Système - Réseaux

Programmation Système et Réseau en C sous Unix

Juillet 2021

Michel Billaud

Résumé

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

Avant-propos

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).

Les principaux appels systèmes sont présentés ici, avec des exemples d'utilisation. [1]

Copyright, versions

(c) 1998-2021 Michel Billaud

Ce document peut être reproduit en totalité ou en partie, sans frais, sous réserve des restrictions suivantes :

— 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 toute responsabilité vis-à-vis des dommages résultant de l'utilisation des informations et des programmes qui figurent dans ce document.

— Version initiale 1998
— Révision 2002
— Révision 2014
— Révision 2016 (Conformité 11)
— Révision 2018 (POSIX 2017)
— Révision 2021 (conformité C17)

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

1. 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.

Chapitre 1

Bases de C

1.1 Exemple, compilation

Un exemple classique de programme écrit en C, à taper dans un fichier `hello.c`

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

int main(void)
{
    printf("Bonjour chez vous.\n");
    return EXIT_SUCCESS;
}
```

On peut le compiler par la commande

```
$ gcc -std=c18 -Wall -Wextra -pedantic -D_XOPEN_SOURCE=700 hello.c
```

Avec ces options
- le compilateur vérifie la conformité au dernier standard du langage C (C18)
- on bénéficie de la bibliothèque POSIX.1-2017, [1]
- un maximum d'avertissements sont affichés.

1.2 Lecture et affichage

1.2.1 Lecture et écriture standards : `printf()` et `scanf()`

```
#include <stdio.h>

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)`.

1. identique à IEEE Standard 1003.1-2017 et The Open Group Technical Standard Base Specifications, Issue 7. voir `http://pubs.opengroup.org/onlinepubs/9699919799`

1.2.2 Lecture et écriture dans une chaîne : `sprintf()` et `sscanf()`

```
#include <stdio.h>

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 connaît 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 `snprintf()` :

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

qui permet d'indiquer un nombre d'octets à ne pas dépasser.

1.2.3 Lancement d'une commande : `system()`

```
#include <stdlib.h>

int system (const char * string);
```

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

Exemple :

```
1  /*
    * Divers/imprimer.c
    *
    * Illustration de system();
5   */

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

10 #define TAILLE_MAX_COMMANDE      150
   #define TAILLE_MAX_NOM_FICHIER   100

   #define FORMAT_COMMANDE_IMPRESSION   "a2ps -4 -A page %s"

15 int main(void)
   {
       char nom_fichier[TAILLE_MAX_NOM_FICHIER];
       printf("nom du fichier ? ");
       fgets(nom_fichier, TAILLE_MAX_NOM_FICHIER, stdin);
20
       char commande[TAILLE_MAX_COMMANDE];
```

```
    snprintf(commande, TAILLE_MAX_COMMANDE,
             FORMAT_COMMANDE_IMPRESSION,
             nom_fichier);

    int resultat = system(commande);

    if (resultat == EXIT_SUCCESS) {
        printf("OK\n");
    } else {
        printf("La commande retourne : %d\n", resultat);
    }
    return resultat;
}
```

1.3 Communication avec l'environnement

1.3.1 Paramètres de `main()`

Le lancement d'un programme C provoque l'appel de sa fonction principale `main()`. Le standard C autorise deux formes pour la déclaration de `main()` :

```
int main(void);
int main(int argc, char *argv[]);
```

- `argc` est le nombre de paramètres sur la ligne de commande (y compris le nom de l'exécutable lui-même) ;
- `argv` est une tableau de chaînes contenant les paramètres de la ligne de commande.
- les noms `argc, argv` sont purement conventionnels.
- déclaration équivalente pour `argv` : `char **argv`

Exemple : programme qui affiche le tableau `argv` :

```
/*
 * Divers/env.c
 */

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

int main(int argc, char *argv[])
{
    printf("Appel avec %d paramètres\n", argc);

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

    return EXIT_SUCCESS;
}
```

1.3.2 `getopt()` : analyse des paramètres de la ligne de commande

La fonction `getopt` facilite l'analyse des options d'une ligne de commande. On lui fournit :
— le tableau des paramètres `argv` et sa taille `argc`
— une *chaîne de spécification d'options*. Par exemple la chaîne `"hxa:"` déclare 3 options possibles, la dernière (a) devra être suivies d'un paramètre.

```c
#include <unistd.h>

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

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

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

À la fin de l'analyse, `getopt()` retourne `-1`, et le tableau `argv` a été réarrangé pour que les paramètres supplémentaires (non liés aux options) soient stockés à partir de l'indice `optind`.

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

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

//  -h et -x sont de simples "switchs"
//  -a requiert un paramètre (:)

#define SPECIFICATION_OPTIONS   "hxa:"

void afficher_aide(const char prog[]); // déclaration avant usage

int main(int argc, char *argv[])
{
    bool  option_x_presente = false;
    char *argument_option_a = NULL;

    int   code = EXIT_SUCCESS;
    bool  options_restantes = true;

    do {
        int c = getopt(argc, argv, SPECIFICATION_OPTIONS);
        switch (c) {
        case -1 :
            options_restantes = false;
            break;
        case 'h':
            afficher_aide(argv[0]);
            break;
        case 'x':
            option_x_presente = true;
```

```
            break;
        case 'a' :
            argument_option_a = optarg;              // optional argument
            break;
        case '?':
            fprintf(stderr, "Option inconnue '-%c'.\n",
                    optopt);                  // option char
            code = EXIT_FAILURE;
            break;
        default:
            code = EXIT_FAILURE;
        }
    } while(options_restantes);

    printf("= option 'x' %s\n",
           (option_x_presente ? "activée" : "désactivée"));

    if (argument_option_a == NULL) {
        printf("= paramètre 'a' absent\n");
    } else {
        printf("= paramètre 'a' présent = %s\n", argument_option_a);
    }

    printf("%d paramètres supplémentaires\n",
           argc - optind);                    // option index
    for (int k = optind; k < argc; k++)
        printf ("    -> %s\n", argv[k]);

    return code;
}

void afficher_aide(const char prog[])          // définition
{
    printf("Help :  %s [options...] parametres ... \n\n",
           prog);
    printf("Options :\n"
           "-h\tCe message d'aide\n"
           "-x\toption x\n"
           "-a nom\t paramètre optionnel\n");
}
```

Exemple.

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

1.3.3 Variables d'environnement

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

```
#include <stdlib.h>

char *getenv(const char *name);
```

Exemple :

```
1  /* Divers/getlang.c */

   #include <stdlib.h>
   #include <stdio.h>
5
   int main(void)
   {
       const char *variable_lang = getenv("LANG");

10     printf("La variable d'environnement LANG ");
       if (variable_lang == NULL) {
           printf("n'est pas définie\n");
       } else {
           printf("contient \"%s\".\n", variable_lang);
15     }
       return EXIT_SUCCESS;
   }
```

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

```
#include <stdlib.h>

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.

Exercice vérifiez que ça ne modifie pas l'environnement du processus père.

1.3.4 exit() : Fin de programme

Pour arrêter un programme, deux solutions simples :
— 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` qui sont définies dans `stdlib.h`.

1.4 Erreurs

1.4.1 Variable errno, fonction perror()

La plupart des fonctions du système peuvent échouer pour diverses raisons. Habituellement, elles le signalent en retournant une valeur spéciale. On peut alors examiner la variable globale `errno` pour déterminer plus précisément la cause de l'échec, et agir en conséquence.

```
#include <stdio.h>

void perror(const char *s);

#include <errno.h>

extern int errno;
```

La fonction `perror()` imprime sur la sortie d'erreur standard un message qui décrit 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 l'appel système *chmod(2)*.

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

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

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

#define DROITS  (S_IRUSR | S_IWUSR)

void ecrire_message_erreur (int numero_erreur);

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

void ecrire_message_erreur (int numero_erreur)
{
    switch (numero_erreur) {
    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_erreur));
        break;
    }
}
```

1.4.2 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 semblant de traitement d'erreurs par *exceptions*. À employer avec précaution.

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. Le programme continue à l'endroit du `setjmp()` comme si celui-ci avait retourné la valeur `val`. (Si le paramètre `val` à 0, la valeur retournée est 1).

Exemple :

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

   #include <setjmp.h>
   #include <stdio.h>
5  #include <stdlib.h>
   #include <stdbool.h>

   enum {                              // définitions de constantes
       PAS_DE_SOLUTION = 1,
10     EQUATION_TRIVIALE = 2
   };

   float solution_equation(float a, float b);
   void traiter_des_equations();
15
   jmp_buf debut_de_main;

   int main(void)
   {
20     int code_retour = setjmp(debut_de_main);
       if ( code_retour == 0 ) {  // toujours 0 après l'appel
           traiter_des_equations();
           return EXIT_SUCCESS;
       }
25
       // si != 0, c'est que le déroulement a appelé longjmp
       // qui a fait revenir au setjmp.
       switch (code_retour) {
       case PAS_DE_SOLUTION:
30         printf("Cette équation n'a pas de solution_equation\n");
           break;
       case EQUATION_TRIVIALE:
           printf("Cette équation est toujours vraie.\n");
```

```
        break;
    }
    return EXIT_FAILURE;
}

float solution_equation(float a, float b)
{
    if (a == 0.0) {
        if (b == 0.0 ) {
            longjmp(debut_de_main, EQUATION_TRIVIALE);
        } else {
            longjmp(debut_de_main, PAS_DE_SOLUTION);
        }
    };
    return ( -b / a);
}

void traiter_des_equations()
{
    while (true) {
        printf("Coefficients de ax+b=0\n");
        float a,b;
        scanf("%f %f", &a, &b);
        float x = solution_equation(a,b);
        printf("x=%f est solution de %f x + %f = 0\n",
               x, a, b);
    }
}
```

1.5 Allocation dynamique

```
#include <stdlib.h>

void *malloc(size_t size);
void  free(void *ptr);
void *realloc(void *ptr, size_t size);
```

- `malloc()` - memory allocation - demande au système d'exploitation l'attribution 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 de redimensionner la zone allouée en conservant son contenu.

Exemple : la fonction `lire_nouvelle_ligne()` 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.

```
/* lireligne.c */
```

```c
#include <stdlib.h>
#include <stdio.h>
#include <stdbool.h>

#define CONTENANCE_INITIALE 16

char *lire_nouvelle_ligne(void);

int main(void)
{
    printf("tapez une grande ligne\n");
    char *nouvelle_ligne = lire_nouvelle_ligne();
    if (nouvelle_ligne == NULL) {
        fprintf(stderr, "Plus de place en mémoire\n");
        return EXIT_FAILURE;
    }
    printf("%s\n", nouvelle_ligne);
    free(nouvelle_ligne);
    return EXIT_SUCCESS;
}

char *lire_nouvelle_ligne(void)
{
    /*
       Renvoie un tampon de texte contenant une ligne
       lue sur l'entrée standard.
       Un nouveau tampon est créé à chaque invocation.
       Renvoie NULL en cas de problème d'allocation
    */

    int taille   = 0;
    int contenance = CONTENANCE_INITIALE;
    char *chaine = malloc(contenance);

    if (chaine == NULL) {          // ne devrait pas se produire
        return NULL;                 // mais il vaut mieux vérifier
    }

    while (true) {
        int c = getchar();
        if ((c == '\n') || (c == EOF)) {
            break;
        }
        chaine[taille++] = c;
        if (taille == contenance) {    // au besoin agrandir pour caser
            contenance *= 2;             // au moins le caractère nul de fin
            char *chaine_plus_grande = realloc(chaine, contenance);
            if (chaine_plus_grande == NULL) {
                free(chaine);
                return NULL;
            }
```

```
            chaine = chaine_plus_grande;
        }
    };
    chaine[taille] = '\0';
    return chaine;
}
```

Attention : dans l'exemple ci-dessus, la fonction `lire_nouvelle_ligne()` 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*. C'est ce que fait l'appel de `free()` dans la fonction `main()`.

Copie de chaîne : 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()`, qui ne fait pas pour l'instant partie des bibliothèques standards du langage C lui-même, mais (ouf !) de la bibliothèque POSIX. Elle apparaîtra dans C23.

```
#include <string.h>

char *strdup (const char *s);
```

Dans un environnement non-POSIX, la fonction peut être redéfinie aisément :

```
char *strdup (const char *s)
{
  char *new_string = malloc(strlen(s) + 1);
  return strcpy(new_string, s);
}
```

Chapitre 2

Fichiers et tuyaux

2.1 Manipulation des fichiers, opérations de haut niveau

2.1.1 Flots standards, entrées et sorties sur la console

Quand un programme est lancé, il y a trois *flots* pré-déclarés et ouverts, qui correspondent à l'entrée et la sortie standards, ainsi qu'à la sortie d'erreur :

```
#include <stdio.h>

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

Vous avez déjà rencontré quelques fonctions qui agissent sur ces flots, implicitement, sans les nommer en paramètre[1]

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

— `printf()` écrit sur `stdout` la valeur d'expressions selon un format donné.
— `scanf()` lit sur `stdin` la valeur de variables.
— pour lire une ligne complète, on fait appel à `fgets()` en utilisant le flot `stdin`, voir plus loin.

```
1  /* facture.c */

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

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

   int main(void)
   {
10     char  article[10];
       float prix;
       int   quantite;
```

1. Elles correspondent à des de fonctions plus générales `fprintf()`, `fscanf()`, etc. que nous verrons plus loin.

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

    if (nb_variables_lues != 3) {
        printf("Erreur dans les données");
        return EXIT_FAILURE;
    }

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

    return EXIT_SUCCESS;
}
```

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

2.1.2 Opérations sur les flots

```
#include <stdio.h>

FILE *fopen (char *path, char *mode);
int   fclose (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);
```

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

2.1.3 Lecture d'une ligne : fgets et getline

À l'ancienne : fgets

La fermeture par `fclose(flot)` provoquera la fermeture de `fd` à la fois en entrée et en sortie.

Quand on fait de la programmation réseau, on a parfois besoin de ne fermer la communication que dans un sens. Dans ce cas, on duplique le descripteur, et on crée un flot pour chacun :

```
int fd = fopen(.....);
...
FILE *entree = fdopen(    fd,  "r");
FILE *sortie = fdopen(dup(fd), "w");
```

La fermeture d'un des deux flots ne clôt qu'un sens de communication.

2.1.4 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 contient
  — SEEK_SET la position est donnée par rapport au début du fichier,
  — SEEK_CUR : déplacement par rapport à la position courante,
  — SEEK_END : déplacement par rapport à la fin.

2.1.5 Divers

```
int   ferror  (FILE *stream);
void  clearerr (FILE *stream);
```

— La fonction `ferror()` indique si une erreur a eu lieu sur un flot,
— `clearerr()` efface l'indicateur d'erreur

2.2 Manipulation des fichiers, opérations de bas niveau

Pendant l'exécution d'un programme, un certain nombre de fichiers sont *ouverts* (en cours d'utilisation).
I

Il existe dans le système une table des fichiers actuellement ouverts par le programme, les opérations de bas niveau désignent les fichiers par leur indice dans cette table. On appelle aussi ce numéro de fichier (`fileno`) un "descripteur de fichier" (`fd` = file descriptor).

Les numéros 0, 1 et 2 correspondent respectivement à l'entrée standard, la sortie standard, et la sortie d'erreur. Dans la programmation, utilisez plutôt les constantes `STDIN_FILENO` `STDOUT_FILENO`, `STDERR_FILENO` définies dans `unistd.h`.

Les opérations de bas niveau communiquent en général avec les fichiers par l'intermédiaire d'un tampon, un tableau d'octets, en indiquant le nombre d'octets à transmettre. Exemple :

```
1 char * message = "Hello, world";

  write(STDOUT_FILENO, message, 5);
```

envoie les 5 premiers caractères du message sur la sortie standard.

2.2.1 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`.

La fonction `open()` retourne le numéro de *descripteur de fichier* (-1 en cas d'erreur), un nombre entier qui sert à référencer le fichier par la suite.

```
#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 */

   /*
      Entrées-sorties de bas niveau
5     Usages :
      1 : copie              (entrée-standard -> sortie standard)
      2 : copie fichier      (fichier -> sortie standard)
      3 : copie source dest  (source -> dest)
   */
10
   #include <stdlib.h>
   #include <stdio.h>
   #include <sys/types.h>
   #include <sys/stat.h>
15 #include <fcntl.h>
   #include <unistd.h>
   #include <assert.h>

   #define TAILLE_TAMPON    4096
20
   void transfert(int fd_entree, int fd_sortie);

   int main(int argc, char *argv[])
   {
25     int fd_entree, fd_sortie;

       switch(argc) {
       case 1 :
           transfert(STDIN_FILENO, STDOUT_FILENO);
30         break;
       case 2:
           fd_entree = open(argv[1], O_RDONLY);
           transfert(fd_entree, STDOUT_FILENO);
           close(fd_entree);
35         break;
       case 3:
           fd_entree = open(argv[1], O_RDONLY);
           fd_sortie = open(argv[2], O_CREAT | O_WRONLY | O_TRUNC, 0666);
           transfert(fd_entree, fd_sortie);
40         close(fd_entree);
           close(fd_sortie);
           break;
       default:
           printf("Usage: copie [src [dest]]\n");
45         break;
       }
       return EXIT_SUCCESS ;
   }
```

```c
void transfert(int fd_entree, int fd_sortie)
{
    char tampon[TAILLE_TAMPON];

    assert( fd_entree >= 0);
    assert( fd_sortie >= 0);

    for(;;) {
        int nb_octets_lus = read(fd_entree, tampon, TAILLE_TAMPON);
        if (nb_octets_lus <= 0) {
            break;
        }
        int nb_octets_ecrits = write(fd_sortie, tampon, nb_octets_lus);
        assert(nb_octets_ecrits == nb_octets_lus);
    }
}
```

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).

2.2.2 Duplication de descripteurs

```c
#include <unistd.h>

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.

L'effet sera que le nouveau descripteur désignera la même fichier que l'ancien.

Exemple :

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

/*
  Le fichier cité en paramètre est passé à travers
  la commande wc.
*/

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

```c
#include <errno.h>
#include <assert.h>

const char COMMANDE [] = "wc";

int main (int argc, char * argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s fichier\n", argv[0]);
        return EXIT_FAILURE;
    }

    int fd_fichier = open(argv[1], O_RDONLY);
    assert (fd_fichier >= 0);

    /* transfert du descripteur dans celui de l'entrée standard */
    assert ( dup2(fd_fichier, STDIN_FILENO) >= 0);

    close(fd_fichier);

    system(COMMANDE); // l'entrée standard est reliée au fichier

    assert(errno == 0);
    return EXIT_SUCCESS;
}
```

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`").

2.2.3 Positionnement

```c
#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.

2.2.4 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. Écrire 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.

2.2.5 `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 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 /*
    Divers/inverse.c

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

    M. Billaud, Octobre 2002, corrigé Juin 2018 :-)
```

```
*/

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

void abandon (char message[], char nom_fichier[]);
void afficher_a_l_envers (const char *texte, size_t taille);
int  trouver_fd_et_taille(int *adr_fd,
                          size_t *adr_taille,
                          const char nom_fichier[]);

int main(int argc, char * argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s fichier\n", argv[0]);
        return EXIT_FAILURE;
    }
    char *nom_fichier = argv[1];

    int fd;
    size_t taille;
    trouver_fd_et_taille(& fd, & taille, nom_fichier);

    char *texte = mmap(NULL, taille, PROT_READ, MAP_PRIVATE, fd, 0);
    if (texte == MAP_FAILED) {
        abandon("Mise en mémoire", nom_fichier);
    }
    afficher_a_l_envers(texte, taille);

    if (munmap(texte, taille) < 0) {
        abandon("Détachement", nom_fichier);
    }

    close(fd);
    return EXIT_SUCCESS;
}

void abandon (char message[], char nom_fichier[])
{
    char tampon[200];
    snprintf(tampon, 200, "\"%s\" : %s", nom_fichier, message);
    perror(tampon);
    exit(EXIT_FAILURE);
}

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

```
60 void afficher_a_l_envers (const char *adr_debut_texte, size_t taille)
   {
       const char *adr_fin_ligne = adr_debut_texte + taille - 1;

       for (const char *p = adr_fin_ligne - 1; p >= adr_debut_texte; p--) {
65         if (*p == '\n') {  // position fin de ligne précédente
               int longueur = adr_fin_ligne - p;     // longueur ligne
               write(STDOUT_FILENO, p + 1,  longueur);
               adr_fin_ligne = p;
           };
70     }
       // affichage première ligne
       int longueur = adr_fin_ligne - adr_debut_texte + 1;
       write(STDOUT_FILENO, adr_debut_texte, longueur);
   }
75
   int trouver_fd_et_taille(int *adr_fd, size_t *adr_taille,
                            const char nom_fichier[])
   {
       struct stat etat;
80     if (stat(nom_fichier, &etat) != 0) {
           return 1;
       }

       if (! S_ISREG(etat.st_mode)) {
85         return 2;
       }

       *adr_fd = open(nom_fichier, O_RDONLY);
       if (*adr_fd < 0) {
90         return 3;
       }
       *adr_taille = etat.st_size;
       return 0;
   }
```

2.3 Fichiers, répertoires etc.

Ici “fichier” est compris dans son sens large (élément d’un système de fichiers), qui inclue aussi les répertoires, les périphériques, les tuyaux et sockets etc. (voir plus loin).

2.3.1 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*.

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

int stat(const char *file_name, struct stat *buf);
int fstat(int filedes, struct stat *buf);
```

2.3.2 Informations sur les fichiers/répertoires/...

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

Exemple :

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

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

15 const char *nom_mode(mode_t mode);

   int main(int argc, char *argv[])
   {
       if (argc < 2) {
20         fprintf(stderr, "Usage : %s fichier ... \n", argv[0]);
           exit(EXIT_FAILURE);
       };

       for (int k=1 ; k<argc ; k++) {
25         printf("%20s :\t", argv[k]);
           struct stat status;
           if (stat(argv[k], &status) == 0) {
               printf("taille %8ld, type %s",
                      status.st_size,
30                    nom_mode(status.st_mode));
           } else {
```

```
            switch(errno) {
            case ENOENT :
                printf("le fichier n'existe pas");
                break;
            default:
                printf("erreur %s", strerror(errno));
                break;
            }
        }
        printf("\n");
    }
    return EXIT_SUCCESS;
}

const char *nom_mode(mode_t mode)
{
    return S_ISREG(mode)    ? "fichier"
           :    S_ISLNK(mode)   ? "lien symbolique"
           :    S_ISDIR(mode)   ? "répertoire"
           :    S_ISCHR(mode)   ? "périphérique mode caractère"
           :    S_ISBLK(mode)   ? "peripherique mode bloc"
           :    S_ISFIFO(mode)  ? "fifo"
           :    S_ISSOCK(mode)  ? "socket"
           :    "inconnu";
}
```

2.3.3 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.

2.4 Tuyaux de communication

Les *tuyaux de communication* (*pipes*) permettent de faire communiquer des processus qui s'exécutent d'une même machine. Une fois ouverts, ils sont accessibles comme les fichiers à travers un "file descriptor". Ce que les processus y écrivent peut être lu immédiatement par d'autres processus.

Nous en présentons deux variétés :

— les tuyaux "simples" qui sont créés par un processus et servent à la communication entre ses descendants et lui.

— *tuyaux nommés*, qui sont visibles (comme les fichiers et répertoires), dans le système de fichiers. Ils peuvent donc être partagés par des programmes indépendants.

Nous verrons plus loin (3.1) une forme de communication plus générale, les sockets.

Nous commençons par les tuyaux nommés, dont l'usage est proche des fichiers ordinaires.

2.4.1 Tuyaux nommés (FIFO)

Les "tuyaux nommés" visibles dans l'arborescence des fichiers et répertoires. Ils sont créés par `mkfifo()`, et utilisés ensuite par `open()`, `read()`, `write()`, `close()`, `fdopen()`, etc.

```c
#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 :

```c
if (mkfifo("/tmp/fifo.courrier", 0644) != 0) {
  perror("mkfifo");
}
```

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. Écrire 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 une information quelconque (*jeton*) dans une FIFO. C'est la seconde option qui dépose le jeton, ce qui débloquera le processus lecteur. Prévoir une option pour créer une FIFO ?

2.4.2 Tuyaux (pipe)

On utilise un *pipe* (tuyau) pour faire communiquer un processus et un de ses descendants[2].

```
#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 5.1.1.

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 3.5) généralise la fonction `pipe`.

2.4.3 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 */

  /* Illustration de popen() */
```

2. ou des descendants - au sens large - du processus qui a créé le tuyau

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

#define TAILLE_MAX_COMMANDE 100


void envoyer_mail(const char *mail_destinataire);
void fabriquer_message_liste_fichiers(FILE *sortie);


int main (int argc, char * argv[])
{
    if (argc != 2) {
        fprintf(stderr,"Usage: %s destinataire\n", argv[0]);
        exit(EXIT_FAILURE);
    };
    envoyer_mail(argv[0]);
    exit(EXIT_FAILURE);
}

void envoyer_mail(const char *mail_destinataire)
{
    // ouverture du stream d'envoi
    char commande_envoi_mail[TAILLE_MAX_COMMANDE];
    snprintf(commande_envoi_mail, TAILLE_MAX_COMMANDE,
             "sendmail %s", mail_destinataire);
    FILE *stream_envoi_mail = popen(commande_envoi_mail, "w");
    assert(stream_envoi_mail != NULL);

    fabriquer_message_liste_fichiers(stream_envoi_mail);
    pclose(stream_envoi_mail);
}

void fabriquer_message_liste_fichiers(FILE *sortie)
{
    // envoi du message
    fprintf(sortie, "Subject: ls -l\n\n"
            "Cher ami,\n"
            "voici mon répertoire :\n");

    FILE *stream_liste_fichiers = popen("ls -l","r");
    assert(stream_liste_fichiers != NULL);

    int c;
    while( (c = fgetc(stream_liste_fichiers)) != EOF) {
        fputc(c, sortie);
    }
```

```
    pclose(stream_liste_fichiers);

    fprintf(sortie, "---\nLe Robot\n");
}
```

2.5 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.

2.5.1 Attente de données provenant de plusieurs sources

Exemple :

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

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

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

#define TAILLE_TAMPON 128

/*
   Mixe les données en provenance de deux
   descripteurs
*/

void mixer(int fd1, int fd2, int sortie)
{
    fd_set fds_ouverts;  /* les descripteurs ouverts */

    FD_ZERO (&fds_ouverts);
    FD_SET  (fd1, &fds_ouverts);
    FD_SET  (fd2, &fds_ouverts);
    int nb_fds_ouverts = 2;

    /* tant qu'il reste des descripteurs ouverts.... */

    while (nb_fds_ouverts > 0) {
        char tampon[TAILLE_TAMPON];

        /* on attend qu'un descripteur au moins soit prêt ...*/
        fd_set fds_prets = fds_ouverts;
        assert (select(FD_SETSIZE, &fds_prets, NULL, NULL, NULL) >= 0 );

        if (FD_ISSET(fd1, &fds_prets)) {   // fd1 est-il prêt ?

            int nb_octets_lus = read(fd1, tampon, TAILLE_TAMPON);
            if(nb_octets_lus >= 0) {
                write(sortie, tampon, nb_octets_lus);
            } else {
                close(fd1);          // fin de fd1 : on l'enlève
                nb_fds_ouverts--;
                FD_CLR(fd1, &fds_ouverts);
            }
        }

        if (FD_ISSET(fd2, &fds_prets)) {
            int nb_octets_lus = read(fd2, tampon, TAILLE_TAMPON);
            if(nb_octets_lus >= 0) {
                write(sortie, tampon, nb_octets_lus);
```

```
            } else {
                close(fd2);
                nb_fds_ouverts--;
                FD_CLR(fd2, &fds_ouverts);
            }
        }
    }
}

int main (int argc, char *argv[])
{
    if (argc != 3) {
        fprintf(stderr, "Usage : %s f1 f2\n", argv[0]);
        return EXIT_FAILURE;
    }

    int fd1 = open(argv[1], O_RDONLY);
    assert(fd1 >= 0);

    int fd2 = open(argv[2], O_RDONLY);
    assert(fd2 >= 0);

    mixer(fd1, fd2, 1);

    return EXIT_SUCCESS;
}
```

2.5.2 Attente de données avec limite de temps

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

```
/*
  SelectFifo/lecteur.c

  Exemple de lecture avec délai (timeout).
  M. Billaud, Septembre 2002 - revu 2016

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

  Exemple d'utilisation :
  - créer une fifo : mkfifo /tmp/mafifo
  - dans une fenêtre, lancer : lecteur /tmp/mafifo
  le programme se met en attente
  - dans une autre fenêtre, faire
  cat > /tmp/mafifo
```

```
   puis taper quelques lignes de texte.

 */

#include <sys/time.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <fcntl.h>
#include <stdbool.h>

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

#define SYMBOLES      "-\\|/"
#define NBSYMBOLES    4

void montrer_animation();

int attendre_donnees(int fd, int millisecondes);

void afficher_donnees(int fd);

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, " Usage: %s fifo\n", argv[0]);
        exit(EXIT_FAILURE);
    }
    printf("> Ouverture fifo %s ...\n", argv[1]);
    int fd = open(argv[1], O_RDONLY);
    if (fd == -1) {
        fprintf(stderr, "Ouverture refusée\n");
        exit(EXIT_FAILURE);
    };
    printf("OK\n");

    afficher_donnees(fd);

    close(fd);
    exit(EXIT_SUCCESS);
}

// ------------------------------------------------------

void afficher_donnees(int fd)
{
    int numero_ligne = 1;
    while (true) {
```

```
        int r = attendre_donnees(fd, DELAI);
        if (r == 0) { // rien reçu
            montrer_animation();
        } else { // on a reçu quelque chose
            char tampon[TAILLE_TAMPON];
            int nb_octets_lus = read(fd, tampon, TAILLE_TAMPON - 1);
            if (nb_octets_lus <= 0) { // fin
                break;
            }
            tampon[nb_octets_lus] = '\0';
            printf("%4d %s", numero_ligne, tampon);
                        numero_ligne += 1;
        }
    }
}

int attendre_donnees(int fd, int millisecondes)
{
    fd_set set;
    FD_ZERO(&set);
    FD_SET(fd, &set);

    struct timeval delai = {
        .tv_sec  = millisecondes / 1000,
        .tv_usec = (millisecondes % 1000) * 1000
    };

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

void montrer_animation()
{
    static int n = 0;
    printf("%c\b", SYMBOLES[n++ % NBSYMBOLES]);
    fflush(stdout);
}
```

Chapitre 3
Communication interprocessus par sockets locaux

3.1 Les sockets

Les *sockets* (*prises*) sont une interface générique pour la communication entre processus, par divers moyens.

On s'en sert pour la communication à travers les réseaux (Internet ou autres), mais ils sont utilisables également pour la communication locale entre processus qui s'exécutent sur une même machine (comme les tuyaux déjà vus, et les files de messages IPC que nous verrons plus loin).

Dans ce chapitre, nous montrons comment s'en servir pour la communication locale, la communication sur le réseau sera vue plus loin.

3.1.1 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. [1]

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 connexion 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,

1. 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`, etc.

évidemment au prix d'un surcoût (numérotation des paquets, accusés de réception, temporisations, retransmissions, etc).

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

3.1.2 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 = sizeof adresse;
```

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

3.2 Communication par datagrammes

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

3.2.1 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[2] est placée dans la structure pointée par `from`, dont la longueur maximale est contenue dans l'entier pointé par `fromlen`.

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  /* LocalDatagrammes/serveur-dgram-local.c */

   /*
      Usage : serveur-dgram-local chemin
5
      Reçoit des datagrammes par un socket du domain local,
      et les affiche. Le paramètre indique le nom du socket.
      S'arrête quand la donnée est "stop".
   */
10
   #include <stdio.h>
   #include <unistd.h>
   #include <stdlib.h>
   #include <sys/types.h>
15 #include <sys/socket.h>
   #include <sys/un.h>
   #include <string.h>
   #include <stdbool.h>

20 #define TAILLE_MAX_DONNEE 1024

   void abandon(char message[]);
   int ouvrir_socket_reception(const char *chemin);

25

   int main (int argc, char * argv[])
   {
       if (argc != 2) {
           fprintf(stderr,"usage : %s chemin\n", argv[0]);
30         abandon("mauvais nombre de parametres");
       }

       int fd = ouvrir_socket_reception(argv[1]);

35     printf("> Serveur démarré sur socket local \"%s\"\n", argv[1]);
       while (true) {
           char tampon[TAILLE_MAX_DONNEE + 1]; // +1 pour ajouter terminateur
           int nb_octets_recus
               = recvfrom(fd, tampon, TAILLE_MAX_DONNEE, 0, NULL, NULL);
40         if (nb_octets_recus <= 0) {
```

2. qui peut servir à expédier une réponse

```
            abandon("Réception datagramme");
        }
        tampon[nb_octets_recus] = '\0'; // ajout terminateur
        printf("Reçu : %s\n", tampon);
        if (strcmp(tampon, "stop") == 0) {
            printf("> arrêt demandé\n");
            break;
        }
    }
    close(fd);
    return EXIT_SUCCESS;
}


int ouvrir_socket_reception(const char *chemin)
{
    // construction de l'adresse
    struct sockaddr_un  adresse;
    adresse.sun_family  = AF_LOCAL;
    strcpy(adresse.sun_path, chemin);

    // construction du socket de réception
    int fd = socket(PF_LOCAL, SOCK_DGRAM, 0);
    if (fd < 0) {
        abandon("Création du socket serveur");
    }

    // association socket/adresse
    socklen_t longueur_adresse  = sizeof adresse;
    if (bind(fd, (struct sockaddr *) &adresse, longueur_adresse) < 0) {
        abandon("Nommage du socket serveur");
    }
    return fd;
}

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

3.2.2 Émission 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  /* LocalDatagrammes/client-dgram-local.c */

   /*
      Usage :  client-dgram-local chemin messages
5
      Envoie des datagrammes à un socket du domain local,
      et les affiche. Le premier paramètre indique le nom du socket,
      les autres des chaînes de caractères.

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

   #include <stdio.h>
   #include <unistd.h>
15 #include <stdlib.h>
   #include <sys/types.h>
   #include <sys/socket.h>
   #include <sys/un.h>
   #include <string.h>
20
   #define TAILLE_MAX_DONNEE 1024

   void abandon(char message[]);

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

       struct sockaddr_un  adresse;
       adresse.sun_family      = AF_LOCAL;
       strcpy(adresse.sun_path,argv[1]);
35     int longueur_adresse        = sizeof adresse; // SUN_LEN (&adresse);

       int fd = socket(PF_LOCAL,SOCK_DGRAM,0);
       if (fd < 0) {
           abandon("Création du socket client");
40     }

       for (int k = 2; k < argc; k++) {
           // limitation
           int nb_octets = strlen(argv[k]);
45         if (nb_octets > TAILLE_MAX_DONNEE) {
               nb_octets = TAILLE_MAX_DONNEE;
           }
```

```
        /* le message est envoyé sans le terminateur '\0' */
        if ( sendto(fd, argv[k], nb_octets, 0,
                    (struct sockaddr *) &adresse,
                    longueur_adresse) < 0) {
            abandon("Expédition du message");
        }
        printf(".");
        fflush(stdout);
        sleep(1);
    }

    printf("OK\n");
    close(fd);
    return EXIT_SUCCESS;
}

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}
```

Exercice : modifier les programmes précédents pour que le serveur envoie une réponse qui sera affichée par le client[3].

3.2.3 Émission et réception en mode connecté

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

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

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

3.3 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 créés par

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

3.4 Architecture client-serveur

La plupart des applications communicantes sont conçues selon une architecture client-serveur, asymétrique, 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 créé, 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 retourne 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 :

```c
/*
  LocalStream/client-stream.c

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

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

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <assert.h>

#define TAILLE_TAMPON 1000

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}

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

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

    /* 2. Initialisation du socket */

    /* 2.1 création du socket   */
    fd = socket(PF_LOCAL, SOCK_STREAM, 0);
    if (fd < 0)
        abandon("Création du socket client");

```

```
    /* 2.2 Remplissage adresse serveur */
    adresse.sun_family = AF_LOCAL;
    strcpy(adresse.sun_path, chemin);
    longueur_adresse = sizeof adresse;

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

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

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

        nb_lus = read(0, tampon, TAILLE_TAMPON);
        if (nb_lus <= 0)
            break;
        nb_envoyes = write(fd, tampon, nb_lus);
        assert (nb_envoyes == nb_lus);
    }
    /* 4. Fin de l'envoi */
    shutdown(fd, SHUT_WR);
    printf("CLIENT> Fin envoi, attente de la réponse.\n");

    /* 5. Réception et affichage de la réponse */
    for (;;) {
        char tampon[TAILLE_TAMPON];
        int nb_lus;
        nb_lus = read(fd, tampon, TAILLE_TAMPON - 1);
        if (nb_lus <= 0)
            break;
        tampon[nb_lus] = '\0';  /* ajout d'un terminateur de chaîne */
        printf("%s", tampon);
    }
    /* et fin */
    close(fd);
    printf("CLIENT> Fin.\n");
    return EXIT_SUCCESS;
}
```

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.

```
/*
  LocalStream/serveur-stream.c

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

  Exemple de serveur qui
```

```
  - attend une connexion
  - lit du texte
  - envoie une réponse

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


#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <assert.h>

#define TAILLE_TAMPON               1000
#define MAX_CONNEXIONS_EN_ATTENTE 4


void dialoguer_avec_client(int fd_client, int numero_client);


int main(int argc, char *argv[])
{
    int numero_connexion = 0;

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

    if (argc != 2) {
        printf("usage : %s chemin\n", argv[0]);
        return EXIT_FAILURE;
    }
    char * chemin = argv[1];

    /* 3. Initialisation du socket de réception */
    /* 3.1 création du socket   */
    int fd_serveur = socket(PF_LOCAL, SOCK_STREAM, 0);
    assert (fd_serveur >= 0);

    /* 3.2 Remplissage adresse serveur */
    struct sockaddr_un adresse;
    adresse.sun_family = AF_LOCAL;
    strcpy(adresse.sun_path, chemin);
    size_t taille_adresse = sizeof adresse;
```

```
    /* 3.3 Association de l'adresse au socket */
    assert (bind(fd_serveur,
                 (struct sockaddr *) &adresse, taille_adresse) >= 0);


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

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

    /* 4. boucle du serveur */
    for (;;) {
        /* 4.1 attente d'une connexion */
        printf("SERVEUR> Attente d'une connexion.\n");
        int fd_client = accept(fd_serveur, NULL, NULL);
        assert (fd_client > 0);

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

        dialoguer_avec_client(fd_client, numero_connexion);

        /* 4.3 fermeture de la connexion */
        close(fd_client);

    }

    /* on ne passe jamais ici */
    return EXIT_SUCCESS;
}

void dialoguer_avec_client(int fd_client, int numero_client)
{
    int compteur = 0;
    while(true) {
        char tampon_lecture[TAILLE_TAMPON];

        int nb_octets_lus
            = read(fd_client, tampon_lecture, TAILLE_TAMPON - 1);
        if (nb_octets_lus <= 0) {
            break;
        }
        compteur += nb_octets_lus;
        tampon_lecture[nb_octets_lus] = '\0';
        printf("%s", tampon_lecture);
    }

    /* 4.4 plus de données, on envoit une réponse */
    printf("SERVEUR> envoi de la réponse.\n");
```

```
    char tampon_reponse[TAILLE_TAMPON];
    int taille_reponse
        = sprintf(tampon_reponse,
                  "*** Fin de la connexion au client #%d\n"
                  "*** Vous m'avez envoyé %d caractères\n"
                  "*** Merci et à bientot.\n",
                  numero_client, compteur);
    write(fd_client, tampon_reponse, taille_reponse);
    printf("SERVEUR> fin de connexion.\n");
}
```

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.

```
/*
  LocalStream/serveur-stream-monotache.c

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

  Exemple de serveur monotâche qui gère plusieurs connexions

  - attend une connexion
  - lit du texte
  - envoie une réponse
 */


#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <sys/un.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <assert.h>
#include <string.h>
#include <sys/select.h>
#include <stdbool.h>


#define TAILLE_TAMPON               1000
```

```
#define MAX_CONNEXIONS_EN_ATTENTE 4
#define MAX_CLIENTS 10

/* les données propres à chaque client */

#define INACTIF -1
struct {
    int fd;
    int numero_connexion;
    int compteur;
} client[MAX_CLIENTS];

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}

int main(int argc, char *argv[])
{
    size_t taille_adresse;
    int nb_connexions = 0;

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

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

    /* 3. Initialisation du socket de réception */
    /* 3.1 création du socket   */
    struct sockaddr_un adresse;
    int fd_serveur = socket(PF_LOCAL, SOCK_STREAM, 0);
    if (fd_serveur < 0) {
        abandon("Création du socket serveur");
    }

    /* 3.2 Remplissage adresse serveur */
    adresse.sun_family = AF_LOCAL;
    strcpy(adresse.sun_path, chemin);
    taille_adresse = sizeof adresse;

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

```
80      /* 3.4 Ce socket attend des connexions mises en file d'attente */
        listen(fd_serveur, MAX_CONNEXIONS_EN_ATTENTE);

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

85      /* 3.5 initialisation du tableau des clients */
        for (int i = 0; i < MAX_CLIENTS; i++) {
            client[i].fd = INACTIF;
        }

90      /* 4. boucle du serveur */

        while(true) {
            /* 4.1 remplissage des masques du select */
            fd_set fds_lecture;
95          FD_ZERO(&fds_lecture);
            // serveur (pour nouvelles connexions)
            FD_SET(fd_serveur, &fds_lecture);
            // clients actifs
            for (int i = 0; i < MAX_CLIENTS; i++) {
100             if (client[i].fd != INACTIF)
                    FD_SET(client[i].fd, &fds_lecture);
            }

            /* 4.2 attente d'un événement (ou plusieurs) */
105         int nb_fds_prets = select(FD_SETSIZE,
                                      &fds_lecture, NULL, NULL,
                                      NULL);
            assert(nb_fds_prets >= 0);

110         /* 4.3 en cas de nouvelle connexion : */
            if (FD_ISSET(fd_serveur, &fds_lecture)) {
                /* si il y a de la place dans la table des clients,
                   on y ajoute la nouvelle connexion */
                nb_fds_prets--;
115             for (int i = 0; i < MAX_CLIENTS; i++) {
                    if (client[i].fd == INACTIF) {
                        int fd_client = accept(fd_serveur, NULL, NULL);
                        if (fd_client < 0)
                            abandon("accept");
120                     nb_connexions++;
                        client[i].fd = fd_client;
                        client[i].numero_connexion = nb_connexions;
                        client[i].compteur = 0;
                        printf("SERVEUR> arrivée de connexion #%d (fd %d)\n",
125                            client[i].numero_connexion, fd_client);
                        break;
                    }
                    if (i >= MAX_CLIENTS) {
                        printf("SERVEUR> trop de connexions !\n");
130                 }
```

```
            }
        };

        /* 4.4 traitement des clients actifs qui ont reçu des données */
        for (int i = 0; (i < MAX_CLIENTS) && (nb_fds_prets > 0); i++) {
            if ((client[i].fd != INACTIF) &&
                    FD_ISSET(client[i].fd, &fds_lecture)) {
                nb_fds_prets--;
                char tampon[TAILLE_TAMPON];
                int nb_octets_lus = read(client[i].fd, tampon,
                                         TAILLE_TAMPON - 1);
                printf("SERVEUR> données reçues de #%d (%d octets)\n",
                       client[i].numero_connexion, nb_octets_lus);
                if (nb_octets_lus > 0)
                    client[i].compteur += nb_octets_lus;
                else {
                    printf("SERVEUR> envoi de la réponse au client #%d.\n",
                           client[i].numero_connexion);
                    int taille_reponse =
                        sprintf(tampon,
                                "*** Fin de la connexion #%d\n"
                                "*** Vous m'avez envoyé %d caractères\n"
                                "*** Merci et à bientôt.\n",
                                client[i].numero_connexion,
                                client[i].compteur);
                    write(client[i].fd, tampon, taille_reponse);
                    close(client[i].fd);
                    /* enlèvement de la liste des clients */
                    client[i].fd = INACTIF;
                }
            }
        }
    }
    /* on ne passe jamais ici (boucle sans fin) */
    return EXIT_SUCCESS;
}
```

3.5 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[4]. `socketpair()` possède deux avantages sur `pipe()` : la possibilité de transmettre des datagrammes, et la bidirectionnalité.

Exemple :

```
/*
  paire-send.c

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

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

struct paquet {
    int type;
    int valeur;
};

/* les différents types de paquets */

#define DONNEE   0
#define RESULTAT 1
#define FIN      2

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}

int additionneur(int fd)
{
    /*
       calcule la somme des entiers qui arrivent sur le descripteur,
       renvoie le résultat
    */
    int somme = 0;
    while(true) {
        struct paquet recu;
        int n = recv(fd, &recu, sizeof recu, 0);
        if (n < 0)
```

4. au sens large, ce qui inclue la communication d'un processus avec un de ses fils

```
            abandon("recv");
        printf("additionneur : réception d'un paquet contenant ");
        if (recu.type == FIN) {
            printf("la marque de fin\n");
            break;
        }
        printf("la donnée %d\n", recu.valeur);
        somme += recu.valeur;
    };
    /* envoi reponse */

    struct paquet reponse;
    reponse.type = RESULTAT;
    reponse.valeur = somme;
    printf("additionneur : envoi du total %d\n", somme);
    int n = send(fd, &reponse, sizeof reponse, 0);
    if (n < 0) {
        abandon("additionneur : send");
    }
    return EXIT_SUCCESS;
}

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

    struct paquet p;

    for (int i = 1; i <= 5; i++) {
        p.type = DONNEE;
        p.valeur = i;
        printf("générateur : envoi de la donnée %d\n", i);
        int n = send(fd, &p, sizeof p, 0);
        if (n < 0)
            abandon("generateur : send");
        sleep(1);
    };
    p.type = FIN;
    printf("générateur : envoi de la marque de fin\n");
    int n = send(fd, &p, sizeof p, 0);
    if (n < 0) {
        abandon("generateur : send");
    }
    printf("generateur : lecture du résultat\n");
    n = recv(fd, &p, sizeof p, 0);
    if (n < 0) {
        abandon("generateur : recv");
    }
    int resultat = p.valeur;
```

```
    printf("generateur : resultat reçu = %d\n", resultat);
    return EXIT_SUCCESS;
}

int main()
{
    int paire_sockets[2];

    socketpair(AF_LOCAL, SOCK_DGRAM, 0, paire_sockets);

    if (fork() == 0) {
        close(paire_sockets[0]);
        return additionneur(paire_sockets[1]);
    } else {
        close(paire_sockets[1]);
        return generateur(paire_sockets[0]);
    }
}
```

Chapitre 4
Communication par signaux

Ce chapitre présente la communication par signaux entre processus.

Il y a essentiellement deux opérations sur les signaux. En gros :

— un appel de fonction demande l'envoi d'un *signal* à un autre processus (destinataire), désigné par son numéro de processus.

— le destinataire a indiqué, par autre appel de fonction (`signal` ou `sigaction`) quel traitement doit être exécuté quand il reçoit un signal.

Un signal est un simple nombre, dont la valeur correspond à un particulier. Il peut être émis par un programme (la fonction d'envoi s'appelle `kill()`) (pour des raisons historiques), ou résulter d'un évènement système (fin d'un processus fils, ...), ou d'une division par zéro, etc.

En particulier, quand on fait tourner un processus depuis un shell et qu'on tape controle-c, ce caractère est reçu par le shell qui envoie alors le signal `SIGINT` (numéro 2) au processus qui tourne en avant-plan et rend la main à la boucle interactive du shell. De même, contrôle-Z envoie un `SIGTSTP` (20) qui demande au destinataire de se mettre en pause (terminal stop). Ce qu'il fait en s'envoyant lui-même le signal `SIGSTOP` (19).

Un signal a un comportement par défaut (arrêter le programme, ne rien faire, ...), la seconde fonction `signal/sigaction` sert à le changer.

On regarde ici deux bibliothèques liées aux signaux :

— la bibliothèque des signaux "classiques" d'UNIX, qui est simple à utiliser, mais n'est pas vraiment portable [1]

— la bibliothèque définie par la norme POSIX, plus riche mais aussi plus complexe.

4.1 Les signaux Unix

4.1.1 `signal()`

L'implémentation GNU se présente sous la forme

```
#include <stdio.h>

sighandler_t signal(int signum, sighandler_t handler);
```

où le type `sighandler_t` désigne les fonctions qui prennent comme paramètre un `int` :

```
typedef void (*sighandler_t)(int);
```

1. le comportement peut être "légèrement" différent selon les versions d'UNIX.

La déclaration “standard” est un peu plus difficile à lire :

```
void ( *signal(int signum, void (*handler)(int)) ) (int);
```

Rôle : `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[2], 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  // Signaux/sig-unix.c */

   #include <stdlib.h>
   #include <signal.h>
5  #include <errno.h>
   #include <unistd.h>
   #include <stdio.h>
   #include <stdbool.h>

10 #define DELAI  1        // secondes

   bool continuer = true;

   void traiter_signal(int numero_signal);
15
   int main(void)
   {
       signal(SIGTSTP, traiter_signal);    // si on reçoit contrôle-Z
       signal(SIGINT,  traiter_signal);    // si contrôle-C
20     signal(SIGTERM, traiter_signal);    // si kill processus

       while (continuer) {
           sleep(DELAI);
           printf(".");
25         fflush(stdout);
       }
       printf("fin\n");
       exit(EXIT_SUCCESS);
   }
30
   void traiter_signal(int numero_signal)
   {
```

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

```
    printf("Signal %d => ", numero_signal);
    switch (numero_signal) {
    case SIGTSTP:
        printf("Je m'endors....\n");
        kill(getpid(), SIGSTOP);            //  auto-endormissement

        // Bloqué ici. On repart si on reçoit SIGCONT

        printf("Je me réveille !\n");
        signal(SIGTSTP, traiter_signal);    //  repositionnement du traitement
        break;
    case SIGINT:
    case SIGTERM:
            continuer = false;
        break;
    }
}
```

4.1.2 kill()

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

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

4.1.3 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.

4.1.4 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()`.

4.2 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.

4.2.1 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.

4.2.2 `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  /* Divers/sig-posix.c */

   #include <stdlib.h>
   #include <signal.h>
5  #include <errno.h>
   #include <unistd.h>
   #include <stdio.h>

   #define DELAI          1        /*secondes */
10 #define NB_ITERATIONS 60

   void traiter_signal(int numero_signal)
   {
       struct sigaction rien, ancien_traitement;
15
       printf("Signal %d => ", numero_signal);

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

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

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

           /* remise en place ancien traitement */
           sigaction(SIGTSTP, &ancien_traitement, NULL);
35         printf("C'est reparti !\n");
           break;
       case SIGINT:
       case SIGTERM:
           printf("On m'a demandé d'arrêter le programme.\n");
40         exit(EXIT_SUCCESS);
           break;
       }
   }

45 int main(void)
   {
       struct sigaction action;
```

```
    action.sa_handler = traiter_signal; /* fonction à lancer */
    sigemptyset(&action.sa_mask);       /* rien à masquer */

    sigaction(SIGTSTP, &action, NULL);  /* pause contrôle-Z */
    sigaction(SIGINT,  &action, NULL);  /* fin contrôle-C */
    sigaction(SIGTERM, &action, NULL);  /* arrêt */


    for (int i = 1; i < NB_ITERATIONS; i++) {
        sleep(DELAI);
        printf("%d", i % 10);
        fflush(stdout);
    }

    printf("Fin\n");
    return EXIT_SUCCESS;
}
```

Chapitre 5
Processus lourds et légers

Dans beaucoup de programmes il est nécessaire d'avoir plusieurs processus qui tournent simultanément pour réaliser ensemble un certain travail. Ils collaboreront à travers des tuyaux, du partage de mémoire et autres techniques décrites ailleurs dans ce document.

Ici nous présentons deux types de processus. Le premier, présent dès les premières versions d'UNIX, s'obtient essentiellement en dupliquant le processus qui le lance (contenu de la mémoire, fichiers ouverts, ressource diverses, etc). Le processus "fils" possède alors un espace mémoire totalement séparé de celui du "père qui l'a lancé".

Cette duplication fait qu'on l'appelle souvent *processus lourd*, par opposition aux *processus légers* introduits plus tard qui, eux, partagent le même espace mémoire et les mêmes ressources. Cette "lourdeur" est à relativiser, le système met en oeuvre des techniques comme le copy-on-write qui limitent les dégâts.

Contrairement aux signaux POSIX par rapport aux signaux Unix, on ne peut pas dire que les processus légers rendent obsolètes les processus "lourds" : les deux ont leur utilité, leurs avantages et inconvénients.

5.1 Les processus lourds

5.1.1 `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  /* Divers/biproc.c */

   /*
    * Illustration de fork() et pipe();
5   *
    * Exemple à deux processus reliés par un tuyau
```

```
 - l'un envoie abcdef...z 10 fois dans le tuyau
 - l'autre écrit ce qui lui arrive du tuyau sur la
  sortie standard, en le formattant.
 */

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

#define TAILLE_LIGNE        30
#define TAILLE_ALPHABET     26
#define NOMBRE_REPETITIONS  10

void produire_donnees (int fd_sortie);
void consommer_donnees (int entree);
int lire_donnees (int fd_entree, char *tampon, size_t taille_tampon);

int main (void)
{
    int pipe_fils_pere[2];

    assert (pipe(pipe_fils_pere) == 0);

    pid_t pid_fils = fork();
    assert(pid_fils >= 0);

    if (pid_fils == 0) {
        /* le processus fils */
        close(pipe_fils_pere[0]);
        close(STDOUT_FILENO);
        produire_donnees(pipe_fils_pere[1]);
    } else {
        /* le processus père continue ici */
        close(STDIN_FILENO);
        close(pipe_fils_pere[1]);

        consommer_donnees(pipe_fils_pere[0]);

        int status;
        wait(&status);
        printf("status fils = %d\n", status);
    }
    return EXIT_SUCCESS;
}

void produire_donnees (int fd_sortie)
{
    char alphabet[TAILLE_ALPHABET];
```

```
    for (int k = 0; k < TAILLE_ALPHABET; k++) {
        alphabet[k] = 'a'+k;
    }

    for (int k = 0; k < NOMBRE_REPETITIONS; k++) {
        int nb_ecrits = write(fd_sortie, alphabet, TAILLE_ALPHABET);
        assert(nb_ecrits == TAILLE_ALPHABET);
    }
    close(fd_sortie);
}

int lire_donnees (int fd_entree, char *tampon, size_t taille_tampon)
{
    /* lecture, en insistant pour remplir le tampon */
    size_t deja_lus = 0;

    while (deja_lus < taille_tampon) {
        int n = read(fd_entree, tampon + deja_lus,
                     taille_tampon - deja_lus);
        if (n < 0) {
            return -1;
        }
        if (n == 0) {
            break; /* plus rien à lire */
        }
        deja_lus += n;
    }
    return deja_lus;
}

void consommer_donnees (int fd_entree)
{
    char ligne[TAILLE_LIGNE+1];
    int taille, numero_ligne = 1;

    while ( (taille = lire_donnees(fd_entree,
                                   ligne, TAILLE_LIGNE)) > 0) {
        ligne[taille] = '\0';
        printf("%3d %s\n", numero_ligne++, ligne);
    };
    assert (taille >= 0);
    close(fd_entree);
}
```

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()` ?

5.1.2 `waitpid()` : attente de changement d'état

La fonction `waitpid()` permet d'attendre un *changement d'état* d'un des processus fils désigné par son *pid* (n'importe lequel si *pid = -1*), et de récupérer éventuellement son code de retour.

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

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

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

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");
...
```

La fonction retourne

— si un processus fils s'est terminé, le numéro du processus fils ;
— `-1` si le processus a reçu un signal, et la variable `errno` contient alors `EINTR` ;
— `0` si l'option `WNOHANG` était indiqué, et qu'aucun processus n'a changé d'état

Les macros suivantes permettent de connaître la nature du changement d'état

— `WIFEXITED(status)` Vrai si le fils s'est terminé normalement (appel à `exit()` ou `return` depuis `main()`. Dans ce cas `WEXITSTATUS(status)` donne la valeur du code de retour status fourni par le processus fils.
— `WIFSIGNALED(status)` Vrai si le fils s'est terminé à cause d'un signal non intercepté. Dans ce cas `WTERMSIG(status)` donne le numéro du signal qui a causé la fin du fils.
— `WIFSTOPPED(status)` Vrai si le fils est actuellement stoppé. Dans ce cas `WSTOPSIG(status)` donne le numéro du signal qui a causé l'arrêt du fils.
— `WIFCONTINUED(status)` indique si le processus a été continué.

```
1  /* Essais/test-waitpid.c */

   /*
    * essais avec wait pid;
5   *
    * Lance un processus fils qui boucle
    * affiche son numéro
    * surveille son état avec waitpid.
    */
10
   #include <unistd.h>
   #include <stdlib.h>
```

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/wait.h>
#include <stdbool.h>

void travail()
{
    while (true) {
        sleep(1);
    }
}

int main(void)
{
    printf("Lancement\n");
    pid_t pid_fils = fork();

    if (pid_fils == 0) {
        /* le processus pid_fils */
        close (STDIN_FILENO);
        close (STDOUT_FILENO);
        travail();
    }
    printf("* envoyez des signaux au processus %d\n", pid_fils);

    bool fini = false;
    do {
        int status;
        waitpid(pid_fils, &status, 0);

        if (WIFEXITED(status)) {
            printf("fin normale, status = %d\n",
                   WEXITSTATUS(status));
            fini = true;
        }
        if (WIFSIGNALED(status)) {
            printf("terminé à cause signal %d\n",
                   WTERMSIG(status));
            fini = true;
        }
        if (WIFSTOPPED(status)) {
            printf("stoppé à cause signal %d\n",
                   WSTOPSIG(status));
        }
        if (WIFCONTINUED(status)) {
            printf("continué\n");
        }
    } while (! fini);
    return EXIT_SUCCESS;
}
```

5.1.3 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 */

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

   #define CHEMIN_COMPILATEUR     "/usr/bin/gcc"
   #define NOM_COMPILATEUR        "gcc"
   #define TAILLE_MAX_PREFIXE     10
10 #define TAILLE_MAX_NOMFICHIER 100


   int main(void)
   {
15     char prefixe[TAILLE_MAX_PREFIXE];
       char nom_source[TAILLE_MAX_NOMFICHIER];

       char *parametres[] = {
           NOM_COMPILATEUR,
20         NULL, /* emplacement pour le nom du fichier source */
           "-o",
           NULL, /* emplacement pour le nom de l'exécutable   */
           NULL  /* fin des paramètres                        */
       };
25
       printf("préfixe du fichier à compiler : ");
       scanf("%s", prefixe);          /* dangereux */

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

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

35     execv(CHEMIN_COMPILATEUR, parametres);
```

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

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

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

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

   #define CHEMIN_COMPILATEUR     "/usr/bin/gcc"
   #define NOM_COMPILATEUR        "gcc"
   #define TAILLE_MAX_PREFIXE     10
10 #define TAILLE_MAX_NOMFICHIER 100


   int main(void)
   {
15     char prefixe[TAILLE_MAX_PREFIXE];
       char nom_fichier[TAILLE_MAX_NOMFICHIER];

       printf("Préfixe du fichier à compiler : ");
       scanf("%s", prefixe);                    /* dangereux */
20
       snprintf(nom_fichier, TAILLE_MAX_NOMFICHIER,
                "%s.c", prefixe);

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

       perror("execl");            /* on ne passe jamais ici */
30     return EXIT_FAILURE;
   }
```

— `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);

// avec execvp
char * args[] = { "gcc", fichier, "-o", prefixe, NULL };
execlp("gcc", args);
```

5.1.4 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.

5.1.5 Programmation d'un démon

Les *démons*[1] 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 */

   #include <unistd.h>
   #include <stdlib.h>
5  #include <fcntl.h>

   int devenir_demon(void)
   {
       /* Le processus se dédouble, et le père se termine */
10     if (fork() != 0) {
           exit(EXIT_SUCCESS);
       }

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

       /* le processus fils crée le processus démon, et
          se termine */
20     if (fork() != 0) {
           exit(EXIT_SUCCESS);
       }

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

       /* l'entrée standard est redirigée vers /dev/null */
       int fd = open("/dev/null", O_RDWR);
       dup2(fd, STDIN_FILENO);
30     close(fd);
```

1. Traduction de l'anglais *daemon*, acronyme de Disk And Extension MONitor", qui désignait une des parties résidentes d'un des premiers systèmes d'exploitation

```
    /* et les sorties vers /dev/console */
    fd = open("/dev/console", O_WRONLY);
    dup2(fd, STDOUT_FILENO);
35  dup2(fd, STDERR_FILENO);
    close(fd);
    return EXIT_SUCCESS;
}
```

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

5.2 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.

5.2.1 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`.

5.2.2 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.

5.2.3 Exemple

Source :

```
1  /* Threads/leger_mutex.c */

   #include <stdlib.h>
   #include <pthread.h>
5  #include <stdio.h>
   #include <unistd.h>

   struct DonneesTache {
       char * chaine;     // chaine à écrire
10     int nombre;        // nombre de répétitions
       int delai;         // délai entre écritures
   };

   int dernier_numero = 0;   //  variable accédée par les threads
```

```
15 pthread_mutex_t verrou;   //  protège l'accès à la variable numero

   int nouveau_numero();
   void * executer_tache(void * ptr);

20 int main(void)
   {
       // verrou partagé entre les threads
       pthread_mutex_init(&verrou, NULL);

25     struct DonneesTache donnees_tache_1 = {
           .nombre = 4,
           .chaine = "Hello",
           .delai = 2
       };
30     struct DonneesTache donnees_tache_2 = {
           .nombre = 5,
           .chaine = "Hello",
           .delai = 1
       };
35
       pthread_t tache1, tache2;
       pthread_create(&tache1, NULL, executer_tache, &donnees_tache_1);
       pthread_create(&tache2, NULL, executer_tache, &donnees_tache_2);

40     pthread_join(tache1, NULL);
       pthread_join(tache2, NULL);

       pthread_mutex_destroy(&verrou);

45     return EXIT_SUCCESS;
   }

   /**
    * Retourne un nouveau numéro, en utilisant un mutex pour
50  * protéger la section critique.
    * @return un numéro
    */

   int nouveau_numero()
55 {
       pthread_mutex_lock(&verrou);   // début section critique
       int n = ++dernier_numero;
       pthread_mutex_unlock(&verrou); // fin section critique
       return n;
60 }

   void * executer_tache(void * ptr)
   {
       struct DonneesTache *adr_donnees  = ptr;
65
```

```
    for (int k = 0; k < adr_donnees->nombre ; k++) {
        int n = nouveau_numero();
        printf("[%d] %s\n", n, adr_donnees->chaine );

70      sleep(adr_donnees->delai);
    };
    return NULL;
}
```

Compilation :
Sous Linux, les programmes doivent être compilés avec l'option `-pthread` :

```
gcc -std=c18 -g -Wall -pedantic  leger_mutex.c -o leger_mutex -pthread
```

Exécution :

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

5.2.4 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.

5.2.5 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 une 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.

Chapitre 6

IPC : Communication locale entre processus

6.1 Les mécanismes IPC System V

Les mécanismes de communication entre processus (*InterProcess Communication*, ou *IPC*), issus d'Unix System V ont été repris dans de nombreuses variantes d'Unix. Il y a 3 mécanismes, permettant le partage d'informations entre processus tournant sur une même machine :

— les segments, permettant de partager des zones de mémoire
— les sémaphores, qui fournissent un moyen d'en contrôler l'accès,
— les files de messages,

Ces objets sont identifiés par des *clés*.

Pour compiler ces programmes, ajoutez l'option `-D_XOPEN_SOURCE=700` à la compilation

6.2 `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 polet 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')`.

6.3 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  /* IPC/prod.c */

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

   #include <sys/ipc.h>
   #include <sys/shm.h>
10 #include <sys/types.h>
   #include <stdlib.h>
   #include <stdio.h>
   #include <errno.h>
   #include <stdbool.h>
15 #include <assert.h>

   struct Donnees {
       int nb;
       int total;
20 };

   int id_nouveau_segment();

   int main(void)
25 {
```

```
    int id = id_nouveau_segment();
    struct Donnees * adr_donnees
        = shmat(id, NULL, SHM_R | SHM_W);  // attachement au segment
    assert(adr_donnees != NULL);

    adr_donnees->nb = 0;
    adr_donnees->total = 0;

    while (true) {
        printf("+ ");
        int reponse;
        if (scanf("%d", &reponse) != 1) {
            break;
        }
        adr_donnees->nb++;
        adr_donnees->total += reponse;
        printf("sous-total %d=%d\n",
               adr_donnees->nb,
               adr_donnees->total);
    }

    assert (shmdt(adr_donnees) != -1);          // détachement
    assert (shmctl(id, IPC_RMID, NULL) != -1); // suppression

    printf("---\n");
    return EXIT_SUCCESS;
}

int id_nouveau_segment()
{
    key_t cle = ftok(getenv("HOME"), 'A');
    assert(cle != -1);

    int id = shmget(cle, sizeof(struct Donnees),  // création segment
                    IPC_CREAT | IPC_EXCL | 0666);
    if (id == -1) {
        switch (errno) {
        case EEXIST:
            printf("Note: le segment existe déjà\n");
            break;
        default:
            printf("Echec shmget\n");
            break;
        }
        exit(EXIT_FAILURE);
    }
    return id;
}
```

Le consommateur :

```
/* IPC/cons.c */
```

```
/*
   Ce programme affiche périodiquement le contenu de la
   mémoire partagée. Arrêt par Contrôle-C
*/

#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/types.h>
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <errno.h>
#include <signal.h>
#include <stdbool.h>

#include <assert.h>

#define DELAI 2

void preparer_signaux();
void arreter_boucle();

struct Donnees {
    int nb;
    int total;
};

bool continuer_boucle = true;

int id_segment_partage();

int main(void)
{
    preparer_signaux();

    int id = id_segment_partage();

    struct Donnees *adr_donnees = shmat(id, NULL, SHM_R);
    assert (adr_donnees != NULL);

    continuer_boucle = true;
    while (continuer_boucle) {
        sleep(DELAI);
        printf("sous-total %d = %d\n",
               adr_donnees->nb,
               adr_donnees->total);
    }
    printf("---\n");

    assert (shmdt(adr_donnees) != -1);
```

```
    return EXIT_SUCCESS;
}

void arreter_boucle(/* int signal */)
{
    continuer_boucle = false;
}

int id_segment_partage()
{
    key_t cle = ftok(getenv("HOME"), 'A');
    assert(cle != -1);

    int id = shmget(cle, sizeof(struct Donnees), 0);;
    if (id == -1) {
        switch (errno) {
        case ENOENT:
            printf("pas de segment\n");
            break;
        default:
            printf("erreur ouverture segment\n");
            break;
        }
        exit(EXIT_FAILURE);
    }
    return id;
}

void preparer_signaux()
{
    struct sigaction action = {
        .sa_handler = arreter_boucle,
        .sa_flags = 0
    };
    sigemptyset(&action.sa_mask);
    sigaction(SIGINT, &action, NULL);
}
```

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).

6.4 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'indicateurs 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 /* IPC/sem.c */

  /*
     Opérations sur des sémaphores
5 */

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

```c
#include <sys/sem.h>
#include <unistd.h>
#include <stdio.h>
#include <stdlib.h>
#include <ctype.h>
#include <assert.h>
#include <stdbool.h>


typedef int SEMAPHORE;          // identifiant de sémaphore

void detruire_sem(SEMAPHORE sem)
{
    assert(semctl(sem, 0, IPC_RMID, 0) == 0);
}

void changer_sem(SEMAPHORE sem, int val)
{
    struct sembuf sb[1];
    sb[0].sem_num = 0;
    sb[0].sem_op = val;
    sb[0].sem_flg = 0;
    assert(semop(sem, sb, 1) == 0);
}

SEMAPHORE creer_sem(key_t key)
{
    SEMAPHORE sem = semget(key, 1, IPC_CREAT | 0666);
    assert(sem >= 0);
    assert(semctl(sem, 0, SETVAL, 0) >= 0); /* valeur initiale = 0 */
    return sem;
}

void P(SEMAPHORE sem)
{
    changer_sem(sem, -1);
}

void V(SEMAPHORE sem)
{
    changer_sem(sem, 1);
}

/* ---------------------------------------------------- */

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s cle\n", argv[0]);
        return EXIT_FAILURE;
    }
```

```
60      key_t key = atoi(argv[1]);
        SEMAPHORE sem = creer_sem(key);

        bool encore = true;
        while (encore) {
65          char reponse;
            printf("p,v,x,q ? ");
            if (scanf("%c", &reponse) != 1)
                break;
            switch (toupper(reponse)) {
70          case 'P':
                P(sem);
                printf("OK.\n");
                break;
            case 'V':
75              V(sem);
                printf("OK.\n");
                break;
            case 'X':
                detruire_sem(sem);
80              printf("Sémaphore détruit\n");
                encore = 0;
                break;
            case 'Q':
                encore = false;
85              break;
            default:
                printf("?\n");
            }
        }
90      printf("Bye.\n");
        return EXIT_SUCCESS;
    }
```

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.

6.5 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  /* IPC/snd.c */

   /*
      envoi des messages dans une file (IPC System V)
5  */

   #include <errno.h>
   #include <stdio.h>
   #include <stdlib.h>
10 #include <string.h>
   #include <sys/types.h>
   #include <sys/msg.h>
   #include <stdbool.h>
   #include <assert.h>
15
   #define MAX_TEXTE 1000

   struct Message {
       long mtype;
20     char mtext[MAX_TEXTE];
   };


   int main(int argc, char *argv[])
25 {
       if (argc != 3) {
           fprintf(stderr, "Usage: %s clé type\n", argv[0]);
           return EXIT_FAILURE;
       }
30     int cle   = atoi(argv[1]);
       int mtype = atoi(argv[2]);
```

```
    int id    = msgget(cle, 0666);
    assert(id > 0);

    while (true) {
        struct Message message;
        printf("> ");
        fgets(message.mtext, MAX_TEXTE, stdin);
        message.mtype = mtype;

        int l = strlen(message.mtext);
        assert( msgsnd(id, & message, l + 1, 0)  == 0);
    }
}
```

```
/* IPC/rcv.c */

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

#include <errno.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdio.h>

#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <stdbool.h>
#include <assert.h>

#define MAX_TEXTE 1000

struct Message {
    long mtype;
    char mtext[MAX_TEXTE];
};

bool continuer_boucle = true;

int main(int argc, char *argv[])
{
    if (argc != 2) {
        fprintf(stderr, "Usage: %s cle\n", argv[0]);
        return EXIT_FAILURE;
    }
    int cle = atoi(argv[1]);
    int id  = msgget(cle, IPC_CREAT | 0666);
    assert(id != -1);
```

```
    while (continuer_boucle) {
        struct Message message;
        int l = msgrcv(id, &message, MAX_TEXTE, 0L, 0);
        assert (l != -1);
        printf("(type=%ld) %s\n",
               message.mtype, message.mtext);
    }

    return EXIT_SUCCESS;
}
```

Chapitre 7
TCP-IP : communication par le réseau

7.1 Communication par TCP-IP, spécificités

Les concepts fondamentaux de la transmission d'informations par le réseau Internet (sockets, adresses, communication par datagrammes et flots, client-serveur etc.) sont les mêmes que pour les sockets locaux (voir 3.1).

Les spécificités concernent essentiellement l'adressage : comment **fabriquer une adresse** de socket 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.

7.2 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`, ...).

7.2.1 `struct sockaddr` : adresses de sockets

Pour désigner un socket sur une machine il faut une *adresse de socket*. Les fonctions réseau comme `bind`

```
int bind(int socket,
         const struct sockaddr *address,
         socklen_t address_len);
```

prennent commme paramètre une **adresse de socket** désignée par un pointeur.

Terminologie : pour éviter les confusions, dans ce texte
- le terme **adresse** se réfère toujours une **adresse de socket** (concept réseau)
- on parle de **numéro IP** pour une adresse d'une machine (**hôte**), indépendamment du port (**service**),
- on utilise systématiquement le mot **pointeur** pour parler des adresses de données en mémoire (concept du langage C)

Ce pointeur est obtenu par transtypage (cast) d'un pointeur sur une structure contenant une adresse de socket. Le type de la structure peut être différent pour ipv4 ou ipv6.

Les adresses de sockets ont un premier champ [1] de type `sa_family_t`, qui comme son nom l'indique est une indication de la famille d'adresses : constantes `AF_INET` pour ipv4, `AF_INET6` pour ipv6.

On utilisera 3 types de structures pour contenir des adresses
- `struct sockaddr_in`, spécifiquement pour les adresses IPv4 ;
- `struct sockaddr_in6`, pour IPv6 ;
- `struct sockaddr_storage`, assez grande pour contenir n'importe quel type d'adresse ;

et les pointeurs de type `struct sockaddr *` serviront à désigner ces structures de façon générique. [2]

7.2.2 `struct sockaddr_in` : adresses de sockets IPv4

Pour la communication par IP en général, l'adresse d'un socket est formée à partir d'un numéro IP, et un numéro de port (service).

Les `struct sockaddr_in` sont destinées spécifiquement aux adresses IPv4, elles ont 3 champs importants :
- `sin_family`, la famille d'adresses, valant `AF_INET`
- `sin_addr`, pour le numéro IP,
- `sin_port`, pour le numéro de port

Attention, les octets du numéro IP et du 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.

Pour renseigner correctement cette structure, il existe des fonctions de conversion, en particulier `getaddrinfo()` que nous verrons plus loin, et qui assure la *résolution d'adresse*, c'est-à-dire la traduction [3] d'un nom de machine (exemple `www.u-bordeaux.fr`) en numéro IP.

7.2.3 `struct sockaddr_in6` : adresses de socket IPv6

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

Même remarque : on utilisera `getaddrinfo()` pour fabriquer ces adresses de socket.

7.2.4 `struct sockaddr_storage` : conteneur d'adresse

Cette structure est destinée à contenir une adresse de socket réseau de n'importe quel type. Son premier champ `ss_family` contient la famille de l'adresse.

1. qui s'appelle `sa_family` pour `struct sockaddr`
2. C'est une façon courante de réaliser un genre de polymorphisme en C
3. sn consultant éventuellement des serveurs de noms (DNS)

7.3 Remplissage d'une adresse de socket : `getaddrinfo()`

Comme pour les sockets locaux, on crée les sockets par `socket()` et on "nomme la prise" par `bind()`, à partir d'une *adresse de socket* que l'on a renseignée préalablement.

Pour constituer cette adresse, on utilise `getaddrinfo()` !

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

int getaddrinfo(const char *node, const char *service,
                const struct addrinfo *hints,
                struct addrinfo **res);
```

qui retourne une liste d'adresses de sockets IPv4 et/ou IPv6, selon les arguments qui lui sont données :

— `node` et `service` : chaines contenant le nom de la machine et du port,

— un pointeur sur une structure `hints` contenant des indications diverses.

Le résultat est une liste chainée de structures `addrinfo` contenant des adresses de sockets :

```
struct addrinfo {
      int              ai_flags;
      int              ai_family;
      int              ai_socktype;
      int              ai_protocol;
      socklen_t        ai_addrlen;
      struct sockaddr *ai_addr;
      char            *ai_canonname;
      struct addrinfo *ai_next;
};
```

ce type de structure sert aussi pour les indications complémentaires pour la requête (voir plus loin)

Les champs qui nous intéressent, dans les résultats, sont

— `ai_family` qui indique la famille d'adresses : constante `AF_INET` pour ipv4, `AF_INET6` pour ipv6 ;

— `ai_addr` qui est un pointeur sur une adresse de socket

— `ai_addrlen`, la longueur de cette adresse ;

— `ai_next`, un pointeur sur le résultat suivant.

Attention, il faudra libérer la liste de résultats après usage, en appelant `freeaddrinfo()`.

```
void freeaddrinfo(struct addrinfo *res);
```

7.3.1 Préparation d'une adresse distante

Dans le cas le plus fréquent, on cherche à joindre une machine

— dont on connait le nom `"www.elysee.fr"` ou le numéro `"8.253.7.126"` ;

— du *service* que l'on veut joindre, que ce soit par un nom (exemple `"http"`) ou un numéro de port `"80"`.[4]

4. le fichier `/etc/services` des machines Unix contient une table de correspondance entre les noms de services et les numéros de port

Un exemple simple : ouverture d'un socket "flot de données" vers le port 80 de la machine `www.elysee.fr`

```
struct addrinfo * adr_premier;

// récupérer dans adr_premier la première adresse
// qui correspond
getaddrinfo("www.elysee.fr", "http", NULL, & adr_premier);

// utilisation de l'adresse
int fd = socket(adr_premier->ai_family, SOCK_STREAM, 0);
bind(fd, adr_premier->ai_addr, adr_premier->ai_addrlen);

freeaddrinfo(adr_premier);
```

Il faudrait y ajouter des vérifications : échec des fonctions, absece de résultats, etc.
On trouvera une utilisation plus détaillée dans l'exemple `client-echo.c`.

7.3.2 Préparation d'une adresse locale

Dans l'exemple ci-dessus, le troisième paramètre de `getaddrinfo()` est un pointeur nul, mais on peut s'en servir pour transmettre l'adresse d'une structure `addrinfo` qui sert à préciser ce que l'on veut.

Les champs pertinents (les plus intéressants) :

— `ai_family` qui indique la ou les familles d'adresses voulues : constante `AF_INET` pour ipv4, `AF_INET6` pour ipv6, `AF_UNSPEC` (par défaut) pour l'un ou l'autre.
— `ai_socktype` indique si on ne doit rechercher que certains types de sockets (`SOCK_STREAM`, `SOCK_DGRAM`) ou tous (0).
— `ai_flags` : combinaison d'indicateurs divers. Y mettre `AI_PASSIVE` pour un serveur qui doit accepter des connexions sur ses différentes adresses réseau.

Les autres doivent être initialisés, 0 est une valeur par défaut raisonnable.

L'initialisation se fait élégamment grâce aux "Designated Initializers" introduits par la norme C99. Exemple[5]

```
struct addrinfo indications_client = {    // ipv4 ou ipv6
   .ai_family   = AF_UNSPEC,
   .ai_socktype = SOCK_STREAM
};

struct addrinfo indications_serveur = {   // pour un serveur IPv6
   .ai_family = AF_INET6,
   .ai_flags  = AI_PASSIVE
};
```

5. Avec ce type d'initialisation des structures, les champs non spécifiés sont initialisés à 0.

7.3.3 Examen d'une adresse : `getnameinfo()`

La fonction 'getnameinfo()' réalise la conversion dans l'autre sens : à partir d'une adresse de socket réseau, obtenir une description du numéro IP de la machine et du numéro de service/port.

```
int getnameinfo(const struct sockaddr *addr, socklen_t addrlen,
                char *host, socklen_t hostlen,
                char *serv, socklen_t servlen,
                int flags);
```

Les paramètres sont
- un pointeur sur la structure contenant l'adresse, et sa taille
- l'adresse d'un tampon pour y ranger le nom de la machine, et sa longueur
- l'adresse d'un tampon pour le port, et sa longueur.
- une combinaison d'indicateurs : `NI_NUMERICHOST` et `NI_NUMERICSERV`, pour avoir les indications sous forme numérique.

Un exemple : programme de résolution d'adresses.

Le programme qui suit s'exécute en ligne de commande. Il prend en paramètre une série de noms de machines ou d'adresses

Pour chaque machine il affiche les adresses IP numériques sous forme numérique IPv4 (décimal pointé) ou IPv6 (hexadécimal).

```
1  /*
     resolution.c

     Résolution d'adresses
5   */

   #include <unistd.h>
   #include <stdio.h>
   #include <stdlib.h>
10
   #include <sys/socket.h>
   #include <netinet/in.h>
   #include <arpa/inet.h>
   #include <netdb.h>
15 #include <limits.h>

   void resoudre_nom(const char nom_machine[]);

   int main(int argc, char *argv[])
20 {
       if (argc == 1) {
           printf("usage : %s adresses...\n", argv[0]);
           return EXIT_FAILURE;
       }
25     for (int i = 1; i < argc; i++) {
           resoudre_nom(argv[i]);
       }
       return EXIT_SUCCESS;
   }
```

```
30
void resoudre_nom(const char nom_machine[])
{
    printf("* Résolution de %s\n", nom_machine);
    struct addrinfo *adr_liste_adresses;
35
    // le type de socket est précisé pour n'avoir qu'une réponse par numéro IP
    struct addrinfo indications = {
        .ai_family = AF_UNSPEC,
        .ai_socktype = SOCK_STREAM
40  };

    int r = getaddrinfo(nom_machine, NULL,
            &indications,
            & adr_liste_adresses);
45  if (r != 0) {
        printf("- échec de la résolution\n");
        return;
    };
    for (struct addrinfo *ptr = adr_liste_adresses;
50          ptr != NULL;
            ptr = ptr->ai_next) {
        // conversion  adresse -> hote
        char ip_hote[HOST_NAME_MAX]; // taille max FQDN, RFC 1035
        getnameinfo(ptr->ai_addr, ptr->ai_addrlen,
55              ip_hote, sizeof(ip_hote),
                NULL, 0, // port non utilisé
                NI_NUMERICHOST);
        printf("-> %s\n", ip_hote);
    }
60  printf("\n");
    freeaddrinfo(adr_liste_adresses);
}
```

7.3.4 Adresse associée à un socket

A partir d'un "field descriptor" ouvert, ces deux fonctions permettent de retrouver l'adresse
— du socket auquel il est lié (local)
— du socket "pair" auquel il est connecté (distant)

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

int   getsockname(int sockfd,
                  struct sockaddr  * name,
                  socklen_t        * namelen  )
int   getpeername(int sockfd,
                  struct sockaddr  * addr,
                  socklen_t        * addrlen);
```

Paramètres :
— le descripteur,
— un pointeur sur un "conteneur d'adresse de socket" (de préférence une structure `sockaddr_storage` pour avoir la compatibilité ipv4/ipv6),
— un pointeur sur un entier qui recevra la longueur de l'adresse.

7.4 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 : `SHUT_RD` pour l'entrée, `SHUT_WR` pour la sortie, `SHUT_RDWR` pour les deux (équivaut à `close()`).

7.5 Communication par datagrammes (UDP)

7.5.1 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 via Internet, indiquez famille d'adresses, le type `SOCK_DGRAM` et le protocole par défaut `0`.

Retourne `-1` en cas d'échec.

7.5.2 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);
```

7.5.3 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);
```

7.5.4 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 remplir une structure avec l'adresse `from` du socket 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);
```

7.5.5 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  /*
     Echo-Datagrammes/client-echo.c

     Envoi de datagrammes
5
     Exemple de client qui
     - ouvre un socket
     - envoie un datagramme sur ce socket (ligne de texte)
     - attend une réponse
10   - affiche la réponse

     Usage :
```

```
      ./client-echo hote port "chaine de texte"

  L'hote peut être un nom de machine, ou une adresse numérique IPv4/IPv6
  Le port un nom de service ou un numéro
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <arpa/inet.h>

#define TAILLE_TAMPON 1000

int fd_serveur;

void abandon(char message[])
{
    printf("CLIENT> Erreur fatale\n");
    perror(message);
    exit(EXIT_FAILURE);
}

int socket_vers_serveur_datagrammes(const char *serveur, const char *service)
{
    struct addrinfo *adr_liste_adresses;
    int r = getaddrinfo(serveur, service, NULL, & adr_liste_adresses);
    if (r != 0) {
        printf("Erreur %s\n", gai_strerror(r));
        abandon("Echec de résolution d'adresses");
    };
    int fd = -1;

    for (struct addrinfo *ptr = adr_liste_adresses;
            ptr != NULL;
            ptr = ptr->ai_next) {

        fd = socket(ptr->ai_family, SOCK_DGRAM, 0);
        if (fd < 0) {
            continue;
        }
        r = connect(fd, ptr->ai_addr, ptr->ai_addrlen);
        if (r < 0) {
            close(fd);
            fd = -1;
        } else {
```

```
            break; // on a trouvé un socket qui marche.
        }
    }
    freeaddrinfo(adr_liste_adresses);
    return fd;
}

int main(int argc, char *argv[])
{
    /* 1. réception des paramètres de la ligne de commande */
    if (argc != 4) {
        printf("Usage : %s hote port message\n", argv[0]);
        abandon("nombre de paramètres incorrect");
    }
    char * nom_serveur = argv[1];
    char * nom_service = argv[2];
    char * requete = argv[3];

    /* 2. Initialisation du socket */

    fd_serveur = socket_vers_serveur_datagrammes(nom_serveur, nom_service);

    /* 3. Envoi de la requête */
    printf("envoi> %s\n", requete);
    int longueur_requete = strlen(requete) + 1;
    if (send(fd_serveur, requete, longueur_requete, 0) < 0) {
        abandon("Envoi requete");
    };

    /* 4. Lecture de la réponse */
    char reponse[TAILLE_TAMPON];
    int longueur_reponse = recv(fd_serveur, reponse, TAILLE_TAMPON, 0);
    if (longueur_reponse < 0) {
        abandon("Attente réponse");
    }
    printf("réponse> %s\n", reponse);
    close(fd_serveur);

    printf("fin>\n");
    return EXIT_SUCCESS;
}
```

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 :

```
/*
  Echo-Datagrammes/serveur-echo.c -  Réception de datagrammes

  Exemple de serveur qui
  - ouvre un socket (IPV6) sur un port en mode non-connecté
```

```
 /*
  - affiche les messages (chaînes de caractères) qu'il reçoit par ce socket,
 ainsi que leur provenance.
  - envoie une réponse : la chaine en majuscules
 */

#include <unistd.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <arpa/inet.h>
#include <ctype.h>
#include <string.h>
#include <stdbool.h>
#include <netdb.h>

#define FAMILLE    AF_INET6

#define TAILLE_TAMPON 1000
static int fd;

void abandon(char message[])
{
    printf("SERVEUR> Erreur fatale\n");
    perror(message);
    exit(EXIT_FAILURE);
}

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

int socket_serveur_datagrammes(const char *service)
{
    struct addrinfo indications = {
        .ai_family = FAMILLE,
        .ai_flags = AI_PASSIVE
    };

    struct addrinfo *adr_liste_adresses;
    int r = getaddrinfo(NULL, service, & indications, & adr_liste_adresses);
    if (r != 0) {
        printf("Erreur %s\n", gai_strerror(r));
        abandon("Echec de résolution d'adresses");
    };
```

```
    int fd = -1;

    for (struct addrinfo *ptr = adr_liste_adresses;
            ptr != NULL;
            ptr = ptr->ai_next) {
        fd = socket(ptr->ai_family, SOCK_DGRAM, 0);
        if (fd < 0) {
            continue;
        }
        r = bind(fd, ptr->ai_addr, ptr->ai_addrlen);
        if (r < 0) {
            close(fd);
            fd = -1;
        } else {
            break; // on a trouvé un socket qui marche.
        }
    }
    freeaddrinfo(adr_liste_adresses);
    return fd;
}

int fabriquer_reponse(const char requete[], char reponse[])
{
    int taille = 0;
    while (requete[taille] != '\0') {
        reponse[taille] = toupper(requete[taille]);
        taille++;
    };
    return taille;
}

int main(int argc, char *argv[])
{
    /* 1. réception des paramètres de la ligne de commande */

    if (argc != 2) {
        printf("usage: %s port\n", argv[0]);
        abandon("mauvais nombre de paramètres");
    }
    char * nom_service = argv[1];

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

    struct sigaction a = {
        .sa_handler = arreter_serveur,
        .sa_flags = 0,
    };
    sigemptyset(&a.sa_mask);

    sigaction(SIGINT, &a, NULL);
```

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

    int fd = socket_serveur_datagrammes(nom_service);
    if (fd < 0) {
        abandon("socket");
    }

    printf("SERVEUR> Le serveur écoute le port %s\n",
            nom_service);

    while (true) {
        struct sockaddr_storage adresse_client;
        char tampon_requete[TAILLE_TAMPON],
                tampon_reponse[TAILLE_TAMPON];

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

        socklen_t taille_adresse_client = sizeof (adresse_client);
        ssize_t lg_requete = recvfrom(fd, tampon_requete, TAILLE_TAMPON,
                0, /* flags */
                (struct sockaddr *) &adresse_client,
                & taille_adresse_client);
        if (lg_requete < 0) {
            abandon("recvfrom");
        }

        /* 5. Affichage message avec sa provenance et sa longueur */
        char hote_client[INET6_ADDRSTRLEN];
        char port_client[8];
        getnameinfo((struct sockaddr *) & adresse_client, taille_adresse_client,
                hote_client, sizeof (hote_client),
                port_client, sizeof (port_client),
                NI_NUMERICHOST | NI_NUMERICSERV
                );

        printf("%s:%s [%ld]\t: %s\n",
                hote_client, port_client, lg_requete, tampon_requete);

        /* 6. Fabrication d'une réponse */

        int taille_reponse = fabriquer_reponse(tampon_requete, tampon_reponse);

        /* 7. Envoi de la réponse */

        if (sendto(fd,
                tampon_reponse,
                taille_reponse,
                0,
                (struct sockaddr *) &adresse_client,
                sizeof adresse_client) < 0) {
```

```
                abandon("Envoi de la réponse");
160         }
    }
    /* on ne passe jamais ici */
    return EXIT_SUCCESS;
}
```

7.6 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);
```

7.6.1 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()`.

7.6.2 Exemple : client web

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

   /*
      Interrogation d'un serveur web
5
      Usage :
      client-web serveur port adresse-document

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

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

15    Fonctionnement :
      - ouverture d'une connexion TCP vers serveur :port
      - envoi de la requête   GET adresse-document HTTP/1.0[cr][lf][cr][lf]
      - affichage de la réponse
    */
20
   #include <unistd.h>
   #include <sys/types.h>
```

```c
#include <sys/socket.h>
#include <netinet/in.h>
#include <signal.h>
#include <stdio.h>
#include <netdb.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>

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

void abandon(char message[])
{
    perror(message);
    exit(EXIT_FAILURE);
}

/* -- connexion vers un serveur TCP ---------------------------- */

int ouvrir_connexion_tcp(const char hote[], const char service[])
{
    // construction adresse du serveur à contacter

    struct addrinfo indications = {
        .ai_family = AF_UNSPEC,
        .ai_socktype = SOCK_STREAM
    };
    struct addrinfo *premier;
    int r = getaddrinfo(hote, service, & indications, & premier);

    if (r != 0 ) {
        abandon("Adresse serveur incorrecte");
    }

    if (premier == NULL) {
        abandon("Adresse serveur inconnue");
    }
    // création prise, et connexion à l'adresse

    int fd = socket(premier->ai_family, SOCK_STREAM, 0);
    if (fd < 0) {
        abandon("échec création socket");
    }
    if (connect(fd, (struct sockaddr *) premier->ai_addr,
            premier->ai_addrlen) < 0) {
        abandon("Echec connexion");
    }

    freeaddrinfo(premier);
```

```
    return (fd);
}

/* ------------------------------------------------------------ */

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

/* ------------------------------------------------------------ */

void afficher_reponse(int fd)
{
    while (true) {
        /* lecture par bloc */
        char tampon[TAILLE_TAMPON];
        int longueur = read(fd, tampon, TAILLE_TAMPON);
        if (longueur <= 0) {
            break;
        }
        write(1, tampon, longueur); /* copie sur sortie standard */
    };
}

/* -- main ------------------------------------------------------ */

int main(int argc, char *argv[])
{
    /* pour tests :
    char *hote = "127.0.0.1";
    char *service = "80";
    char *adresse_document = "/";
    */

    if (argc != 4) {
        printf("Usage: %s hote service adresse-document\n",
                argv[0]);
        abandon("nombre de paramètres incorrect");
    }

    char *hote = argv[1];
    char *service = argv[2];
    char *adresse_document = argv[3];

    int fd = ouvrir_connexion_tcp(hote, service);
```

```
125     demander_document(fd, adresse_document);
        afficher_reponse(fd);
        close(fd);

        return EXIT_SUCCESS;
130 }
```

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.

7.6.3 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()` retourne un autre socket, qui sert 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 simultanément des connexions venant de plusieurs clients. 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.

Chapitre 8
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.

Attention : ces serveurs ne traitent que les requêtes GET de HTTP/1.0, et ignorent les entêtes HTTP, les "keep-alive", qui gardent les connexions ouvertes etc.

8.1 Serveur Web (avec processus)

8.1.1 Principe et pseudo-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
```

8.1.2 Code du serveur

```
1  /* Serveur-avec-processus/serveur-web.c */

   /* ------------------------------------------------------
      Serveur TCP
5
      serveur web, qui renvoie les fichiers (textes)
      d'un répertoire sous forme de pages HTML

     usage :  serveur-web [-p port] [-d repertoire]
10   exemple: serveur-web -p 8000 -d /usr/doc/exemples
   ------------------------------------------------------*/
```

```c
#include <unistd.h>
#include <sys/socket.h>
#include <sys/wait.h>
#include <netinet/in.h>
#include <arpa/inet.h>
#include <signal.h>
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <netdb.h>
#include <limits.h>

#include "constantes.h"
#include "reseau.h"

#include "traitement-client.h"

/* variable globale, pour partager
          avec traitement signal fin_serveur */

bool serveur_en_marche;

static void arreter_serveur(/* int numero_signal */)
{
    serveur_en_marche = false;
}

static void attendre_sous_serveur(/* int numero_signal */)
{
    /* cette fonction est appelée chaque fois
       qu'un signal SIGCHLD
       indique la fin d'un processus fils _au moins_.  */
    while (waitpid(-1, NULL, WNOHANG) > 0) {
        /* attente des fils arrêtés, tant qu'il y en a */
        continue;
    }
}

static void installer_signaux()
{
    struct sigaction action_int, action_chld;
    /* arrêt du serveur si signal SIGINT */
    action_int.sa_handler = arreter_serveur;
    sigemptyset(&action_int.sa_mask);
    action_int.sa_flags = 0;
    sigaction(SIGINT, &action_int, NULL);

    /* attente fils si SIGCHLD */
    action_chld.sa_handler = attendre_sous_serveur;
    sigemptyset(&action_chld.sa_mask);
```

```
    action_chld.sa_flags = SA_NOCLDSTOP;
    sigaction(SIGCHLD, &action_chld, NULL);
}

static void afficher_informations_client(int fd_client, int numero_client)
{
    struct sockaddr_storage adresse_client;
    socklen_t long_adresse_client = sizeof adresse_client;

    char hote[HOST_NAME_MAX];
    char service[20];

    getpeername(fd_client,
            (struct sockaddr *) &adresse_client, &long_adresse_client);
    getnameinfo((struct sockaddr *)& adresse_client,
            long_adresse_client,
            hote, sizeof(hote),
            service, sizeof(service),
            NI_NUMERICHOST | NI_NUMERICSERV);

    printf("> client %d : %s [%s]\n", numero_client, hote, service);
}

static void demarrer_serveur(const char port[], char repertoire[])
{
    installer_signaux();

    serveur_en_marche = true;
    int fd_serveur = ouvrir_serveur_tcp(port);

    printf("> Serveur " VERSION
           " (port=%s, répertoire de documents=\"%s\")\n",
           port, repertoire);

    for (int numero_client = 1; serveur_en_marche; numero_client++) {
        int fd_client = ouvrir_client(fd_serveur);
        afficher_informations_client(fd_client, numero_client);
        if (fork() == 0) {
            close(fd_serveur);
            servir_client(fd_client, repertoire);
            close(fd_client);
            exit(EXIT_SUCCESS);
        }
        close(fd_client);
    }
    shutdown(fd_serveur, SHUT_RDWR);
    printf("> Arrêt du serveur\n");
}

/* ---------------------------------------------------------------------*/
```

```
static void usage(char prog[])
{
    printf("Usage : %s [options\n\n", prog);
    printf("Options :"
           "-h\tcemessage\n"
           "-p port\tport du serveur          [%s]\n"
           "-d dir \trépertoire des documents [%s]\n",
           PORT_PAR_DEFAUT, REPERTOIRE_PAR_DEFAUT);
}

/* ------------------------------------------------------------------*/
int main(int argc, char * argv[])
{
    char *port = PORT_PAR_DEFAUT;
    char *repertoire = REPERTOIRE_PAR_DEFAUT; /* la racine
                                                 des documents */
    char c;
    while ((c = getopt(argc, argv, "hp:d:")) != -1)
        switch (c) {
        case 'h':
            usage(argv[0]);
            exit(EXIT_SUCCESS);
            break;
        case 'p':
            port = optarg;
            break;
        case 'd':
            repertoire = optarg;
            break;
        case '?':
            fprintf(stderr, "Option inconnue -%c. "
                    "-h pour aide.\n",
                    optopt);
            break;
        };
    demarrer_serveur(port, repertoire);
    exit(EXIT_SUCCESS);
}
```

8.1.3 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.

8.2 Serveur Web (avec threads)

8.2.1 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
```

8.2.2 Code du serveur

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

   /* ------------------------------------------------------
      Serveur TCP
5
      serveur web, qui renvoie les fichiers (textes)
      d'un répertoire sous forme de pages HTML

     usage :  serveur-web [-p port] [-d repertoire]
10   exemple: serveur-web -p 8000 -d /usr/doc/exemples

     Version basée sur les threads. Au lieu de créer
     un processus par connexion, on gère un pool de tâches
     (sous-serveurs).
15   - au démarrage du serveur les sous-serveurs sont créés,
     et bloqués par un verrou
     - quand un client se connecte, la connexion est
     confiée à un sous-serveur inactif, qui est débloqué
     pour l'occasion.
20 ------------------------------------------------------*/

   #include <pthread.h>
   #include <unistd.h>
   #include <sys/types.h>
25 #include <sys/errno.h>
   #include <sys/socket.h>
   #include <sys/wait.h>
   #include <sys/stat.h>
   #include <netinet/in.h>
30 #include <arpa/inet.h>
   #include <signal.h>
   #include <stdio.h>
```

```
#include <stdlib.h>
#include <stdbool.h>
#include <netdb.h>
#include <bits/local_lim.h>

#include "constantes.h"
#include "reseau.h"
#include "traitement-client.h"

#define NB_SOUS_SERVEURS 5

struct DonneesThread {
    pthread_t id; //  identificateur de thread
    pthread_mutex_t verrou;
    bool actif; //  sous-serveur occupé
    int fd; // socket du client
    char *racine_documents;
};

// variables globales
struct DonneesThread threads[NB_SOUS_SERVEURS];
bool serveur_en_marche;

/* ------------------------------------------------------------------
 Traitement des signaux
------------------------------------------------------------------ */

void arreter_serveur(/* int numero_signal */)
{
    serveur_en_marche = false;
}

/* ------------------------------------------------------------
   sous_serveur
   ------------------------------------------------------------ */
void *executer_sous_serveur(void *ptr)
{
    struct DonneesThread *adr_donnees = ptr;
    while (true) {
        pthread_mutex_lock(&adr_donnees->verrou);
        servir_client(adr_donnees->fd,
                adr_donnees->racine_documents);
        close(adr_donnees->fd);
        adr_donnees->actif = false;
    }
    return NULL; /* jamais exécuté */
}

/* ------------------------------------------------------------ */
void creer_sous_serveurs(char repertoire[])
{
```

```
    for (int k = 0; k < NB_SOUS_SERVEURS; k++) {
        struct DonneesThread *adr_donnees = &(threads[k]);
        adr_donnees->actif = false;
        adr_donnees->racine_documents = repertoire;
        pthread_mutex_init(&adr_donnees->verrou, NULL);
        pthread_mutex_lock(&adr_donnees->verrou);
        pthread_create(&adr_donnees->id, NULL,
                executer_sous_serveur, adr_donnees);
    }
}

void supprimer_sous_serveurs()
{
    for (int k = 0; k < NB_SOUS_SERVEURS; k++) {
        pthread_kill(threads[k].id, SIGKILL);
    }
}

void affecter_sous_serveur(int numero_client, int fd_client)
{
    struct sockaddr_storage a;
    socklen_t l = sizeof a;
    getsockname(fd_client, (struct sockaddr *) &a, &l);
    char ip[HOST_NAME_MAX], port[20];
    getnameinfo((struct sockaddr *) & a, l,
        ip, sizeof(ip),
        port, sizeof(port),
            NI_NUMERICHOST | NI_NUMERICSERV);

    /* recherche d'un sous-serveur inoccupé */
    for (int k = 0; k < NB_SOUS_SERVEURS; k++) {
        if (!threads[k].actif) {
            // affectation du travail et déblocage
            printf("> client %d [%s port %s] traité par sous-serveur %d\n",
                    numero_client, ip, port, k);
            threads[k].fd = fd_client;
            threads[k].actif = true;
            pthread_mutex_unlock(&threads[k].verrou);
            return;
        }
    }
    // tout le monde est occupé...
    printf("> client %d [%s port %s] rejeté (surcharge)\n",
            numero_client, ip, port);
    close(fd_client);
}

/* ------------------------------------------------------
   demarrer_serveur: crée le socket serveur
   et lance des processus pour chaque client
   ------------------------------------------------------ */
```

```
135
void demarrer_serveur(const char port[], char racine_documents[])
{
    printf("> Serveur " VERSION "+threads "
           "(port=%s, répertoire de documents=\"%s\")\n",
140        port, racine_documents);

    /* signal SIGINT -> arrêt du serveur */
    struct sigaction action_fin = {
        .sa_flags = 0
145 };
    sigemptyset(&action_fin.sa_mask);
    sigaction(SIGINT, &action_fin, NULL);

    /* création du socket serveur et du pool de sous-serveurs */
150 serveur_en_marche = true;
    int fd_serveur = ouvrir_serveur_tcp(port);
    creer_sous_serveurs(racine_documents);

    /* boucle du serveur */
155 int numero_client = 0;
    while (serveur_en_marche) {
        int fd_client = ouvrir_client(fd_serveur);
        numero_client++;
        affecter_sous_serveur(numero_client, fd_client);
160 }
    printf("=> fin du serveur\n");
    supprimer_sous_serveurs();
    shutdown(fd_serveur, SHUT_RDWR); /* utile ? */
    close(fd_serveur);
165 }

/* ------------------------------------------------------------------ */
void usage(char prog[])
{
170 printf("Usage : %s [options\n\n", prog);
    printf("Options :"
           "-h\tcemessage\n"
           "-p port\tport du serveur          [%s]\n"
           "-d dir \trépertoire des documents [%s]\n",
175        PORT_PAR_DEFAUT, REPERTOIRE_PAR_DEFAUT);
}

/* ------------------------------------------------------------------ */
int main(int argc, char *argv[])
180 {
    char *port = PORT_PAR_DEFAUT;
    char *racine_documents
           = REPERTOIRE_PAR_DEFAUT; /*  racine des documents */
    char c;
185
```

```
    while ((c = getopt(argc, argv, "hp:d:")) != -1)
        switch (c) {
            case 'h':
                usage(argv[0]);
                exit(EXIT_SUCCESS);
                break;
            case 'p':
                port = optarg;
                break;
            case 'd':
                racine_documents = optarg;
                break;
            case '?':
                fprintf(stderr,
                        "Option inconnue -%c. -h pour aide.\n", optopt);
                break;
        }
    demarrer_serveur(port, racine_documents);
    exit(EXIT_SUCCESS);
}
```

8.2.3 Discussion de la solution

Les inconvénients de cette solution sont 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 (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).

8.3 Parties communes aux deux serveurs

8.3.1 Déclarations et entêtes de fonctions

```
/* Serveur Web - constantes.h */

#ifndef CONSTANTES_H
#define CONSTANTES_H

#define CRLF "\r\n"
#define VERSION "MegaSoft 2019.20.08 pour Unix"
#define CHARSET "utf-8"
```

```
#define PORT_PAR_DEFAUT            "8000"
#define REPERTOIRE_PAR_DEFAUT  "/tmp"
#endif
```

8.3.2 Les fonctions réseau

— `serveur_tcp()` : création du socket du serveur TCP.
— `attendre_client()`

```
/*
  Projet serveurs Web - reseau.c
  Fonctions réseau
 */

#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <stdbool.h>

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


#include "reseau.h"

/**
 *  démarre un service TCP sur le port indiqué
 */

int ouvrir_serveur_tcp(const char port[])
{
    struct addrinfo *adr_premier;
    struct addrinfo indications = {
        .ai_family = AF_INET6,
        .ai_flags = AI_PASSIVE
        };
    int r = getaddrinfo(NULL, port, &indications, &adr_premier);
    if (r != 0) {
        perror("resolution nom de service");
        exit(EXIT_FAILURE);
    }
    int fd = -1;

```

```
    for (struct addrinfo * ptr = adr_premier; ptr != NULL; ptr = ptr->ai_next) {
        fd = socket(ptr->ai_family, SOCK_STREAM, 0);
        if (fd < 0) {
            continue;
        }
        r = bind(fd, ptr->ai_addr, ptr->ai_addrlen);
        if (r < 0) {
                close(fd);
                fd = -1;
            } else {
                break; // on a trouvé un socket qui marche .
            }
        }
        freeaddrinfo(adr_premier);
        listen(fd, 4);
        return fd;
    }

    /**
     * retourne le file-descriptor du prochain
     * client connecté sur le serveur
     */
    int ouvrir_client(int fd_serveur)
    {
        /* A cause des signaux SIGCHLD, la fonction accept()
           peut être interrompue quand un fils se termine.
           Dans ce cas, on relance accept().
         */
        while (true) {
            int fd_client = accept(fd_serveur, NULL, NULL);
            if (fd_client > 0) {
                return fd_client;
            }
            if (errno != EINTR) {
                perror("attente client");
                exit(EXIT_FAILURE);
            };
        };
    }
```

8.3.3 Les fonctions de dialogue avec le client

- `dialogue_client()` : lecture et traitement de la requête d'un client
- `envoyer_document()`,
- `document_non_trouve()`,
- `requete_invalide()`.

```
1 /*
  Serveurs-Web/traitement-client.c
```

```c
  projet serveur WEB
  Communication avec un client
 */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <stdbool.h>

#include "constantes.h"
#include "traitement-client.h"

void afficher_entete(FILE* in)
{
    char *ligne = NULL;
    size_t taille_ligne = 0L;
    while (true) {
        ssize_t n = getline(& ligne, &taille_ligne, in);
        printf("-- %s", ligne);
        if (n <= 2) break;
    }
    free(ligne);
}

void servir_client(int fd_client, const char *racine_documents)
{
    FILE *in = fdopen(fd_client, "r");
    // si on attache deux FILEs au même descripteur,
    // la fermeture de l'un entraine la fermeture de l'autre.
    // Ici on veutDonc on duplique.
    FILE *out = fdopen(dup(fd_client), "w");

    /* lecture de la première ligne de requête,
     * du genre "GET quelquechose ..." */
    char *ligne = NULL;
    size_t taille_ligne = 0L;
    getline(& ligne, &taille_ligne, in);
    printf("> %s\n", ligne);
    afficher_entete(in);
    char verbe[100], nom_document[100];
    sscanf(ligne,"%100s %100s", verbe, nom_document);
    free(ligne);


    if (strcmp(verbe, "GET") == 0) {
        retourner_document(out, nom_document, racine_documents);
    } else {
        repondre_erreur_400(out);
    }
```

```
    fflush(out);
    fclose(in);
    fclose(out);
}

void transcoder_contenu_fichier(FILE *out, FILE *fichier)
{
    while (true) {
        int c = fgetc(fichier);
        if (c == EOF || c < 0) break;
        switch (c) {
        case '<':
            fputs("&lt;", out);
            break;
        case '>':
            fputs("&gt;", out);
            break;
        case '&':
            fputs("&amp;", out);
            break;
        case '\n':
            fputs(CRLF, out);
            break;
        default:
            fputc(c, out);
        };
    }
    /* balises de fin */
    fputs("</listing></table></center></body></html>"
          CRLF, out);
}

void retourner_document(FILE *out,
                        const char *nom_document,
                        const char *racine_documents)
{
    char nom_fichier[200];
    snprintf(nom_fichier, 200,
             "%s%s", racine_documents, nom_document);

    if (strstr(nom_fichier, "/../") != NULL) {
        /* tentative d'accès hors du répertoire ! */
        repondre_erreur_404(out, nom_document);
        return;
    };
    FILE * fichier = fopen(nom_fichier, "r");
    if (fichier == NULL) {
        repondre_erreur_404(out, nom_document);
        return;
    };
```

```
105
    fprintf(out,
            "HTTP/1.1 200 OK" CRLF
            "Server: " VERSION CRLF
            "Content-Type: text/html; " CHARSET CRLF
110         CRLF);

    fprintf(out,
            "<!doctype html>" CRLF
            "<html><head>" CRLF
115         "<meta charset=\"" CHARSET "\">" CRLF
            "<title>Fichier %s</title></head>" CRLF
            "<body bgcolor=\"white\">"
            "<h1>Fichier %s</h1>" CRLF
            "<center><table>"
120         "<tr><td bgcolor=\"yellow\"></td></tr>"
            "<listing>" CRLF,
            nom_document, nom_fichier);

    transcoder_contenu_fichier(out, fichier);
125 }

    void repondre_erreur_404(FILE *out, const char *nom_document)
    {
        /* envoi de la réponse : entête */
130     fprintf(out,
            "HTTP/1.1 404 Not Found" CRLF
            "Server: " VERSION CRLF
            "Content-Type: text/html; charset=" CHARSET CRLF
            CRLF);
135
        /* corps de la réponse */
        fprintf(out,
            "<!doctype HTML>" CRLF
            "<HTML><HEAD>" CRLF
140         "<meta charset=\"" CHARSET "\">" CRLF
            "<TITLE>404 Not Found</TITLE>" CRLF
            "</HEAD><BODY BGCOLOR=\"yellow\">" CRLF
            "<H1>Pas trouvé !</H1>" CRLF
            "Le document <font color=\"red\"><tt>%s</tt></font> "
145         "demandé<br>n'est pas disponible.<P>" CRLF
            "<hr> Le webmaster"
            "</BODY></HTML>" CRLF,
            nom_document);
        fflush(out);
150 }

    void repondre_erreur_400(FILE *out)
    {
        fprintf(out,
155             "<!doctype html>" CRLF
```

```
                "<HTML><HEAD>" CRLF
                "<meta charset=\"" CHARSET "\">" CRLF
                "<TITLE>400 Bad Request</TITLE>" CRLF
                "</HEAD><BODY BGCOLOR=\"yellow\">" CRLF
160             "<H1>Bad Request</H1>"
                "Vous avez envoyé une requête que "
                "ce serveur ne comprend pas." CRLF
                "<hr>Le webmaster"
                "</BODY></HTML>" CRLF);
165 }
```

8.3.4 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.

Annexe A
Transmission d'un descripteur

L'exemple ci-dessous montre comment transmettre un descripteur d'un processus à un autre.

```
1
   /*
    exemple adapté de
    http ://www.mail-archive.com/
5   linux-development-sys@senator-bedfellow.mit.edu/msg01240.html
    (remplacement de alloca par malloc + modifications mineures)
    */

   /* passfd.c -- sample program which passes a file descriptor */
10
   /* We behave like a simple /bin/cat, which only handles one argument
    *    (a file name). We create Unix domain sockets through
    *    socketpair(), and then fork(). The child opens the file whose
    *    name is passed on the command line, passes the file descriptor
15  *    and file name back to the parent, and then exits. The parent
    *    waits for the file descriptor from the child, then copies data
    *    from that file descriptor to stdout until no data is left. The
    *    parent then exits. */

20 #include <malloc.h>
   #include <fcntl.h>
   #include <stdio.h>
   #include <string.h>
   #include <sys/socket.h>
25 #include <sys/uio.h>
   #include <sys/un.h>
   #include <sys/wait.h>
   #include <unistd.h>
   #include <stdlib.h>
30
   void die (char *message)
   {
       perror(message);
       exit(EXIT_FAILURE);
35 }
```

```
/* Copies data from file descriptor 'from' to file descriptor 'to'
 *    until nothing is left to be copied.
 Exits if an error occurs.   */
void copyData (int from, int to)
{
    char buf[1024];
    int amount;
    while ( (amount = read(from, buf, sizeof(buf))) > 0) {
        if (write(to, buf, amount) != amount) {
            die("write");
        }
    }
    if (amount < 0)
        die("read");
}

/* The child process. This sends the file descriptor. */
int childProcess (char *filename, int sock)
{
    int fd = open(filename, O_RDONLY);

    /* Open the file whose descriptor will be passed. */
    if (fd < 0) {
        perror("open");
        return EXIT_FAILURE;
    }


    /* Send the file name down the socket, including the trailing '\0' */
    struct iovec vector = {     /* some data to pass w/ the fd */
        .iov_base = filename,
        .iov_len = strlen(filename) + 1
    };

    /* Put together the first part of the message.
       Include the file name iovec */
    struct msghdr msg = {       /* the complete message */
        .msg_name = NULL,
        .msg_namelen = 0,
        .msg_iov = &vector,
        .msg_iovlen = 1
    };
    /* Now for the control message. We have to allocate room for
     *        the file descriptor. */

    struct cmsghdr *cmsg;       /* the control message,
                                   which will include the fd */
    size_t len = sizeof(struct cmsghdr) + sizeof(fd);
    cmsg = malloc(len);
    cmsg->cmsg_len = len;
```

```
    cmsg->cmsg_level = SOL_SOCKET;
    cmsg->cmsg_type = SCM_RIGHTS;

    /* copy the file descriptor onto the end of
    the control message */
    memcpy(CMSG_DATA(cmsg), &fd, sizeof(fd));

    msg.msg_control = cmsg;
    msg.msg_controllen = cmsg->cmsg_len;

    if (sendmsg(sock, &msg, 0) != (ssize_t) vector.iov_len) {
        die("sendmsg");
    }

    free(cmsg);
    return EXIT_SUCCESS;
}

/* The parent process. This receives the file descriptor. */
int parentProcess (int sock)
{
    char buf[80];               /* space to read file name into */

    struct cmsghdr *cmsg;       /* control message with the fd */
    int fd;

    /* set up the iovec for the file name */
    struct iovec vector = {     /* file name from the child */
        .iov_base = buf,
        .iov_len = 80,
    };

    /* the message we're expecting to receive */
    struct msghdr msg = {       /* full message */
        .msg_name = NULL,
        .msg_namelen = 0,
        .msg_iov = &vector,
        .msg_iovlen = 1
    };;

    /* dynamically allocate so we can leave room for the file
     *        descriptor */
    cmsg = malloc(sizeof(struct cmsghdr) + sizeof(fd));
    cmsg->cmsg_len = sizeof(struct cmsghdr) + sizeof(fd);
    msg.msg_control = cmsg;
    msg.msg_controllen = cmsg->cmsg_len;

    if (!recvmsg(sock, &msg, 0)) {
        return EXIT_FAILURE;
    }
```

```
    printf("got file descriptor for '%s'\n", (char *) vector.iov_base);

    /* grab the file descriptor from the control structure */
    memcpy(&fd, CMSG_DATA(cmsg), sizeof(fd));
    copyData(fd, 1);
    free(cmsg);

    return EXIT_SUCCESS;
}

int main (int argc, char **argv)
{
    int socks[2];
    int status;

    if (argc != 2) {
        fprintf(stderr, "only a single argument is supported\n");
        return 1;
    }

    /* Create the sockets. The first is for the parent and the
     *       second is for the child (though we could reverse that
     *       if we liked. */
    if (socketpair(PF_UNIX, SOCK_STREAM, 0, socks)) {
        die("socketpair");
    }

    if ( fork() == 0 ) { /* child */
        close(socks[0]);
        return childProcess(argv[1], socks[1]);
    }

    /* parent */
    close(socks[1]);
    parentProcess(socks[0]);

    wait(&status);

    if (WEXITSTATUS(status)) {
        fprintf(stderr, "child failed\n");
    }

    return EXIT_SUCCESS;
}
```

Annexe B
Documentation

B.1 Documents

Les documents suivants ont été très utiles (il y a longtemps) pour la rédaction de ce texte et la programmation des exemples :

— *Unix Frequently Asked Questions* `http://www.faqs.org/faqs/unix-faq/faq/`
— *Advanced Unix Programming*, Marc J. Rochkind, Prentice-Hall Software Series (1985). ISBN 0-13-011800-1. Accessible en ligne ici `https://kupdf.net/download/advanced-unix-programming-2nd-edition_5b076dbce2b6f5de1ce9ea25_pdf`
— *The GNU C Library Reference Manual*, Sandra Loosemore, Richard M. Stallman, Roland McGrath, Andrew Oram and Urlruch Drepper. `https://www.gnu.org/software/libc/manual/pdf/libc.pdf`
— *What is multithreading ?*, Martin McCarthy, Linux Journal 34, Février 1997, pages 31 à 40. Accessible sur `https://www.linuxjournal.com/article/1363`
— *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` (n'existe plus).

B.2 Standard C

Le standard C 18 :

— *C18 Draft Standard* `http://www.open-std.org/jtc1/sc22/wg14/www/docs/n2479.pdf`

Index