 0)
118                         client[i].compteur += nb_lus;
119                     else {
120                         int nb_car;
121                         printf
122                             ("SERVEUR> envoi de la réponse au client #%d.\n",
123                              client[i].numero_connexion);
124                         nb_car =
125                             sprintf(tampon,
126                                     "*** Fin de la connexion #%d\n",
```

```
127                                     client[i].numero_connexion);
128                         write(client[i].fd, tampon, nb_car);
129                         nb_car =
130                             sprintf(tampon,
131                                     "*** Vous m'avez envoyé %d caractères\n",
132                                     client[i].compteur);
133                         write(client[i].fd, tampon, nb_car);
134                         nb_car =
135                             sprintf(tampon, "*** Merci et à bientôt.\n");
136                         write(client[i].fd, tampon, nb_car);
137                         close(client[i].fd);
138                         /* enlèvement de la liste des clients */
139                         client[i].fd = INACTIF;
140                     }
141                 }
142             }
143             assert(nbfd == 0);      /* tous les descripteurs ont été traités */
144         }
145         /* on ne passe jamais ici (boucle sans fin) */
146         return EXIT_SUCCESS;
147     }
```

10.5 socketpair()

La fonction `socketpair()` construit une paire de sockets locaux, bi-directionnels, reliés l'un à l'autre.

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

int socketpair(int d, int type, int protocol, int sv[2]);
```

Dans l'état actuel des implémentations, le paramètre `d` (domaine) doit être égal à `AF_LOCAL`, et `type` à `SOCK_DGRAM` ou `SOCK_STREAM`, avec le protocole par défaut (valeur `0`).

Cette fonction remplit le tableau `sv[]` avec les descripteurs de deux sockets du type indiqué. Ces deux sockets sont reliés entre eux et bidirectionnels : ce qu'on écrit sur le descripteur `sv[0]` peut être lu sur `sv[1]`, et réciproquement.

On utilise `socketpair()` comme `pipe()`, pour la communication entre descendants d'un même processus [7]. `socketpair()` possède deux avantages sur `pipe()` : la possibilité de transmettre des datagrammes, et la bidirectionnalité.

Exemple :

```
1   /*
2     paire-send.c

3     échange de données à travers des sockets locaux créés par
4     socketpair()
5   */

6   #include <unistd.h>
7   #include <stdio.h>
8   #include <stdlib.h>
9   #include <sys/socket.h>
```

[7] au sens large, ce qui inclue la communication d'un processus avec un de ses fils

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

11  struct paquet {
12      int type;
13      int valeur;
14  };

15  /* les différents types de paquets */

16  #define DONNEE   0
17  #define RESULTAT 1
18  #define FIN      2

19  void abandon(char message[])
20  {
21      perror(message);
22      exit(EXIT_FAILURE);
23  }

24  int additionneur(int fd)
25  {
26      /*
27         calcule la somme des entiers qui arrivent sur le descripteur,
28         renvoie le résultat
29       */
30      int somme = 0;
31      struct paquet p;
32      int n;

33      for (;;) {
34          n = recv(fd, &p, sizeof p, 0);
35          if (n < 0)
36              abandon("recv");
37          printf("additionneur: réception d'un paquet contenant ");
38          if (p.type == FIN) {
39              printf("la marque de fin\n");
40              break;
41          }
42          printf("la donnée %d\n", p.valeur);
43          somme += p.valeur;
44      };
45      /* envoi reponse */
46      p.type = RESULTAT;
47      p.valeur = somme;
48      printf("additionneur: envoi du total %d\n", somme);
49      n = send(fd, &p, sizeof p, 0);
50      if (n < 0)
51          abandon("additionneur: send");
52      return EXIT_SUCCESS;
53  }

54  int generateur(int fd)
55  {
56      /*
57         envoie une suite d'entiers, récupère et affiche le résultat.
```

```
58          */
59          struct paquet p;
60          int i, n, resultat;

61          for (i = 1; i <= 5; i++) {
62              p.type = DONNEE;
63              p.valeur = i;
64              printf("générateur: envoi de la donnée %d\n", i);
65              n = send(fd, &p, sizeof p, 0);
66              if (n < 0)
67                  abandon("generateur: send");
68              sleep(1);
69          };
70          p.type = FIN;
71          printf("générateur: envoi de la marque de fin\n");
72          n = send(fd, &p, sizeof p, 0);
73          if (n < 0)
74              abandon("generateur: send");

75          printf("generateur: lecture du résultat\n");
76          n = recv(fd, &p, sizeof p, 0);
77          if (n < 0)
78              abandon("generateur: recv");
79          resultat = p.valeur;
80          printf("generateur: resultat reçu = %d\n", resultat);
81          return EXIT_SUCCESS;
82      }

83      int main()
84      {
85          int paire_sockets[2];

86          socketpair(AF_LOCAL, SOCK_DGRAM, 0, paire_sockets);

87          if (fork() == 0) {
88              close(paire_sockets[0]);
89              return additionneur(paire_sockets[1]);
90          } else {
91              close(paire_sockets[1]);
92              return generateur(paire_sockets[0]);
93          }
94      }
```

11 Processus

11.1 `fork()`, `wait()`

```
#include <unistd.h>
pid_t fork(void);
pid_t wait(int *status)
```

La fonction `fork()` crée un nouveau processus (*fils*) semblable au processus courant (*père*). La valeur renvoyée n'est pas la même pour le fils (0) et pour le père (numéro de processus du fils). -1 indique un échec.

La fonction `wait()` attend qu'un des processus fils soit terminé. Elle renvoie le numéro du fils, et son `status` (voir `exit()`) en paramètre passé par adresse.

Attention. Le processus fils *hérite* des descripteurs ouverts de son père. Il convient que chacun des processus ferme les descripteurs qui ne le concernent pas.

Exemple :

```
 1  /* biproc.c */

 2  /*
 3   * Illustration de fork() et pipe();
 4   *
 5   * Exemple à deux processus reliés par un tuyau
 6   - l'un envoie abcdef...z 10 fois dans le tuyau
 7   - l'autre écrit ce qui lui arrive du tuyau sur la
 8   sortie standard, en le formattant.
 9   */

10  #include <unistd.h>
11  #include <stdlib.h>
12  #include <stdio.h>
13  #include <sys/types.h>
14  #include <sys/wait.h>

15  #define TAILLE_LIGNE       30
16  #define TAILLE_ALPHABET    26
17  #define NOMBRE_REPETITIONS 10

18  void abandon(char raison[])
19  {
20      perror(raison);
21      exit(EXIT_FAILURE);
22  }

23  void envoyer_donnees(int sortie)
24  {
25      char alphabet[TAILLE_ALPHABET];
26      int k;

27      for (k = 0; k < TAILLE_ALPHABET; k++) {
28          alphabet[k] = 'a' + k;
29      }

30      for (k = 0; k < NOMBRE_REPETITIONS; k++)
31          if (write(sortie, alphabet, TAILLE_ALPHABET) !=
32              TAILLE_ALPHABET) {
33              abandon("write");
34          }
35      close(sortie);
36  }

37  int lire_donnees(int fd, char *tampon, size_t taille_tampon)
38  {
39      /* lecture, en insistant pour remplir le tampon */
40      int deja_lus = 0;
```

```
41          int n;

42          while (deja_lus < taille_tampon) {
43              n = read(fd, tampon + deja_lus, taille_tampon - deja_lus);
44              if (n < 0)
45                  return (-1);
46              if (n == 0)
47                  break;              /* plus rien à lire */
48              deja_lus += n;
49          }
50          return deja_lus;
51      }

52      void recevoir_donnees(int entree)
53      {
54          char ligne[TAILLE_LIGNE + 1];
55          int nb, numero_ligne = 1;

56          while ((nb = lire_donnees(entree, ligne, TAILLE_LIGNE)) > 0) {
57              ligne[nb] = '\0';
58              printf("%3d %s\n", numero_ligne++, ligne);
59          };
60          if (nb < 0) {
61              abandon("read");
62          }
63          close(entree);
64      }

65      int main(void)
66      {
67          int fd[2], status;
68          pid_t fils;

69          if (pipe(fd) != 0)
70              abandon("Echec création pipe");

71          if ((fils = fork()) < 0)
72              abandon("echec fork");

73          if (fils == 0) {
74              /* le processus fils */
75              close(fd[0]);
76              close(1);

77              envoyer_donnees(fd[1]);
78          } else {
79              /* le processus père continue ici */
80              close(0);
81              close(fd[1]);

82              recevoir_donnees(fd[0]);

83              wait(&status);
84              printf("status fils = %d\n", status);
85          }
```

```
86          return EXIT_SUCCESS;
87      }
```

Exercice : Observez ce qui se passe si, dans la fonction `affiche()`, on remplace l'appel à `lire()` par un `read()` ? Et si on ne fait pas le `wait()` ?

11.2 waitpid()

La fonction `waitpid()` permet d'attendre l'arrêt d'un des processus fils désigné par son *pid* (n'importe lequel si *pid=-1*), et de récupérer éventuellement son code de retour. Elle retourne le numéro du processus fils.

L'option `WNOHANG` rend `waitpid` non bloquant (qui retourne alors `-1` si le processus attendu n'est pas terminé).

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

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

Exemple :

```
int pid_fils;
int status;

if( (pid_fils = fork()) != 0) {
   code_processus_fils();
   exit(EXIT_SUCCESS);
   };
...
if (waitpid(pid_fils,NULL,WMNOHANG) == -1)
   printf("Le processus fils n'est pas encore terminé\n");
...
```

11.3 exec()

```
#include <unistd.h>
int execv (const char *FILENAME, char *const ARGV[])
int execl (const char *FILENAME, const char *ARG0,...)
int execve(const char *FILENAME, char *const ARGV[], char *const ENV[])
int execle(const char *FILENAME, const char *ARG0,...char *const ENV[])
int execvp(const char *FILENAME, char *const ARGV[])
int execlp(const char *FILENAME, const char *ARG0, ...)
```

Ces fonctions font toutes la même chose : activer un exécutable *à la place* du processus courant. Elles diffèrent par la manière d'indiquer les paramètres.

– `execv()` : les paramètres de la commande sont transmis sous forme d'un tableau de pointeurs sur des chaînes de caractères (le dernier étant `NULL`). Exemple :

```
1     /* Divers/execv.c */

2     #include <unistd.h>
3     #include <stdio.h>
4     #include <stdlib.h>

5     #define CHEMIN_COMPILATEUR    "/usr/bin/gcc"
6     #define NOM_COMPILATEUR       "gcc"
```

```
 7    #define TAILLE_MAX_PREFIXE    10
 8    #define TAILLE_MAX_NOMFICHIER 100

 9    int main(void)
10    {
11        char prefixe[TAILLE_MAX_PREFIXE];
12        char nom_source[TAILLE_MAX_NOMFICHIER];

13        char *parametres[] = {
14            NOM_COMPILATEUR,
15            NULL,                   /* exmplacement pour le nom du fichier source */
16            "-o",
17            NULL,                   /* emplacement pour le nom de l'exécutable   */
18            NULL                    /* fin des paramètres       */
19        };

20        printf("préfixe du fichier à compiler : ");
21        scanf("%s", prefixe);       /* dangereux */

22        snprintf(nom_source, TAILLE_MAX_NOMFICHIER, "%s.c", prefixe);

23        parametres[1] = nom_source;
24        parametres[3] = prefixe;

25        execv(CHEMIN_COMPILATEUR, parametres);

26        perror("execv");            /* normalement on ne passe pas ici */
27        return EXIT_FAILURE;
28    }
```

– `execl()` reçoit un nombre variable de paramètres. Le dernier est `NULL`). Exemple :

```
 1    /* Divers/execl.c */

 2    #include <unistd.h>
 3    #include <stdio.h>
 4    #include <stdlib.h>

 5    #define CHEMIN_COMPILATEUR    "/usr/bin/gcc"
 6    #define NOM_COMPILATEUR       "gcc"
 7    #define TAILLE_MAX_PREFIXE    10
 8    #define TAILLE_MAX_NOMFICHIER 100

 9    int main(void)
10    {
11        char prefixe[TAILLE_MAX_PREFIXE];
12        char nom_fichier[TAILLE_MAX_NOMFICHIER];

13        printf("Préfixe du fichier à compiler : ");
14        scanf("%s", prefixe);       /* dangereux */

15        snprintf(nom_fichier, TAILLE_MAX_NOMFICHIER, "%s.c", prefixe);

16        execl(CHEMIN_COMPILATEUR,
17              NOM_COMPILATEUR, nom_fichier, "-o", prefixe, NULL);

18        perror("execl");            /* on ne passe jamais ici */
```

```
19          return EXIT_FAILURE;
20      }
```

- `execve()` et `execle()` ont un paramètre supplémentaire pour préciser l'*environnement*.
- `execvp()` et `execlp()` utilisent la variable d'environnement `PATH` pour localiser l'exécutable à lancer. On pourrait donc écrire simplement :

```
execlp("gcc","gcc",fichier,"-o",prefixe,NULL);
```

11.4 Numéros de processus : `getpid()`, `getppid()`

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

`getpid()` permet à un processus de connaître son propre numéro, et `getppid()` celui de son père.

11.5 Programmation d'un démon

Les *démons*[8] sont des processus qui tournent normalement en arrière-plan pour assurer un service. Pour programmer correctement un démon, il ne suffit pas de faire un `fork()`, il faut aussi s'assurer que le processus restant ne bloque pas de ressources. Par exemple il doit libérer le terminal de contrôle du processus, revenir à la racine, faute de quoi il empêchera le démontage éventuel du système de fichiers à partir duquel il a été lancé.

```
1   /* Divers/demon.c */

2   #include <unistd.h>
3   #include <stdlib.h>
4   #include <fcntl.h>

5   int devenir_demon(void)
6   {
7       int fd;

8       /* Le processus se dédouble, et le père se termine */
9       if (fork() != 0)
10          exit(EXIT_SUCCESS);

11      /* le processus fils devient le leader d'un nouveau
12         groupe de processus */
13      setsid();

14      /* le processus fils crée le processus démon, et
15         se termine */
16      if (fork() != 0)
17          exit(EXIT_SUCCESS);

18      /* le démon déménage vers la racine */
19      chdir("/");

20      /* l'entrée standard est redirigée vers /dev/null */
21      fd = open("/dev/null", O_RDWR);
```

[8]Traduction de l'anglais *daemon*, acronyme de « Disk And Extension MONitor », qui désignait une des parties résidentes d'un des premiers systèmes d'exploitation.

```
22          dup2(fd, 0);
23          close(fd);

24          /* et les sorties vers /dev/console */
25          fd = open("/dev/console", O_WRONLY);
26          dup2(fd, 1);
27          dup2(fd, 2);
28          close(fd);
29          return EXIT_SUCCESS;
30      }
```

Voir FAQ Unix : *1.7 How do I get my program to act like a daemon*

12 Signaux

12.1 signal()

```
#include <stdio.h>
void (*signal(int signum, void (*handler)(int)))(int);
```

La fonction `signal()` demande au système de lancer la fonction `handler` lorsque le signal `signum` est reçu par le processus courant. La fonction `signal()` renvoie la fonction qui était précédemment associée au même signal.

Il y a une trentaine de signaux différents, [9] parmi lesquels
– `SIGINT` (program interrupt, émis par Ctrl-C),
– `SIGTST` (terminal stop, émis par Ctrl-Z)
– `SIGTERM` (demande de fin de processus)
– `SIGKILL` (arrêt immédiat de processus)
– `SIGFPE` (erreur arithmétique),
– `SIGALRM` (fin de délai, voir fonction `alarm()`), etc.
La fonction *handler()* prend en paramètre le numéro du signal reçu, et ne renvoie rien.

Exemple :

```
1   /* sig.c */

2   #include <stdlib.h>
3   #include <signal.h>
4   #include <errno.h>
5   #include <unistd.h>
6   #include <stdio.h>

7   #define DELAI  1                /* secondes */

8   void traitement(int numero_signal)
9   {
10      printf("Signal %d => ", numero_signal);
11      switch (numero_signal) {
12      case SIGTSTP:
13          printf("Je m'endors....\n");
14          kill(getpid(), SIGSTOP);        /* auto-endormissement */
15          printf("Je me réveille !\n");
```

[9] La liste complète des signaux, leur signification et leur comportement sont décrits dans la page de manuel `signal` (chapitre 7 pour Linux).

```
16              signal(SIGTSTP, traitement);    /* repositionnement */
17              break;
18          case SIGINT:
19          case SIGTERM:
20              printf("Fin du programme.\n");
21              exit(EXIT_SUCCESS);
22              break;
23          }
24      }

25      int main(void)
26      {
27          signal(SIGTSTP, traitement);        /* si on reçoit contrôle-Z */
28          signal(SIGINT, traitement); /* si contrôle-C */
29          signal(SIGTERM, traitement);        /* si kill processus */

30          while (1) {
31              sleep(DELAI);
32              printf(".");
33              fflush(stdout);
34          }
35          printf("fin\n");
36          exit(EXIT_SUCCESS);
37      }
```

12.2 kill()

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

La fonction `kill()` envoie un signal à un processus.

12.3 alarm()

```
#include <unistd.h>
long alarm(long delai);
```

La fonction `alarm()` demande au système d'envoyer un signal `SIGALRM` au processus dans un délai fixé (en secondes). Si une alarme était déjà positionnée, elle est remplacée. Un délai nul supprime l'alarme existante.

12.4 pause()

La fonction `pause()` bloque le processus courant jusqu'à ce qu'il reçoive un signal.

```
#include<unistd.h>
int pause(void);
```

Exercice : Écrire une fonction équivalente à `sleep()`.

13 Les signaux Posix

Le comportement des signaux classiques d'UNIX est malheureusement différent d'une version à l'autre. On emploie donc de préférence les mécanismes définis par la norme POSIX, qui offrent de plus la possibilité de masquer des signaux.

13.1 Manipulation des ensembles de signaux

Le type `sigset_t` représente les ensembles de signaux.

```
#include <signal.h>

int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum);
```

La fonction `sigemptyset()` crée un ensemble vide, `sigaddset()` ajoute un élément, etc.

13.2 `sigaction()`

```
#include <signal.h>

int sigaction(int signum,
              const  struct  sigaction  *act,
              struct sigaction *oldact);
```

La fonction `sigaction()` change l'action qui sera exécutée lors de la réception d'un signal. Cette action est décrite par une structure `struct sigaction`

```
struct sigaction {
  void (*sa_handler)(int);
  void (*sa_sigaction)(int, siginfo_t *, void *);
  sigset_t sa_mask;
  int sa_flags;
  void (*sa_restorer)(void); /* non utilisé */
}
```

- `sa_handler` indique l'action associée au signal `signum`. Il peut valoir `SIG_DFL` (action par défaut), `SIG_IGN` (ignorer), ou un pointeur vers une fonction de traitement de lu signal.
- le masque `sa_mask` indique l'ensemble de signaux qui seront bloqués pendant l'exécution de ce signal. Le signal lui-même sera bloqué, sauf si `SA_NODEFER` ou `SA_NOMASK` figurent parmi les *flags*.

Le champ `sa_flags` contient une combinaison d'indicateurs, parmi lesquels
- `SA_NOCLDSTOP` pour le signal `SIGCHLD`, ne pas recevoir la notification d'arrêt des processus fils (quand les processus fils reçoivent `SIGSTOP`, `SIGTSTP`, `SIGTTIN` ou `SIGTTOU`).
- `SA_ONESHOT` ou `SA_RESETHAND` remet l'action par défaut quand le handler a été appelé (c'est le comportement par défaut du `signal()` classique).
- `SA_SIGINFO` indique qu'il faut utiliser la fonction `sa_sigaction()` à trois paramètres à la place de `sa_handler()`.

Exemple :

```
1   /* sig-posix.c */

2   #include <stdlib.h>
3   #include <signal.h>
4   #include <errno.h>
5   #include <unistd.h>
6   #include <stdio.h>

7   #define DELAI          1        /*secondes */
```

```
 8  #define NB_ITERATIONS 60

 9  void traiter_signal(int numero_signal)
10  {
11      struct sigaction rien, ancien_traitement;

12      printf("Signal %d => ", numero_signal);

13      switch (numero_signal) {
14      case SIGTSTP:
15          printf("J'ai reçu un SIGTSTP.\n");

16          /* on désarme le signal SIGTSTP, avec sauvegarde de
17             du "traitant" précédent */
18          rien.sa_handler = SIG_DFL;
19          rien.sa_flags = 0;
20          sigemptyset(&rien.sa_mask); /* rien à masquer */
21          sigaction(SIGTSTP, &rien, &ancien_traitement);

22          printf("Alors je m'endors....\n");
23          kill(getpid(), SIGSTOP);        /* auto-endormissement */
24          printf("On me réveille ?\n");

25          /* remise en place ancien traitement */
26          sigaction(SIGTSTP, &ancien_traitement, NULL);
27          printf("C'est reparti !\n");
28          break;
29      case SIGINT:
30      case SIGTERM:
31          printf("On m'a demandé d'arrêter le programme.\n");
32          exit(EXIT_SUCCESS);
33          break;
34      }
35  }

36  int main(void)
37  {
38      struct sigaction a;
39      int i;

40      a.sa_handler = traiter_signal;      /* fonction à lancer */
41      sigemptyset(&a.sa_mask);        /* rien à masquer */

42      sigaction(SIGTSTP, &a, NULL);   /* pause contrôle-Z */
43      sigaction(SIGINT, &a, NULL);    /* fin contrôle-C */
44      sigaction(SIGTERM, &a, NULL);   /* arrêt */

45      for (i = 1; i < NB_ITERATIONS; i++) {
46          sleep(DELAI);
47          printf("%d", i % 10);
48          fflush(stdout);
49      }

50      printf("Fin\n");
51      return EXIT_SUCCESS;
```

```
52      }
```

14 Les processus légers (Posix 1003.1c)

Les processus classiques d'UNIX possèdent des ressources séparées (espace mémoire, table des fichiers ouverts...). Lorsqu'un nouveau *fil d'exécution* (processus fils) est créé par `fork()`, il se voit attribuer une *copie* des ressources du processus père.

Il s'ensuit deux problèmes :
– problème de performances, puisque la duplication est un mécanisme coûteux
– problème de communication entre les processus, qui ont des variables séparées.
Il existe des moyens d'atténuer ces problèmes : technique du *copy-on-write* dans le noyau pour ne dupliquer les pages mémoires que lorsque c'est strictement nécessaire), utilisation de segments de mémoire partagée (IPC) pour mettre des données en commun. Il est cependant apparu utile de définir un mécanisme permettant d'avoir plusieurs *fils d'exécution* (threads) dans un même espace de ressources non dupliqué : c'est ce qu'on appelle les *processus légers*. Ces processus légers peuvent se voir affecter des priorités.

On remarquera que la commutation entre deux threads d'un même groupe est une opération économique, puisqu'il n'est pas utile de recharger entièrement la table des pages de la MMU.

Ces processus légers ayant vocation à communiquer entre eux, la norme Posix 1003.1c définit également des mécanismes de synchronisation : exclusion mutuelle (*mutex*), *sémaphores*, et *conditions*.

Remarque : les sémaphores ne sont pas définis dans les bibliothèques de AIX 4.2 et SVR4 d'ATT/Motorola. Ils existent dans Solaris et les bibliothèques pour Linux.

14.1 Threads

```
#include <pthread.h>

int pthread_create(pthread_t      *thread,
                   pthread_attr_t *attr,
                   void           *(*start_routine)(void *),
                   void           *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);
```

La fonction `pthread_create` demande le lancement d'un nouveau processus léger, avec les attributs indiqués par la structure pointée par `attr` (`NULL` = attributs par défaut). Ce processus exécutera la fonction `start_routine`, en lui donnant le pointeur `arg` en paramètre. L'identifiant du processus léger est rangé à l'endoit pointé par `thread`.

Ce processus léger se termine (avec un code de retour) lorsque la fonction qui lui est associée se termine par `return` *retcode*, ou lorsque le processus léger exécute un `pthread_exit` *(retcode)*.

La fonction `pthread_join` permet au processus père d'attendre la fin d'un processus léger, et de récupérer éventuellement son code de retour.

Priorités : Le fonctionnement des processus légers peut être modifié (priorités, algorithme d'ordonnancement, etc.) en manipulant les *attributs* qui lui sont associés. Voir les fonctions `pthread_attr_init`, `pthread_attr_destroy`, `pthread_attr_set-detachstate`, `pthread_attr_getdetachstate`, `pthread_attr_setschedparam`, `pthread_attr_getschedparam`, `pthread_attr_setschedpolicy`, `pthread_attr_getschedpolicy`, `pthread_attr_setinheritsched`, `pthread_attr_getinheritsched`, `pthread_attr_setscope`, `pthread_attr_getscope`.

14.2 Verrous d'exclusion mutuelle (mutex)

```
#include <pthread.h>

int pthread_mutex_init(pthread_mutex_t   *mutex,
                       const pthread_mutexattr_t *mutexattr);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

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

int pthread_mutex_trylock(pthread_mutex_t *mutex);
```

Les verrous d'exclusion mutuelle (`mutex`) sont créés par `pthread_mutex_init`. Il en est de différents types (rapides, récursifs, etc.), selon les attributs pointés par le paramètre `mutexattr`. La valeur par défaut (`mutexattr=NULL`) fait généralement l'affaire. L'identificateur du verrou est placé dans la variable pointée par `mutex`.

`pthread_mutex_destroy` détruit le verrou. `pthread_mutex_lock` tente de le bloquer (et met le thread en attente si le verrou est déjà bloqué), `pthread_mutex_unlock` le débloque. `pthread_mutex_trylock` tente de bloquer le verrou, et échoue si le verrou est déjà bloqué.

14.3 Exemple

Source :

```
 1  /* leger_mutex.c */

 2  #include <pthread.h>
 3  #include <stdio.h>
 4  #include <unistd.h>

 5  struct DonneesTache {
 6      char *chaine;               /* chaine à écrire       */
 7      int nombre;                 /* nombre de répétitions */
 8      int delai;                  /* délai entre écritures */
 9  };

10  int numero = 0;
11  pthread_mutex_t verrou;         /* protêge l'accès à numero */

12  void *executer_tache(void *data)
13  {
14      int k;
15      struct Donnees *d = data;
16      for (k = 0; k < d->nombre; k++) {

17          pthread_mutex_lock(&verrou);        /* DEBUT SECTION CRITIQUE */
18          numero++;
19          printf("[%d] %s\n", numero, d->chaine);
20          pthread_mutex_unlock(&verrou);      /* FIN SECTION CRITIQUE */

21          sleep(d->delai);
22      };
23      return NULL;
```

```
24      }

25      int main(void)
26      {
27          pthread_t t1, t2;
28          struct DonneesTache d1, d2;

29          d1.nombre = 3;
30          d1.chaine = "Hello";
31          d1.delai = 1;

32          d2.nombre = 2;
33          d2.chaine = "World";
34          d2.delai = 2;

35          pthread_mutex_init(&verrou, NULL);

36          pthread_create(&t1, NULL, executer_tache, (void *) &d1);
37          pthread_create(&t2, NULL, executer_tache, (void *) &d2);

38          pthread_join(t1, NULL);
39          pthread_join(t2, NULL);

40          pthread_mutex_destroy(&verrou);

41          printf(" %d lignes.\n", numero);
42          exit(0);
43      }
```

Compilation :

Sous Linux, avec la bibliothèque `linuxthreads` de Xavier Leroy (INRIA), ce programme doit être compilé avec l'option `-D_REENTRANT` et la bibliothèque `libpthread` :

```
gcc -g -Wall -pedantic -D_REENTRANT leger_mutex.c -o leger_mutex -lpthread
```

Exécution :

```
% leger_mutex
[1] Hello
[2] World
[3] Hello
[4] Hello
[5] World
 5 lignes.
%
```

14.4 Sémaphores

Les sémaphores, qui font partie de la norme POSIX, ne sont pas implémentés dans toutes les bibliothèques de threads.

```
#include <semaphore.h>

int sem_init(sem_t *sem, int pshared, unsigned int value);
int sem_destroy(sem_t * sem);
```

```
int sem_wait(sem_t * sem);
int sem_post(sem_t * sem);

int sem_trywait(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
```

Les sémaphores sont créés par `sem_init`, qui place l'identificateur du sémaphore à l'endroit pointé par `sem`. La valeur initiale du sémaphore est dans `value`. Si `pshared` est nul, le sémaphore est local au processus lourd (le partage de sémaphores entre plusieurs processus lourds n'est pas implémenté dans la version courante de `linuxthreads`.).

`sem_wait` et `sem_post` sont les équivalents respectifs des primitives `P` et `V` de Dijkstra. La fonction `sem_trywait` échoue (au lieu de bloquer) si la valeur du sémaphore est nulle. Enfin, `sem_getvalue` consulte la valeur courante du sémaphore.

Exercice : Utiliser un sémaphore au lieu d'un `mutex` pour sécuriser l'exemple.

14.5 Conditions

Les *conditions* servent à mettre en attente des processus légers derrière un `mutex`. Une primitive permet de débloquer d'un seul coup tous les threads bloqués par un même condition.

```
#include <pthread.h>

int pthread_cond_init(pthread_cond_t  *cond,
                      pthread_condattr_t *cond_attr);
int pthread_cond_destroy(pthread_cond_t *cond);

int pthread_cond_wait(pthread_cond_t *cond,
                      pthread_mutex_t *mutex);

int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
```

Les conditions sont créées par `phtread_cond_init`, et détruites par `phtread_cond_destroy`.

Un processus se met en attente en effectuant un `phtread_cond_wait` (ce qui bloque au passage un `mutex`). La primitive `phtread_cond_broadcast` débloque tous les processus qui attendent sur une condition, `phtread_cond_signal` en débloque un seul.

15 Communication entre processus (IPC System V)

Les mécanismes de communication entre processus (*InterProcess Communication*, ou *IPC*) d'Unix System V ont été repris dans de nombreuses variantes d'Unix. Il y a 3 mécanismes :
– les segments de mémoire partagée,
– les sémaphores,
– les files de messages.
Ces trois types d'objets sont identifiés par des *clés*.

15.1 `ftok()` constitution d'une clé

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

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

key_t ftok ( char *pathname, char project )
```

La fonction `ftok()` constitue une clé à partir d'un chemin d'accès et d'un caractère indiquant un « projet ». Plutôt que de risquer une explication abstraite, étudions deux cas fréquents :

- On dispose d'un logiciel commun dans `/opt/jeux/OuiOui`. Ce logiciel utilise deux objets partagés. On pourra utiliser les clés `ftok("/opt/jeux/OuiOui",'A')` et `ftok("/opt/jeux/OuiOui",'B')`. Ainsi tous les processus de ce logiciel se réfèreront aux mêmes objets qui seront partagés entre tous les utilisateurs.
- On distribue un exemple aux étudiants, qui le recopient chez eux et le font tourner. On souhaite que les processus d'un même étudiant communiquent entre eux, mais qu'ils n'interfèrent pas avec d'autres. On basera donc la clé sur une donnée personnelle, par exemple le répertoire d'accueil, avec les clés `ftok(getenv("HOME"),'A')` et `ftok(getenv("HOME"),'B')`.

15.2 Mémoires partagées

Ce mécanisme permet à plusieurs programmes de partager des *segments mémoire*. Chaque segment mémoire est identifié, au niveau du système, par une `clé` à laquelle correspond un *identifiant*. Lorsqu'un segment est *attaché* à un programme, les données qu'il contient sont accessibles en mémoire par l'intermédiaire d'un pointeur.

```
#include <sys/ipc.h>
#include <sys/shm.h>

int shmget(key_t key, int size, int shmflg);
char *shmat (int shmid, char *shmaddr, int shmflg )
int shmdt (char *shmaddr)
int shmctl(int shmid, int cmd, struct shmid_ds *buf);
```

La fonction `shmget()` donne l'identifiant du segment ayant la clé `key`. Un nouveau segment (de taille `size`) est créé si `key` est `IPC_PRIVATE`, ou bien si les indicateurs de `shmflg` contiennent `IPC_CREAT`. Combinées, les options `IPC_EXCL | IPC_CREAT` indiquent que le segment ne doit pas exister préalablement. Les bits de poids faible de `shmflg` indiquent les droits d'accès.

`shmat()` attache le segment `shmid` en mémoire, avec les droits spécifiés dans `shmflag` (`SHM_R, SHM_W, SHM_RDONLY`). `shmaddr` précise où ce segment doit être situé dans l'espace mémoire (la valeur `NULL` demande un placement automatique). `shmat()` renvoie l'adresse où le segment a été placé.

`shmdt()` "libère" le segment. `shmctl()` permet diverses opérations, dont la destruction d'une mémoire partagée (voir exemple).

Exemple (deux programmes) :

Le producteur :

```
1  /* prod.c */

2  /*
3     Ce programme lit une suite de nombres, et effectue le cumul dans une
4     variable en mémoire partagée. */

5  #include <sys/ipc.h>
6  #include <sys/shm.h>
7  #include <sys/types.h>
8  #include <stdlib.h>
9  #include <stdio.h>
```

```
10  #include <errno.h>

11  void abandon(char message[])
12  {
13      perror(message);
14      exit(EXIT_FAILURE);
15  }

16  struct donnees {
17      int nb;
18      int total;
19  };

20  int main(void)
21  {
22      key_t cle;
23      int id;
24      struct donnees *commun;
25      int reponse;

26      cle = ftok(getenv("HOME"), 'A');
27      if (cle == -1)
28          abandon("ftok");

29      id = shmget(cle, sizeof(struct donnees),
30                  IPC_CREAT | IPC_EXCL | 0666);
31      if (id == -1) {
32          switch (errno) {
33          case EEXIST:
34              abandon("Note: le segment existe déjà\n");
35          default:
36              abandon("shmget");
37          }
38      }
39      commun = (struct donnees *) shmat(id, NULL, SHM_R | SHM_W);
40      if (commun == NULL)
41          abandon("shmat");

42      commun->nb = 0;
43      commun->total = 0;

44      while (1) {
45          printf("+ ");
46          if (scanf("%d", &reponse) != 1)
47              break;
48          commun->nb++;
49          commun->total += reponse;
50          printf("sous-total %d= %d\n", commun->nb, commun->total);
51      }
52      printf("---\n");

53      if (shmdt((char *) commun) == -1)
54          abandon("shmdt");

55      /* suppression segment */
```

```
56          if (shmctl(id, IPC_RMID, NULL) == -1)
57              abandon("shmctl(remove)");

58          return EXIT_SUCCESS;
59      }
```

Le consommateur :

```
 1      /* cons.c */

 2      /*
 3         Ce programme affiche périodiquement le contenu de la
 4         mémoire partagée. Arrêt par Contrôle-C
 5      */

 6      #include <sys/ipc.h>
 7      #include <sys/shm.h>
 8      #include <sys/types.h>
 9      #include <unistd.h>
10      #include <stdlib.h>
11      #include <stdio.h>
12      #include <errno.h>
13      #include <signal.h>

14      #define DELAI 2

15      void abandon(char message[])
16      {
17          perror(message);
18          exit(EXIT_FAILURE);
19      }

20      struct donnees {
21          int nb;
22          int total;
23      };

24      int continuer_boucle = 1;

25      void arreter_boucle(int signal)
26      {
27          continuer_boucle = 0;
28      }

29      int main(void)
30      {
31          key_t cle;
32          int id;
33          struct donnees *commun;
34          struct sigaction a;

35          cle = ftok(getenv("HOME"), 'A');
36          if (cle == -1)
37              abandon("ftok");

38          id = shmget(cle, sizeof(struct donnees), 0);
39          if (id == -1) {
```

```
40              switch (errno) {
41              case ENOENT:
42                  abandon("pas de segment\n");
43              default:
44                  abandon("shmget");
45              }
46          }

47          commun = (struct donnees *) shmat(id, NULL, SHM_R);
48          if (commun == NULL)
49              abandon("shmat");

50          continuer_boucle = 1;

51          a.sa_handler = arreter_boucle;
52          sigemptyset(&a.sa_mask);
53          a.sa_flags = 0;
54          sigaction(SIGINT, &a, NULL);

55          while (continuer_boucle) {
56              sleep(DELAI);
57              printf("sous-total %d= %d\n", commun->nb, commun->total);
58          }

59          printf("---\n");
60          if (shmdt((char *) commun) == -1)
61              abandon("shmdt");

62          return EXIT_SUCCESS;
63      }
```

Question : le second programme n'affiche pas forcément des informations cohérentes. Pourquoi ? Qu'y faire ?

Problème : écrire deux programmes qui partagent deux variables `i, j`. Voici le pseudo-code :

```
processus P1                    processus P2
| i=0 j=0                       | tant que i==j
| repeter indefiniment          |   faire rien
|   i++ j++                     | ecrire i
fin                             fin
```

Au bout de combien de temps le processus P2 s'arrête-t-il ? Faire plusieurs essais.

Exercice : la commande `ipcs` affiche des informations sur les segments qui existent. Ecrire une commande qui permet d'afficher le contenu d'un segment (on donne le *shmid* et la longueur en paramètres).

15.3 Sémaphores

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>

int semget(key_t key, int nsems, int semflg )
int semop(int semid, struct sembuf *sops, unsigned nsops)
int semctl(int semid, int semnum, int cmd, union semun arg )
```

Les opérations System V travaillent en fait sur des tableaux de sémaphores généralisés (pouvant évoluer par une valeur entière quelconque).

La fonction `semget()` demande à travailler sur le sémaphore généralisé qui est identifié par la clé `key` (même notion que pour les clés des segments partagés) et qui contient `nsems` sémaphores individuels. Un nouveau sémaphore est créé, avec les droits donnés par les 9 bits de poids faible de `semflg`, si `key` est `IPC_PRIVATE`, ou si `semflg` contient `IPC_CREAT`.

`semop()` agit sur le sémaphore `semid` en appliquant simultanément à plusieurs sémaphores individuels les actions décrites dans les `nsops` premiers éléments du tableau `sops`. Chaque `sembuf` est une structure de la forme :

```
struct sembuf
{
  ...
  short sem_num;  /* semaphore number: 0 = first */
  short sem_op;   /* semaphore operation */
  short sem_flg;  /* operation flags */
  ...
}
```

`sem_flg` est une combinaison d'indicateur qui peut contenir `IPC_NOWAIT` et `SEM_UNDO` (voir manuel). Ici nous supposons que `sem_flg` est 0.

`sem_num` indique le numéro du sémaphore individuel sur lequel porte l'opération. `sem_op` est un entier destiné (sauf si il est nul) à être ajouté à la valeur courante *semval* du sémaphore. L'opération se bloque si `sem_op` + `semval` < 0.

Cas particulier : si *sem_ op* est 0, l'opération est bloquée tant que `semval` est non nul.

Les valeurs des sémaphores ne sont mises à jour que lorsque aucun d'eux n'est bloqué.

`semctl` permet de réaliser diverses opérations sur les sémaphores, selon la commande demandée. En particulier, on peut fixer le `n`-ième sémaphore à la valeur `val` en faisant :

```
semctl(sem,n,SETVAL,val);
```

Exemple : primitives sur les sémaphores traditionnels.

```
 1  /* sem.c */

 2  /*
 3     Opérations sur des sémaphores
 4  */

 5  #include <sys/types.h>
 6  #include <sys/ipc.h>
 7  #include <sys/sem.h>
 8  #include <unistd.h>
 9  #include <stdio.h>
10  #include <stdlib.h>
11  #include <ctype.h>

12  typedef int SEMAPHORE;

13  void abandon(char message[])
14  {
15      perror(message);
16      exit(EXIT_FAILURE);
17  }

18  void detruire_sem(SEMAPHORE sem)
```

```
19  {
20      if (semctl(sem, 0, IPC_RMID, 0) != 0)
21          abandon("detruire_sem");
22  }

23  void changer_sem(SEMAPHORE sem, int val)
24  {
25      struct sembuf sb[1];
26      sb[0].sem_num = 0;
27      sb[0].sem_op = val;
28      sb[0].sem_flg = 0;
29      if (semop(sem, sb, 1) != 0)
30          abandon("changer_sem");
31  }

32  SEMAPHORE creer_sem(key_t key)
33  {
34      SEMAPHORE sem;
35      int r;

36      sem = semget(key, 1, IPC_CREAT | 0666);
37      if (sem < 0) {
38          perror("creer_sem");
39          exit(EXIT_FAILURE);
40      }
41      r = semctl(sem, 0, SETVAL, 0);      /* valeur initiale = 0 */
42      if (r < 0) {
43          perror("initialisation sémaphore");
44          exit(EXIT_FAILURE);
45      }
46      return sem;
47  }

48  void P(SEMAPHORE sem)
49  {
50      changer_sem(sem, -1);
51  }

52  void V(SEMAPHORE sem)
53  {
54      changer_sem(sem, 1);
55  }

56  /* ------------------------------------------- */

57  int main(int argc, char *argv[])
58  {
59      SEMAPHORE sem;
60      key_t key;
61      int encore = 1;

62      if (argc != 2) {
63          fprintf(stderr, "Usage: %s cle\n", argv[0]);
64          abandon("Mauvais nombre de paramètres");
65      }
```

```
66          key = atoi(argv[1]);
67          sem = creer_sem(key);
68          while (encore) {
69              char reponse;
70              printf("p,v,x,q ? ");
71              if (scanf("%c", &reponse) != 1)
72                  break;
73              switch (toupper(reponse)) {
74              case 'P':
75                  P(sem);
76                  printf("OK.\n");
77                  break;
78              case 'V':
79                  V(sem);
80                  printf("OK.\n");
81                  break;
82              case 'X':
83                  detruire_sem(sem);
84                  printf("Sémaphore détruit\n");
85                  encore = 0;
86                  break;
87              case 'Q':
88                  encore = 0;
89                  break;
90              default:
91                  printf("?\n");
92              }
93          }
94          printf("Bye.\n");
95          return EXIT_SUCCESS;
96      }
```

Exercice : que se passe-t-il si on essaie d'interrompre `semop()` ?

Exercice : utilisez les sémaphores pour "sécuriser" l'exemple présenté sur les mémoires partagées.

15.4 Files de messages

Ce mécanisme permet l'échange de messages par des processus. Chaque message possède un *corps* de longueur variable, et un *type* (entier strictement positif) qui peut servir à préciser la nature des informations contenues dans le corps.

Au moment de la réception, on peut choisir de sélectionner les messages d'un type donné.

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>

int msgget (key_t key, int msgflg)
int msgsnd (int msqid, struct msgbuf *msgp, int msgsz, int msgflg)
int msgrcv (int msqid, struct msgbuf *msgp, int msgsz,
                       long msgtyp, int msgflg)
int msgctl ( int msqid, int  cmd, struct msqid_ds *buf )
```

`msgget()` demande l'accès à (ou la création de) la file de message avec la clé `key`. `msgget()` retourne la valeur de l'*identificateur de file*.

`msgsnd()` envoie un message dans la file `msqid`. Le *corps* de ce message contient `msgsz` octets, il est placé, précédé par le *type* dans le tampon pointé par `msgp`. Ce tampon de la forme :

```
struct msgbuf {
     long mtype;     /* message type, must be > 0 */
     char mtext[...] /* message data */
};
```

`msgrcv()` lit dans la file un message d'un type donné (si `type` $>$ `0`) ou indifférent (si `type==0`), et le place dans le tampon pointé par `msgp`. La taille du corps ne pourra excéder `msgsz` octets, sinon il sera tronqué. `msgrcv()` renvoie la taille du corps du message.

Exemple. Deux programmes, l'un pour envoyer des messages (lignes de texte) sur une file avec un type donné, l'autre pour afficher les messages reçus.

```
 1  /* snd.c */

 2  /*
 3     envoi des messages dans une file (IPC System V)
 4  */

 5  #include <errno.h>
 6  #include <stdio.h>
 7  #include <stdlib.h>
 8  #include <stdio.h>

 9  #include <sys/types.h>
10  #include <sys/ipc.h>
11  #include <sys/msg.h>

12  #define MAX_TEXTE 1000

13  struct tampon {
14      long mtype;
15      char mtext[MAX_TEXTE];
16  };

17  void abandon(char message[])
18  {
19      perror(message);
20      exit(EXIT_FAILURE);
21  }

22  int main(int argc, char *argv[])
23  {
24      int cle, id, mtype;
25      if (argc != 3) {
26          fprintf(stderr, "Usage: %s clé type\n", argv[0]);
27          abandon("mauvais nombre de paramètres");
28      }
29      cle = atoi(argv[1]);
30      mtype = atoi(argv[2]);
31      id = msgget(cle, 0666);
32      if (id == -1)
33          abandon("msgget");

34      while (1) {
```

```
35              struct tampon msg;
36              int l, r;

37              printf("> ");
38              fgets(msg.mtext, MAX_TEXTE, stdin);
39              l = strlen(msg.mtext);
40              msg.mtype = mtype;
41              r = msgsnd(id, (struct msgbuf *) &msg, l + 1, 0);
42              if (r == -1)
43                  abandon("msgsnd");
44          }
45      }
```

```
 1      /* rcv.c */

 2      /*
 3         affiche les messages qui proviennent
 4         d'une file (IPC System V)
 5      */

 6      #include <errno.h>
 7      #include <stdio.h>
 8      #include <stdlib.h>
 9      #include <stdio.h>

10      #include <sys/types.h>
11      #include <sys/ipc.h>
12      #include <sys/msg.h>

13      #define MAX_TEXTE 1000
14      struct tampon {
15          long mtype;
16          char mtext[MAX_TEXTE];
17      };

18      int continuer_boucle = 1;

19      void abandon(char message[])
20      {
21          perror(message);
22          exit(EXIT_FAILURE);
23      }

24      int main(int argc, char *argv[])
25      {
26          int cle, id;
27          if (argc != 2) {
28              fprintf(stderr, "Usage: %s cle\n", argv[0]);
29              abandon("Mauvais nombnre de paramètres");
30          }
31          cle = atoi(argv[1]);
32          id = msgget(cle, IPC_CREAT | 0666);
33          if (id == -1)
34              abandon("msgget");

35          while (continuer_boucle) {
```

```
36              struct tampon msg;
37              int l = msgrcv(id, (struct msgbuf *) &msg, MAX_TEXTE, 0L, 0);
38              if (l == -1)
39                  abandon("msgrcv");
40              printf("(type=%ld) %s\n", msg.mtype, msg.mtext);
41          }

42          return EXIT_SUCCESS;
43      }
```

16 Communication par le réseau TCP-IP

Les concepts fondamentaux (sockets, adresses, communication par datagrammes et flots, client-serveur etc.) sont les mêmes que pour les sockets locaux (voir 10 (sockets)). Les particularités viennent des adresses : comment fabriquer une adresse à partir d'un nom de machine (résolution) et d'un numéro de port, comment retrouver le nom d'une machine à partir d'une adresse (résolution inverse) etc.

16.1 Sockets, addresses

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

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

Pour communiquer, les applications doivent créer des *sockets* (prises bidirectionnelles) par la fonction `socket()` et les relier entre elles. On peut ensuite utiliser ces sockets comme des fichiers ordinaires (par `read`, `write`, ...) ou par des opérations spécifiques (`send`, `sendto`, `recv`, `recvfrom`, ...).

Pour désigner un socket sur une machine il faut une *adresse de socket*. Comme il existe différents types de sockets, les opérations sur les adresses concerne un type d'adresse général abstrait (`struct sockaddr`) qui recouvre tous les types concrets particuliers.

Pour TCP-IP (Inet), les adresses de sockets sont déterminées par un numéro IP, et un numéro de port. Pour IPv4, on utilise des `struct sockaddr_in`, qui possèdent 3 champs importants :
– `sin_family`, la famille d'adresses, valant `AF_INET`
– `sin_addr`, pour l'adresse IP.
– `sin_port`, pour le numéro de port
Attention, les octets de l'adresse IP et le numéro de port sont stockés dans *l'ordre réseau* (big-endian), qui n'est pas forcément celui de la machine hôte sur laquelle s'exécute le programme. Voir plus loin les fonctions de conversion hôte/réseau.

Pour IPv6, on utilise des `struct sockaddr_in6`, avec
– `sin6_family`, valant `AF_INET6`
– `sin6_addr`, l'adresse IP sur 6 octets
– `sin6_port`, pour le numéro de port.

16.2 Remplissage d'une adresse

Comme pour les sockets locaux, on crée les sockets par `socket()` et on « nomme la prise » par `bind()`.

16.2.1 Préparation d'une adresse distante

Dans le cas le plus fréquent, on dispose du nom de la machine destinataire (par exemple la chaîne de caractères `"www.elysee.fr"`), et du numéro de port (un entier).

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
```

`gethostbyname()` retourne un pointeur vers une structure `hostent` qui contient diverses informations sur la machine en question, en particulier une adresse `h_addr` [10] que l'on mettra dans `sin_addr`.

Le champ `sin_port` est un entier court *en ordre réseau*, pour y mettre un entier ordinaire il faut le convertir par `htons()`[11] .

Voir exemple `client-echo.c`.

16.2.2 Préparation d'une adresse locale

Pour une adresse sur la machine locale[12], on utilise
- dans le cas le plus fréquent, l'adresse `INADDR_ANY` (`0.0.0.0`). Le socket est alors ouvert (avec le même numéro de port) sur toutes les adresses IP de toutes les interfaces de la machine.
- l'adresse `INADDR_LOOPBACK` correspondant à l'adresse locale `127.0.0.1` (alias `localhost`). Le socket n'est alors accessible que depuis la machine elle-même.
- une des adresses IP de la la machine.
- l'adresse de diffusion générale (*broadcast*) `INADDR_BROADCAST` (`255.255.255.255`)[13]

Voir exemple `serveur-echo.c`.

Pour IPV6
- Famille d'adresses `AF_INET6`, famille de protocoles `PF_INET6`
- adresses prédéfinies `IN6ADD_LOOPBACK_INIT` (`: :1`) `IN6ADD_ANY_INIT`

16.2.3 Examen d'une adresse

La fonction `getsockname()` permet de retrouver l'adresse associée à un socket.

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

int   getsockname(int s,
                  struct sockaddr  *name,
                  socklen_t * namelen )
```

Le numéro de port est dans le champ `sin_port`, pour le convertir en entier normal, utiliser `ntohs()` (Network TO Host short).

L'adresse IP peut être convertie en chaîne de caractères en notation décimale pointée (exemple `"147.210.94.194"`) par `inet_ntoa()` (network to ascii),

```
#include <sys/socket.h>
#include <netinet/in.h>
#include <arpa/inet.h>

char *inet_ntoa(struct in_addr in);
```

[10] Une machine peut avoir plusieurs adresses
[11] Host TO Network Short
[12] ne pas confondre avec la notion de *socket du domaine local* vue en 10 (sockets)
[13] L'utilisation de l'adresse de diffusion est soumise à restrictions, voir manuel

C'est ce qui est utilisé dans `serveur-echo.c` pour déterminer et afficher la provenance des requêtes.

On peut également tenter une *résolution inverse*, c'est-à-dire de retrouver le nom à partir de l'adresse, en passant par `gethostbyaddr`, qui retourne un pointeur vers une structure `hostent`, dont le champ `h_name` désigne le nom officiel de la machine.

Cette résolution n'aboutit pas toujours, parce que tous les numéros IP ne correspondent pas à des machines déclarées.

```
#include <netdb.h>
extern int h_errno;

struct hostent *gethostbyaddr(const char *addr, int len, int type);

struct hostent {
  char    *h_name;        /* official name of host */
  char    **h_aliases;    /* alias list */
  int     h_addrtype;     /* host address type */
  int     h_length;       /* length of address */
  char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */
```

16.3 Fermeture d'un socket

Un socket peut être fermé par `close()` ou par `shutdown()`.

```
int shutdown(int fd, int how);
```

Un socket est bidirectionnel, le paramètre `how` indique quelle(s) moitié(s) on ferme : `0` pour la sortie, `1` pour l'entrée, `2` pour les deux (équivaut à `close()`).

16.4 Communication par datagrammes (UDP)

16.4.1 Création d'un socket

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

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

Cette fonction construit un socket et retourne un numéro de descripteur. Pour une liaison par datagrammes sur IPv4, utilisez la famille `AF_INET`, le type `SOCK_DGRAM` et le protocole par défaut `0`.

Retourne `-1` en cas d'échec.

16.4.2 Connexion de sockets

La fonction `connect` met en relation un socket (de cette machine) avec un autre socket désigné, qui sera le « correspondant par défaut » pour la suite des opérations.

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

int  connect(int sockfd,
```

```
                const struct sockaddr *serv_addr,
                socklen_t addrlen);
```

16.4.3 Envoi de datagrammes

Sur un socket connecté (voir ci-dessus), on peut expédier des datagrammes (contenus dans un tampon `t` de longueur `n`) par `write(sockfd,t,n)`.

La fonction `send()`

```
int send(int s, const void *msg, size_t len, int flags);
```

permet d'indiquer des *flags*, par exemple `MSG_DONTWAIT` pour une écriture non bloquante.

Enfin, `sendto()` envoie un datagramme à une adresse spécifiée, sur un socket connecté ou non.

```
int sendto(int s, const void *msg, size_t len, int flags,
           const struct sockaddr *to, socklen_t tolen);
```

16.4.4 Réception de datagrammes

Inversement, la réception peut se faire par un simple `read()`, par un `recv()` (avec des flags), ou par un `recvfrom`, qui permet de récupérer l'adresse `from` de l'émetteur.

```
int recv(int s, void *buf, size_t len, int flags);
int recvfrom(int  s,  void  *buf,  size_t len, int flags,
             struct sockaddr *from, socklen_t *fromlen);
```

16.4.5 Exemple UDP : serveur d'écho

Principe :
- Le client envoie une chaîne de caractères au serveur.
- Le serveur l'affiche, la convertit en minuscules, et la réexpédie.
- Le client affiche la réponse.

Usage :
- sur le serveur : `serveur-echo` *numéro-de-port*
- pour chaque client : `client-echo` *nom-serveur numéro-de-port « message à expédier »*

Le client

```
 1  /*
 2    client-echo.c

 3    Envoi de datagrammes

 4    Exemple de client qui
 5    - ouvre un socket
 6    - envoie des datagrammes sur ce socket (lignes de textes
 7    de l'entrée standard)
 8    - attend une réponse
 9    - affiche la réponse
10    */

11  #include <unistd.h>
12  #include <sys/types.h>
13  #include <sys/socket.h>
```

```
14  #include <netinet/in.h>
15  #include <signal.h>
16  #include <stdio.h>
17  #include <netdb.h>
18  #include <string.h>
19  #include <stdlib.h>

20  #define TAILLE_TAMPON 1000
21  static int fd;

22  void abandon(char message[])
23  {
24      printf("CLIENT> Erreur fatale\n");
25      perror(message);
26      exit(EXIT_FAILURE);
27  }

28  int main(int argc, char *argv[])
29  {
30      struct sockaddr_in adresse_socket_serveur;
31      struct hostent *hote;
32      int taille_adresse_socket_serveur;
33      char *nom_serveur;
34      int numero_port_serveur;
35      char *requete, reponse[TAILLE_TAMPON];
36      int longueur_requete, longueur_reponse;

37      /* 1. réception des paramètres de la ligne de commande */
38      if (argc != 4) {
39          printf("Usage: %s hote port message\n", argv[0]);
40          abandon("nombre de paramètres incorrect");
41      }
42      nom_serveur = argv[1];
43      numero_port_serveur = atoi(argv[2]);
44      requete = argv[3];

45      /* 2. Initialisation du socket */

46      /* 2.1 création du socket en mode datagramme */
47      fd = socket(AF_INET, SOCK_DGRAM, 0);
48      if (fd < 0)
49          abandon("Creation socket");

50      /* 2.2 recherche de la machine serveur */

51      hote = gethostbyname(nom_serveur);
52      if (hote == NULL)
53          abandon("Recherche serveur");

54      /* 2.3 Remplissage adresse serveur */
55      adresse_socket_serveur.sin_family = AF_INET;
56      adresse_socket_serveur.sin_port = htons(numero_port_serveur);
57      adresse_socket_serveur.sin_addr =
58          *(struct in_addr *) hote->h_addr;
59      taille_adresse_socket_serveur = sizeof adresse_socket_serveur;
```

```
60          /* 3. Envoi de la requête */
61          printf("REQUETE> %s\n", requete);
62          longueur_requete = strlen(requete) + 1;
63          if (sendto(fd, requete, longueur_requete, 0,        /* flags */
64                     (struct sockaddr *) &adresse_socket_serveur,
65                     taille_adresse_socket_serveur)
66              < 0)
67              abandon("Envoi requete");

68          /* 4. Lecture de la réponse */
69          longueur_reponse =
70              recvfrom(fd, reponse, TAILLE_TAMPON, 0, NULL, 0);
71          if (longueur_reponse < 0)
72              abandon("Attente réponse");
73          printf("REPONSE> %s\n", reponse);
74          close(fd);

75          printf("CLIENT> Fin.\n");
76          return EXIT_SUCCESS;
77      }
```

Exercice : faire en sorte que le client réexpédie sa requête si il ne reçoit pas la réponse dans un délai fixé. Fixer une limite au nombre de tentatives.

Le serveur

```
1       /*
2         serveur-echo.c -  Réception de datagrammes

3         Exemple de serveur qui
4         - ouvre un socket sur un port en mode non-connecté
5         - affiche les messages (chaînes de caractères)
6         qu'il reçoit par ce socket.
7         - envoie une réponse
8         */

9       #include <unistd.h>
10      #include <sys/types.h>
11      #include <sys/socket.h>
12      #include <netinet/in.h>
13      #include <signal.h>
14      #include <stdio.h>
15      #include <stdlib.h>
16      #include <arpa/inet.h>
17      #include <ctype.h>
18      #include <string.h>

19      #define TAILLE_TAMPON 1000
20      static int fd;

21      void abandon(char message[])
22      {
23          printf("SERVEUR> Erreur fatale\n");
24          perror(message);
25          exit(EXIT_FAILURE);
```

```
26  }

27  void arreter_serveur(int signal)
28  {
29      close(fd);
30      printf("SERVEUR> Arrêt du serveur (signal %d)\n", signal);
31      exit(EXIT_SUCCESS);
32  }

33  int main(int argc, char *argv[])
34  {
35      struct sockaddr_in adresse_serveur;
36      size_t taille_adresse_serveur;
37      int numero_port_serveur;
38      char *src, *dst;
39      struct sigaction a;

40      /* 1. réception des paramètres de la ligne de commande */

41      if (argc != 2) {
42          printf("usage: %s port\n", argv[0]);
43          abandon("mauvais nombre de paramètres");
44      }
45      numero_port_serveur = atoi(argv[1]);

46      /* 2. Si une interruption se produit, arrêt du serveur */
47      /* signal(SIGINT, arreter_serveur); */

48      a.sa_handler = arreter_serveur;
49      sigemptyset(&a.sa_mask);
50      a.sa_flags = 0;
51      sigaction(SIGINT, &a, NULL);

52      /* 3. Initialisation du socket de réception */

53      /* 3.1 Création du socket en mode non-connecté
54         (datagrammes) */

55      fd = socket(AF_INET, SOCK_DGRAM, 0);
56      if (fd < 0)
57          abandon("socket");

58      /* 3.2 Remplissage de l'adresse de réception
59         (protocole Internet TCP-IP, réception acceptée sur toutes
60         les adresses IP du serveur, numéro de port indiqué)
61       */

62      adresse_serveur.sin_family = AF_INET;
63      adresse_serveur.sin_addr.s_addr = INADDR_ANY;
64      adresse_serveur.sin_port = htons(numero_port_serveur);

65      /* 3.3 Association du socket au port de réception */

66      taille_adresse_serveur = sizeof adresse_serveur;
67      if (bind(fd,
```

```
68                  (struct sockaddr *) &adresse_serveur,
69                  taille_adresse_serveur) < 0)
70              abandon("bind");

71          printf("SERVEUR> Le serveur écoute le port %d\n",
72                 numero_port_serveur);

73          while (1) {
74              struct sockaddr_in adresse_client;
75              int taille_adresse_client;
76              char tampon_requete[TAILLE_TAMPON],
77                  tampon_reponse[TAILLE_TAMPON];
78              int lg_requete, lg_reponse;

79              /* 4. Attente d'un datagramme (requête) */

80              taille_adresse_client = sizeof(adresse_client);
81              lg_requete = recvfrom(fd, tampon_requete, TAILLE_TAMPON, 0,     /* flags */
82                                    (struct sockaddr *) &adresse_client,
83                                    (socklen_t *) & taille_adresse_client);
84              if (lg_requete < 0)
85                  abandon("recfrom");

86              /* 5. Affichage message avec sa provenance et sa longueur */

87              printf("%s:%d [%d]\t: %s\n",
88                     inet_ntoa(adresse_client.sin_addr),
89                     ntohs(adresse_client.sin_port), lg_requete,
90                     tampon_requete);

91              /* 6. Fabrication d'une réponse */

92              src = tampon_requete;
93              dst = tampon_reponse;
94              while ((*dst++ = toupper(*src++)) != '\0');
95              lg_reponse = strlen(tampon_reponse) + 1;

96              /* 7. Envoi de la réponse */

97              if (sendto(fd,
98                         tampon_reponse,
99                         lg_reponse,
100                        0,
101                        (struct sockaddr *) &adresse_client,
102                        taille_adresse_client)
103                 < 0)
104                 abandon("Envoi de la réponse");
105         }
106         /* on ne passe jamais ici */
107         return EXIT_SUCCESS;
108     }
```

16.5 Communication par flots de données (TCP)

La création d'un socket pour TCP se fait ainsi

```
int fd;
..
fd=socket(AF_INET,SOCK_STREAM,0);
```

16.5.1 Programmation des clients TCP

Le socket d'un client TCP doit être relié (par `connect()`) à celui du serveur, et il est utilisé ensuite par des `read()` et des `write()`, ou des entrées-sorties de haut niveau `fprintf()`, `fscanf()`, etc. si on a défini des flots par `fdopen()`.

16.5.2 Exemple : client web

```
1   /* client-web.c */

2   /*
3      Interrogation d'un serveur web

4      Usage:
5      client-web serveur port adresse-document

6      retourne le contenu du document d'adresse
7      http://serveur:port/adresse-document

8      Exemple:
9      client-web www.info.prive 80 /index.html

10     Fonctionnement:
11     - ouverture d'une connexion TCP vers serveur:port
12     - envoi de la requête   GET adresse-document HTTP/1.0[cr][lf][cr][lf]
13     - affichage de la réponse
14  */

15  #include <unistd.h>
16  #include <sys/types.h>
17  #include <sys/socket.h>
18  #include <netinet/in.h>
19  #include <signal.h>
20  #include <stdio.h>
21  #include <netdb.h>
22  #include <string.h>
23  #include <stdlib.h>

24  #define CRLF "\r\n"
25  #define TAILLE_TAMPON 1000

26  void abandon(char message[])
27  {
28      perror(message);
29      exit(EXIT_FAILURE);
30  }

31  /* -- connexion vers un serveur TCP -------------------------- */
32  int ouvrir_connexion_tcp(char nom_serveur[], int port_serveur)
33  {
```

```
34          struct sockaddr_in addr_serveur;
35          struct hostent *serveur;
36          int fd;

37          fd = socket(AF_INET, SOCK_STREAM, 0);       /* création prise */
38          if (fd < 0)
39              abandon("socket");

40          serveur = gethostbyname(nom_serveur);       /* recherche adresse serveur */
41          if (serveur == NULL)
42              abandon("gethostbyname");

43          addr_serveur.sin_family = AF_INET;
44          addr_serveur.sin_port = htons(port_serveur);
45          addr_serveur.sin_addr = *(struct in_addr *) serveur->h_addr;

46          if (connect(fd,             /* connexion au serveur */
47                      (struct sockaddr *) &addr_serveur,
48                      sizeof addr_serveur)
49              < 0)
50              abandon("connect");

51          return (fd);
52      }

53      /* ------------------------------------------------------ */

54      void demander_document(int fd, char adresse_document[])
55      {
56          char requete[TAILLE_TAMPON];
57          int longueur;
58          /* constitution de la requête, suivie d'une ligne vide */
59          longueur = snprintf(requete, TAILLE_TAMPON,
60                              "GET %s HTTP/1.0" CRLF CRLF,
61                              adresse_document);
62          write(fd, requete, longueur);       /* envoi */
63      }

64      /* ------------------------------------------------------ */

65      void afficher_reponse(int fd)
66      {
67          char tampon[TAILLE_TAMPON];
68          int longueur;
69          while (1) {
70              longueur = read(fd, tampon, TAILLE_TAMPON);     /* lecture par bloc */
71              if (longueur <= 0)
72                  break;
73              write(1, tampon, longueur);     /* copie sur sortie standard */
74          };
75      }

76      /* -- main ------------------------------------------------ */

77      int main(int argc, char *argv[])
```

```
78      {
79          char *nom_serveur, *adresse_document;
80          int port_serveur;
81          int fd;

82          if (argc != 4) {
83              printf("Usage: %s serveur port adresse-document\n", argv[0]);
84              abandon("nombre de paramètres incorrect");
85          }

86          nom_serveur = argv[1];
87          port_serveur = atoi(argv[2]);
88          adresse_document = argv[3];

89          fd = ouvrir_connexion_tcp(nom_serveur, port_serveur);

90          demander_document(fd, adresse_document);
91          afficher_reponse(fd);
92          close(fd);

93          return EXIT_SUCCESS;
94      }
```

Remarque : souvent il est plus commode de créer des flots de haut niveau au dessus du socket (voir `fdopen()`) que de manipuler des `read` et des `write`. Voir dans l'exemple suivant.

16.5.3 Réaliser un serveur TCP

Un serveur TCP doit traiter des connexions venant de plusieurs clients.

Après avoir créé et nommé le socket, le serveur spécifie qu'il accepte les communications entrantes par `listen()`, et se met effectivement en attente d'une connexion de client par `accept()`.

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

int listen(int s, int backlog);
int accept(int s,  struct  sockaddr  *addr,
                   socklen_t  *addrlen);
```

Le paramètre `backlog` indique la taille maximale de la file des connexions en attente. Sous Linux la limite est donnée par la constante `SOMAXCONN` (qui vaut 128), sur d'autres systèmes elle est limitée à 5.

La fonction `accept()` renvoie un autre socket, qui servira à la communication avec le client. L'adresse du client peut être obtenue par les paramètres `addr` et `addrlen`.

En général les serveurs TCP doivent traiter plusieurs connexions simultanément. La solution habituelle est de lancer, après l'appel à `accept()` un processus fils (par `fork()`)qui traite la communication avec un seul client. Ceci induit une gestion des processus, donc des signaux liés à la terminaison des processus fils.

17 Exemples TCP : serveurs Web

Dans ce qui suit nous présentons un serveur Web rudimentaire, capable de fournir des pages Web construites à partir des fichiers d'un répertoire. Nous donnons deux implémentations possibles, à l'aide de processus lourds et légers.

17.1 Serveur Web (avec processus)

17.1.1 Principe et psedo-code

Cette version suit l'approche traditionnelle. Un processus est créé chaque fois qu'un client contacte le serveur.

Pseudo-code :

```
ouvrir socket serveur (socket/bind/listen)
répéter indéfiniment
|    attendre l'arrivée d'un client (accept)
|    créer un processus (fork) et lui déléguer
|      la communication avec le client
fin-répeter
```

17.1.2 Code du serveur

```
 1  /* ------------------------------------------------
 2     Serveur TCP

 3     serveur web, qui renvoie les fichiers (textes)
 4     d'un répertoire sous forme de pages HTML

 5    usage :  serveur-web port repertoire
 6    exemple: serveur-web 8000 /usr/doc/exemples
 7  ------------------------------------------------*/

 8  #include <unistd.h>
 9  #include <sys/types.h>
10  #include <sys/errno.h>
11  #include <sys/socket.h>
12  #include <sys/wait.h>
13  #include <sys/stat.h>
14  #include <netinet/in.h>
15  #include <arpa/inet.h>
16  #include <signal.h>
17  #include <stdio.h>
18  #include <stdlib.h>

19  #include "declarations.h"

20  void arreter_serveur(int numero_signal);
21  void attendre_sous_serveur(int numero_signal);

22  int fd_serveur;                /* variable globale, pour partager
23                                    avec traitement signal fin_serveur */

24  void demarrer_serveur(int numero_port, char repertoire[])
25  {
26      int numero_client = 0;
27      int fd_client;
28      struct sigaction action_int, action_chld;

29      fd_serveur = serveur_tcp(numero_port);

30      /* arrêt du serveur si signal SIGINT */
```

```
31          action_int.sa_handler = arreter_serveur;
32          sigemptyset(&action_int.sa_mask);
33          action_int.sa_flags = 0;
34          sigaction(SIGINT, &action_int, NULL);

35          /* attente fils si SIGCHLD */
36          action_chld.sa_handler = attendre_sous_serveur;
37          sigemptyset(&action_chld.sa_mask);
38          action_chld.sa_flags = SA_NOCLDSTOP;
39          sigaction(SIGCHLD, &action_chld, NULL);

40          printf("> Serveur " VERSION
41                 " (port=%d, répertoire de documents=\"%s\")\n",
42                 numero_port, repertoire);

43          while (1) {
44              struct sockaddr_in a;
45              size_t l = sizeof a;

46              fd_client = attendre_client(fd_serveur);
47              getsockname(fd_client, (struct sockaddr *) &a, &l);
48              numero_client++;
49              printf("> client %d [%s]\n", numero_client,
50                     inet_ntoa(a.sin_addr));

51              if (fork() == 0) {
52                  /* le processus fils ferme le socket serveur et s'occupe du client */
53                  close(0);
54                  close(1);
55                  close(2);
56                  close(fd_serveur);
57                  servir_client(fd_client, repertoire);
58                  close(fd_client);
59                  exit(EXIT_SUCCESS);
60              }
61              /* le processus père n'a plus besoin du socket client.
62                 Il le ferme et repart dans la boucle */
63              close(fd_client);
64          }
65      }

66      /* ---------------------------------------------------------------
67       Traitement des signaux
68      ---------------------------------------------------------------- */

69      void arreter_serveur(int numero_signal)
70      {
71          printf("=> fin du serveur\n");
72          shutdown(fd_serveur, 2);    /* utile ? */
73          close(fd_serveur);
74          exit(EXIT_SUCCESS);
75      }

76      void attendre_sous_serveur(int numero_signal)
77      {
```

```
78          /* cette fonction est appelée chaque fois qu'un signal SIGCHLD
79             indique la fin d'un processus fils _au moins_.  */
80          while (waitpid(-1, NULL, WNOHANG) > 0) {
81              /* attente des fils arrêtés, tant qu'il y en a */
82          }
83      }

84      /* ---------------------------------------------------------------*/
85      void usage(char prog[])
86      {
87          printf("Usage : %s [options\n\n", prog);
88          printf("Options :"
89                 "-h\tcemessage\n"
90                 "-p port\tport du serveur          [%d]\n"
91                 "-d dir \trépertoire des documents [%s]\n",
92                 PORT_PAR_DEFAUT, REPERTOIRE_PAR_DEFAUT);
93      }

94      /* ---------------------------------------------------------------*/
95      int main(int argc, char *argv[])
96      {
97          int port = PORT_PAR_DEFAUT;
98          char *repertoire = REPERTOIRE_PAR_DEFAUT;   /* la racine
99                                                         des documents */
100         char c;

101         while ((c = getopt(argc, argv, "hp:d:")) != -1)
102             switch (c) {
103             case 'h':
104                 usage(argv[0]);
105                 exit(EXIT_SUCCESS);
106                 break;
107             case 'p':
108                 port = atoi(optarg);
109                 break;
110             case 'd':
111                 repertoire = optarg;
112                 break;
113             case '?':
114                 fprintf(stderr, "Option inconnue -%c. -h pour aide.\n",
115                         optopt);
116                 break;
117             };
118         demarrer_serveur(port, repertoire);
119         exit(EXIT_SUCCESS);
120     }
```

17.1.3 Discussion de la solution

Un avantage est la simplicité de la solution, et sa robustesse : une erreur d'exécution dans un processus fils est normalement sans incidence sur le fonctionnement des autres parties du serveur.

En revanche, la création d'un processus est une opération relativement coûteuse, qui introduit un temps de latence au début de chaque communication. D'où l'idée de lancer les processus avant d'en avoir besoin (préchargement), et de réutiliser ceux qui ont terminé leur tâche.

17.2 Serveur Web (avec threads)

17.2.1 Principe et pseudo-code

Les processus légers permettent une autre approche : on crée préalablement un « pool » de processus que l'on bloque. Lorsqu'un client se présente, on confie la communication à un processus inoccupé.

```
ouvrir socket serveur (socket/bind/listen)
créer un pool de processus
répéter indéfiniment
|    attendre l'arrivée d'un client (accept)
|    trouver un processus libre, et lui
|        confier la communication avec le client
fin-répeter
```

17.2.2 Code du serveur

```
 1  /* serveur-web.c */

 2  /* ------------------------------------------------
 3     Serveur TCP

 4     serveur web, qui renvoie les fichiers (textes)
 5     d'un répertoire sous forme de pages HTML

 6    usage :  serveur-web port repertoire
 7    exemple: serveur-web 8000 /usr/doc/exemples

 8    Version basée sur les threads. Au lieu de créer
 9    un processus par connexion, on gère un pool de tâches
10    (sous-serveurs).
11    - au démarrage du serveur les sous-serveurs sont créees,
12    et bloqués par un verrou
13    - quand un client se connecte, la connexion est
14    confiée à un sous-serveur inactif, qui est débloqué
15    pour l'occasion.
16  ------------------------------------------------*/

17  #include <pthread.h>
18  #include <unistd.h>
19  #include <sys/types.h>
20  #include <sys/errno.h>
21  #include <sys/socket.h>
22  #include <sys/wait.h>
23  #include <sys/stat.h>
24  #include <netinet/in.h>
25  #include <arpa/inet.h>
26  #include <signal.h>
27  #include <stdio.h>
28  #include <stdlib.h>

29  #include "declarations.h"

30  void arreter_serveur(int numero_signal);
```

```
31  #define NB_SOUS_SERVEURS 5

32  struct donnees_sous_serveur {
33      pthread_t id;               /* identificateur de thread  */
34      pthread_mutex_t verrou;
35      int actif;                  /* 1 => sous-serveur occupé */
36      int fd;                     /* socket du client          */
37      char *repertoire;
38  };

39  struct donnees_sous_serveur pool[NB_SOUS_SERVEURS];
40  int fd_serveur;                 /* variable globale, pour partager
41                                     avec traitement signal fin_serveur */

42  /* ------------------------------------------------------
43     sous_serveur
44     ------------------------------------------------------ */
45  void *executer_sous_serveur(void *data)
46  {
47      struct donnees_sous_serveur *d = data;
48      while (1) {
49          pthread_mutex_lock(&d->verrou);
50          servir_client(d->fd, d->repertoire);
51          close(d->fd);
52          d->actif = 0;
53      }
54      return NULL;                /* jamais exécuté */
55  }

56  /* ------------------------------------------------------ */
57  void creer_sous_serveurs(char repertoire[])
58  {
59      int k;

60      for (k = 0; k < NB_SOUS_SERVEURS; k++) {
61          struct donnees_sous_serveur *d = pool + k;
62          d->actif = 0;
63          d->repertoire = repertoire;
64          pthread_mutex_init(&d->verrou, NULL);
65          pthread_mutex_lock(&d->verrou);
66          pthread_create(&d->id, NULL, executer_sous_serveur,
67                         (void *) d);
68      }
69  }

70  /* ------------------------------------------------------
71     demarrer_serveur: crée le socket serveur
72     et lance des processus pour chaque client
73     ------------------------------------------------------ */

74  int demarrer_serveur(int numero_port, char repertoire[])
75  {
76      int numero_client = 0;
77      int fd_client;
78      struct sigaction action_fin;
```

```
79          printf("> Serveur " VERSION "+threads "
80                 "(port=%d, répertoire de documents=\"%s\")\n",
81                 numero_port, repertoire);

82          /* signal SIGINT -> arrêt du serveur */

83          action_fin.sa_handler = arreter_serveur;
84          sigemptyset(&action_fin.sa_mask);
85          action_fin.sa_flags = 0;
86          sigaction(SIGINT, &action_fin, NULL);

87          /* création du socket serveur et du pool de sous-serveurs */
88          fd_serveur = serveur_tcp(numero_port);
89          creer_sous_serveurs(repertoire);

90          /* boucle du serveur */
91          while (1) {
92              struct sockaddr_in a;
93              size_t l = sizeof a;
94              int k;

95              fd_client = attendre_client(fd_serveur);
96              getsockname(fd_client, (struct sockaddr *) &a, &l);
97              numero_client++;

98              /* recherche d'un sous-serveur inoccupé */
99              for (k = 0; k < NB_SOUS_SERVEURS; k++)
100                 if (pool[k].actif == 0)
101                     break;
102             if (k == NB_SOUS_SERVEURS) {    /* pas de sous-serveur libre ? */
103                 printf("> client %d [%s] rejeté (surcharge)\n",
104                        numero_client, inet_ntoa(a.sin_addr));
105                 close(fd_client);
106             } else {
107                 /* affectation du travail et déblocage du sous-serveur */
108                 printf("> client %d [%s] traité par sous-serveur %d\n",
109                        numero_client, inet_ntoa(a.sin_addr), k);
110                 pool[k].fd = fd_client;
111                 pool[k].actif = 1;
112                 pthread_mutex_unlock(&pool[k].verrou);
113             }
114         }
115     }

116     /* ------------------------------------------------------------
117      Traitement des signaux
118     ------------------------------------------------------------ */

119     void arreter_serveur(int numero_signal)
120     {
121         printf("=> fin du serveur\n");
122         shutdown(fd_serveur, 2);    /* utile ? */
123         close(fd_serveur);
124         exit(EXIT_SUCCESS);
```

```
125     }

126     /* ---------------------------------------------------------------*/
127     void usage(char prog[])
128     {
129         printf("Usage : %s [options\n\n", prog);
130         printf("Options :"
131                "-h\tcemessage\n"
132                "-p port\tport du serveur          [%d]\n"
133                "-d dir \trépertoire des documents [%s]\n",
134                PORT_PAR_DEFAUT, REPERTOIRE_PAR_DEFAUT);
135     }

136     /* ---------------------------------------------------------------*/
137     int main(int argc, char *argv[])
138     {
139         int port = PORT_PAR_DEFAUT;
140         char *repertoire = REPERTOIRE_PAR_DEFAUT;   /* la racine
141                                                        des documents */
142         char c;

143         while ((c = getopt(argc, argv, "hp:d:")) != -1)
144             switch (c) {
145             case 'h':
146                 usage(argv[0]);
147                 exit(EXIT_SUCCESS);
148                 break;
149             case 'p':
150                 port = atoi(optarg);
151                 break;
152             case 'd':
153                 repertoire = optarg;
154                 break;
155             case '?':
156                 fprintf(stderr,
157                         "Option inconnue -%c. -h pour aide.\n", optopt);
158                 break;
159             }
160         demarrer_serveur(port, repertoire);
161         exit(EXIT_SUCCESS);
162     }
```

17.2.3 Discussion de la solution

Cette solution présente les inconvénients symétriques de ses avantages. Les processus légers partageant une grande partie leur espace mémoire, le crash d'un processus léger risque d'emporter le reste du serveur.

On peut utiliser le même mécanisme de « pool de processus » avec des processus classiques. La difficulté technique réside dans la transmission le descripteur résultant de l'`accept()` du serveur vers un processus fils. Dans la première solution (`fork()` pour chaque connexion) le fils est lancé *après* l'`accept()`, et peut donc hériter du descripteur. Dans le cas d'un préchargement de processus fils, ce n'est plus possible.

Pour ce faire, on peut utiliser le mécanisme (certes assez baroque) de transmission des *informations de service* (*Ancillary messages*à travers une liaison par *socket Unix* entre deux processus : des options convenables de `sendmsg()` permettent de faire passer un ensemble de descripteurs de fichiers d'un processus à un autre (qui

tournent sur la même machine, puisqu'ils communiquent par un socket Unix). Voir exemple à la fin de cette section.

17.3 Parties communes aux deux serveurs

17.3.1 Déclarations et entêtes de fonctions

```
1   /* Serveur Web - declarations.h */

2   #define CRLF "\r\n"
3   #define VERSION "MegaSoft 0.0.7 pour Unix"

4   #define PORT_PAR_DEFAUT 8000
5   #define REPERTOIRE_PAR_DEFAUT "/tmp"

6   #define FATAL(err) { perror((char *) err); exit(1);}

7   extern void servir_client(int fd_client, char repertoire[]);

8   extern void envoyer_document(FILE * out,
9                                char nom_document[], char repertoire[]);

10  extern void document_non_trouve(FILE * out, char nom_document[]);

11  extern void requete_invalide(FILE * out);

12  extern int serveur_tcp(int numero_port);

13  extern int attendre_client(int fd_serveur);
```

17.3.2 Les fonctions orientées-réseau

- `serveur_tcp()` : création du socket du serveur TCP.
- `attendre_client()`

```
1   /*
2     Projet serveur Web - reseau.c
3     Fonctions réseau
4   */

5   #include <sys/types.h>
6   #include <sys/errno.h>
7   #include <sys/socket.h>
8   #include <sys/wait.h>
9   #include <sys/stat.h>
10  #include <netinet/in.h>
11  #include <signal.h>
12  #include <stdio.h>
13  #include <stdlib.h>

14  #include "declarations.h"

15  /* ------------------------------------------------------------
16    Fonctions réseau
17  ------------------------------------------------------------ */
```

```
18  int serveur_tcp(int numero_port)
19  {
20      int fd;

21      /* démarre un service TCP sur le port indiqué */

22      struct sockaddr_in addr_serveur;
23      size_t lg_addr_serveur = sizeof addr_serveur;

24      /* création de la prise */
25      fd = socket(AF_INET, SOCK_STREAM, 0);
26      if (fd < 0)
27          FATAL("socket");

28      /* nommage de la prise */
29      addr_serveur.sin_family = AF_INET;
30      addr_serveur.sin_addr.s_addr = INADDR_ANY;
31      addr_serveur.sin_port = htons(numero_port);
32      if (bind(fd, (struct sockaddr *) &addr_serveur, lg_addr_serveur) <
33          0)
34          FATAL("bind");

35      /* ouverture du service */
36      listen(fd, 4);
37      return (fd);
38  }

39  int attendre_client(int fd_serveur)
40  {
41      int fd_client;
42      /* A cause des signaux SIGCHLD, la fonction accept()
43         peut etre interrompue quand un fils se termine.
44         Dans ce cas, on relance accept().
45       */
46      while ((fd_client = accept(fd_serveur, NULL, NULL)) < 0) {
47          if (errno != EINTR)
48              FATAL("Fin anormale de accept().");
49      }
50      return (fd_client);
51  }
```

17.3.3 Les fonction de dialogue avec le client

- `dialogue_client()` : lecture et traitement de la requête d'un client
- `envoyer_document()`,
- `document_non_trouve()`,
- `requete_invalide()`.

```
1   /*
2     traitement-client.c

3     projet serveur WEB
4     Communication avec un client
5    */

6   #include <stdio.h>
```

```
 7  #include <stdlib.h>
 8  #include <string.h>
 9  #include <unistd.h>

10  #include "declarations.h"

11  void servir_client(int fdClient, char repertoire[])
12  {
13      FILE *in, *out;
14      char verbe[100], nom_document[100];
15      int fd2;

16      /* Ouverture de fichiers de haut niveau */
17      in = fdopen(fdClient, " r");
18      /* note : si on attache out au même descripteur que in,
19         la fermeture de l'un entraine la fermeture de l'autre */
20      fd2 = dup(fdClient);
21      out = fdopen(fd2, "w");

22      /* lecture de la requête, du genre
23         "GET quelquechose ..." */
24      fscanf(in, "%100s %100s", verbe, nom_document);
25      fclose(in);

26      if (strcmp(verbe, "GET") == 0)
27          envoyer_document(out, nom_document, repertoire);
28      else
29          requete_invalide(out);
30      fflush(out);                /* utile ? */
31      fclose(out);
32  }

33  void envoyer_document(FILE * out, char nom_document[],
34                        char repertoire[])
35  {
36      char nom_fichier[100];
37      FILE *fichier;
38      char ligne[100];

39      sprintf(nom_fichier, "%s%s", repertoire, nom_document);

40      if (strstr(nom_fichier, "/../") != NULL) {
41          /* tentative d'accès hors du répertoire ! */
42          document_non_trouve(out, nom_document);
43          return;
44      };
45      fichier = fopen(nom_fichier, "r");
46      if (fichier == NULL) {
47          document_non_trouve(out, nom_document);
48          return;
49      };

50      fprintf(out,
51              "HTTP/1.1 200 OK" CRLF
52              "Server: " VERSION CRLF
```

```
53                 "Content-Type: text/html; charset=iso-8859-1" CRLF CRLF);

54         fprintf(out,
55                 "<html><head><title>Fichier %s</title></head>" CRLF
56                 "<body bgcolor=\"white\"><h1>Fichier %s</h1>" CRLF
57                 "<center><table>"
58                 "<tr><td bgcolor=\"yellow\">"
59                 "<listing>" CRLF, nom_document, nom_fichier);

60         /* le corps du fichier */
61         while (fgets(ligne, 100, fichier) != NULL) {
62             char *p;
63             for (p = ligne; *p != '\0'; p++) {
64                 switch (*p) {
65                 case '<':
66                     fputs("&lt;", out);
67                     break;
68                 case '>':
69                     fputs("&gt;", out);
70                     break;
71                 case '&':
72                     fputs("&amp;", out);
73                     break;
74                 case '\n':
75                     fputs(CRLF, out);
76                     break;
77                 default:
78                     fputc(*p, out);
79                 };
80             };
81         };

82         /* balises de fin */
83         fputs("</listing></table></center></body></html>" CRLF, out
84     }

85     void document_non_trouve(FILE * out, char nom_document[])
86     {
87         /* envoi de la réponse : entête */
88         fprintf(out,
89                 "HTTP/1.1 404 Not Found" CRLF
90                 "Server: MegaSoft 0.0.7 (CP/M)" CRLF
91                 "Content-Type: text/html; charset=iso-8859-1" CRLF CRLF);

92         /* corps de la réponse */
93         fprintf(out,
94                 "<HTML><HEAD>" CRLF
95                 "<TITLE>404 Not Found</TITLE>" CRLF
96                 "</HEAD><BODY BGCOLOR=\"yellow\">" CRLF
97                 "<H1>Pas trouvé !</H1>" CRLF
98                 "Le document <font color=\"red\"><tt>%s</tt></font> "
99                 "demandé<br>n'est pas disponible.<P>" CRLF
100                "<hr> Le webmaster" "</BODY></HTML>" CRLF, nom_document);
101        fflush(out);
102    }
```

```
103     void requete_invalide(FILE * out)
104     {
105         fprintf(out,
106                 "<HTML><HEAD>" CRLF
107                 "<TITLE>400 Bad Request</TITLE>" CRLF
108                 "</HEAD><BODY BGCOLOR=\"yellow\">" CRLF
109                 "<H1>Bad Request</H1>"
110                 "Vous avez envoyé une requête que "
111                 "ce serveur ne comprend pas." CRLF
112                 "<hr> Le webmaster" "</BODY></HTML>" CRLF);
113         fflush(out);
114     }
```

17.3.4 Exercices, extensions...

Exercice : modifier `traitement-client` pour qu'il traite le cas des répertoires. Il devra alors afficher le nom des objets de ce répertoire, leur type, leur taille et un lien.

Exercice : Utiliser le mécanisme de transmission de descripteur (voir exemple plus loin) pour réaliser un serveur à processus préchargés.

17.3.5 Transmission d'un descripteur

L'exemple ci-dessous montre comment transmettre un decripteur d'un processus à un autre.

```
1     /*
2      exemple adapté de
3      http://www.mail-archive.com/linux-development-sys@senator-bedfellow.mit.edu/msg01240.html
4      (remplacement de alloca par malloc + modifications mineures)
5      */

6     /* passfd.c -- sample program which passes a file descriptor */

7     /* We behave like a simple /bin/cat, which only handles one
8      *    argument (a file name). We create Unix domain sockets through
9      *    socketpair(), and then fork(). The child opens the file whose
10     *    name is passed on the command line, passes the file descriptor
11     *    and file name back to the parent, and then exits. The parent waits
12     *    for the file descriptor from the child, then copies data from that
13     *    file descriptor to stdout until no data is left. The parent then
14     *    exits. */

15    #include <malloc.h>
16    #include <fcntl.h>
17    #include <stdio.h>
18    #include <string.h>
19    #include <sys/socket.h>
20    #include <sys/uio.h>
21    #include <sys/un.h>
22    #include <sys/wait.h>
23    #include <unistd.h>
24    #include <stdlib.h>

25    void die(char *message)
```

```
26      {
27          perror(message);
28          exit(EXIT_FAILURE);
29      }

30      /* Copies data from file descriptor 'from' to file descriptor 'to'
31       *    until nothing is left to be copied. Exits if an error occurs.   */
32      void copyData(int from, int to)
33      {
34          char buf[1024];
35          int amount;
36          while ((amount = read(from, buf, sizeof(buf))) > 0) {
37              if (write(to, buf, amount) != amount) {
38                  die("write");
39              }
40          }
41          if (amount < 0)
42              die("read");
43      }

44      /* The child process. This sends the file descriptor. */
45      int childProcess(char *filename, int sock)
46      {
47          int fd;
48          struct iovec vector;        /* some data to pass w/ the fd */
49          struct msghdr msg;          /* the complete message */
50          struct cmsghdr *cmsg;       /* the control message, which will include the fd */

51          /* Open the file whose descriptor will be passed. */
52          if ((fd = open(filename, O_RDONLY)) < 0) {
53              perror("open");
54              return EXIT_FAILURE;
55          }

56          /* Send the file name down the socket, including the trailing '\0' */
57          vector.iov_base = filename;
58          vector.iov_len = strlen(filename) + 1;

59          /* Put together the first part of the message. Include the file name iovec */
60          msg.msg_name = NULL;
61          msg.msg_namelen = 0;
62          msg.msg_iov = &vector;
63          msg.msg_iovlen = 1;

64          /* Now for the control message. We have to allocate room for
65           *        the file descriptor. */

66          cmsg = malloc(sizeof(struct cmsghdr) + sizeof(fd));
67          cmsg->cmsg_len = sizeof(struct cmsghdr) + sizeof(fd);
68          cmsg->cmsg_level = SOL_SOCKET;
69          cmsg->cmsg_type = SCM_RIGHTS;

70          /* copy the file descriptor onto the end of the control message */
71          memcpy(CMSG_DATA(cmsg), &fd, sizeof(fd));
```

```
72          msg.msg_control = cmsg;
73          msg.msg_controllen = cmsg->cmsg_len;

74          if (sendmsg(sock, &msg, 0) != vector.iov_len)
75              die("sendmsg");

76          free(cmsg);
77          return EXIT_SUCCESS;
78      }

79      /* The parent process. This receives the file descriptor. */
80      int parentProcess(int sock)
81      {
82          char buf[80];               /* space to read file name into */
83          struct iovec vector;        /* file name from the child */
84          struct msghdr msg;          /* full message */
85          struct cmsghdr *cmsg;       /* control message with the fd */
86          int fd;

87          /* set up the iovec for the file name */
88          vector.iov_base = buf;
89          vector.iov_len = 80;

90          /* the message we're expecting to receive */
91          msg.msg_name = NULL;
92          msg.msg_namelen = 0;
93          msg.msg_iov = &vector;
94          msg.msg_iovlen = 1;

95          /* dynamically allocate so we can leave room for the file
96           *        descriptor */
97          cmsg = alloca(sizeof(struct cmsghdr) + sizeof(fd));
98          cmsg->cmsg_len = sizeof(struct cmsghdr) + sizeof(fd);
99          msg.msg_control = cmsg;
100         msg.msg_controllen = cmsg->cmsg_len;

101         if (!recvmsg(sock, &msg, 0))
102             return EXIT_FAILURE;

103         printf("got file descriptor for '%s'\n",
104                (char *) vector.iov_base);

105         /* grab the file descriptor from the control structure */
106         memcpy(&fd, CMSG_DATA(cmsg), sizeof(fd));

107         copyData(fd, 1);

108         return EXIT_SUCCESS;
109     }

110     int main(int argc, char **argv)
111     {
112         int socks[2];
113         int status;
```

```
114         if (argc != 2) {
115             fprintf(stderr, "only a single argument is supported\n");
116             return 1;
117         }

118         /* Create the sockets. The first is for the parent and the
119          *        second is for the child (though we could reverse that
120          *        if we liked. */
121         if (socketpair(PF_UNIX, SOCK_STREAM, 0, socks))
122             die("socketpair");

123         if (fork() == 0) {          /* child */
124             close(socks[0]);
125             return childProcess(argv[1], socks[1]);
126         }

127         /* parent */
128         close(socks[1]);
129         parentProcess(socks[0]);

130         wait(&status);

131         if (WEXITSTATUS(status))
132             fprintf(stderr, "child failed\n");

133         return EXIT_SUCCESS;
134     }
```

18 Documentation

Les documents suivants ont été très utiles pour la rédaction de ce texte et la programmation des exemples :

- *Unix Frequently Asked Questions*
  `http://www.erlenstar.daemon.co.uk/unix/faq_toc.html`
- *HP-UX Reference, volume 2 (section 2 and 3, C programming routines).* HP-UX release 8.0, Hewlett-Packard. (Janvier 1991).
- *Advanced Unix Programming*, Marc J. Rochkind, Prentice-Hall Software Series (1985). ISBN 0-13-011800-1.
- *Linux online man pages* (Distribution Slackware 3.1).
- *The GNU C Library Reference Manual*, Sandra Loosemore, Richard M. Stallman, Roland McGrath, and Andrew Oram. Edition 0.06, 23-Dec-1994, pour version 1.09beta. Free Software Foundation, ISBN 1-882114-51-1.
- *What is multithreading ?*, Martin McCarthy, Linux Journal 34, Février 1997, pages 31 à 40.
- *Systèmes d'exploitation distribués*, Andrew Tanenbaum, InterEditions 1994, ISBN 2-7296-0706-4.
- Page Web de Xavier Leroy sur les threads :
  `http://pauillac.inria.fr/~xleroy/linuxthreads`

Sources :

```
$Source: /usr/labri/billaud/public_html/travaux/SYSRESEAU-juin-2004/RCS/sysreseau.sgml,v $
$Date: 2004/07/11 12:04:49 $
```