Entrées / sorties

Haut niveau

Ouverture/Fermeture

fopen

Ouverture d'un fichier dont le nom est donné par `path`

```
// Signature de la fonction
FILE *fopen(const char *path, const char *mode);
```

Paramètres

path → chemin d'accès au fichier

mode → indique le type d'opération à effectuer sur le fichier :

- `"r"` : lecture (read) [le fichier doit exister]
- `"w"` : écriture (write) [le fichier est créé s'il n'existe pas, tronqué s'il existe]
- `"r+"` lecture + écriture
- `"w+"` écriture + lecture
- `"a"` écriture en ajout (on écrit à la fin du fichier)
- `"a+"` écriture en ajout + lecture

Retour

Retourne un descripteur de fichier ou `NULL`

fdopen

Permet de construire un descripteur de fichier haut niveau à partir d'un descripteur de fichier bas niveau.

```
// Signature de la fonction
FILE *fdopen(int fd, const char *mode);
```

freopen

Ouvre le fichier donné par path en réutilisant le descripteur stream => utile par exemple pour rediriger les flux standards

```
// Signature de la fonction
FILE *freopen(const char *path, const char *mode, FILE *stream);
```

fclose

```
// Signature de la fonction
int fclose(FILE *stream);
```

Paramètre

stream → Le descripteur de fichier à fermer.

Retour

- 0 en cas de succès
- `EOF` en cas d'échec

Lecture écriture

fread

```
// Signature de la fonction
size_t fread(void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Paramètre

ptr → Zone mémoire ou on écrit les données lues

size → taille d'un objet à lire

nmemb → nombre d'objets à lire

stream → Descripteur ou les données sont lues

Retour

- le nombre d'objets réellement lu < nmemb si fin du fichier ou si erreur. Utiliser `feof(FILE *stream)` qui retourne 0 si on est à la fin du fichier

fwrite

```
// Signature de la fonction
size_t fwrite(const void *ptr, size_t size, size_t nmemb, FILE *stream);
```

Paramètre

ptr → Zone mémoire ou on lit les données à écrire

size → taille d'un objet à écrire

nmemb → nombre d'objets à écrire

stream → Descripteur ou les données sont écrites

Retour

- le nombre d'objets réellement ecrit < nmemb si fin du fichier ou si erreur. Utiliser `feof(FILE *stream)` qui retourne 0 si on est à la fin du fichier

Entrées/Sorties Formatées

Écriture

```
int printf(const char *format, ...);
int fprintf(FILE *stream, const char *format, ...);
int dprintf(int fd, const char *format, ...);
int sprintf(char *str, const char *format, ...);
int snprintf(char *str, size_t size, const char *format, ...);
```

Paramètres :

- `%s` → `char*`
- `%d,i` → `int`
- `%x,X` → int to hexadecimal
- `%f` → `double`

Par caractère :

```
int fputc(int c, FILE *f);
int putc(int c, FILE *f);
int putchar(int c);
```

Lecture

```
int scanf(const char *format, ...); //lecture depuis stdin
int fscanf(FILE *stream, const char *format, ...); //lecture depuis stream
int sscanf(const char *str, const char *format, ...);// lecture depuis chaîne str
```

Par caractère :

```
int fgetc(FILE *f);
int getc(FILE *f);
int getchar(void);
```

Exemple

```
int main(void) {
  FILE* f;
  printf("Message avant \n"); //sur l'ecran de la console
  f = freopen("/tmp/toto.txt", "w", stdout); //mis en place de redirection
  // arrêt en cas d'erreur
  if(f == NULL) {
    perror("fopen");
    return -1;
  }
  printf("Message après \n");//dans le fichier "/tmp/toto.txt"
  //Le fichier est ouvert
  if(fclose(f) != 0) {
    perror("erreur fermeture");
    return -1;
  }
  return 0;
}
```

Bas niveau

open

```
int open(const char * pathname, int flags); // Ouvre un fichier existant
int open(const char * pathname, int flags, mode_t mode); // Ouvre un fichier
int creat(const char * pathname, mode_t mode);
```

Paramètres :

- pathname → Chemin du fichier à ouvrir
- flags
  - `O_APPEND` → les écritures se font à la fin du fichier (ajout),
  - `O_CREAT` → le fichier est créé s'il n'existe pas,
  - `O_EXCL` → échec si le fichier existe déjà,
  - `O_TRUNC` → si le fichier existe, il est tronqué)
- mode (obligatoire avec `O_CREAT`):
  - **S_IRWXU** 00700 user (file owner) has read, write, and execute permission
  - **S_IRUSR** 00400 user has read permission
  - **S_IWUSR** 00200 user has write permission
  - **S_IXUSR** 00100 user has execute permission
  - **S_IRWXG** 00070 group has read, write, and execute permission

- **S_IRGRP** 00040 group has read permission
- **S_IWGRP** 00020 group has write permission
- **S_IXGRP** 00010 group has execute permission
- **S_IRWXO** 00007 others have read, write, and execute permission
- **S_IROTH** 00004 others have read permission
- **S_IWOTH** 00002 others have write permission
- **S_IXOTH** 00001 others have execute permission

close

Permet de fermer le descripteur de fichier `fd`

```
int close(int fd);
```

Retourne :

- 0 si succès
- -1 si erreur

read

```
ssize_t read(int fd, void *buf, size_t count);
```

Paramètres :

- fd → le descripteur de ficher
- buf → le pointeur du buffer ou on stock ce qu'on lit
- count → le nombre à lire

Retourne :

- -1 si erreur
- le nombre de bytes lus : 0 = fin du fichier

write

```
ssize_t write(int fd, const void *buf, size_t count);
```

Paramètres :

- fd → le descripteur de ficher
- buf → le pointeur du buffer ou on stock ce qu'on vas écrire
- count → le nombre à écrire

Retourne :

- -1 si erreur
- le nombre de bytes écrit : 0 = fin du fichier

Exemple en bas niveau :

```
#define TAILLE_TAMPON 4096

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

    const char *nom = argv[1];
    char tampon[TAILLE_TAMPON];
    int fich, nlus, necrits;
    int total = 0;
    fich = open(nom, O_RDONLY);
    if (fich < 0) {
        perror("Ouverture du fichier");
        return EXIT_FAILURE;
    }
    do {
        nlus = read(fich, tampon, TAILLE_TAMPON);
        if (nlus < 0) {
            perror("Erreur de lecture");
            close(fich);
            return EXIT_FAILURE;
        }
        total += nlus;
        necrits = write(1, tampon, nlus);
        if (necrits != nlus)
            fprintf(stderr, "%d caractères écrits au lieu de %d lus\n",
                    necrits, nlus);
    } while (nlus != 0);
    fprintf(stderr, "%d caractères du fichier %s ont été copiés\n",
            total, nom);
    close(fich);
    return EXIT_SUCCESS;
}
```

Redirection

Haut niveau :

freopen

Exemple :

```
int main(void) {
  FILE *redir;
  fprintf(stdout, "Sortie standard\n");
  fprintf(stderr, "Sortie d'erreur\n");
  redir = freopen("/tmp/stdout.txt", w, stdout);
  if (redir == NULL){
    perror("Erreur freopen");
  }
  fprintf(stdout, "Sortie standard APRES\n");
  fprintf(stderr, "Sortie d'erreur APRES\n");
}
```

Bas niveau :

Mise en oeuvre en 4 étapes:

1. créer [Si besoin] un descripteur de fichier (open par exemple)
2. fermer le descripteur qu'on souhaite rediriger (close)
3. dupliquer le descripteur du point 1
4. fermer le descripteur ouvert au point 1 [Facultatif]

Exemple :

```
int main(void) {
  int fd = open("/tmp/stdout.txt", O_WRONLY|O_CREATE|O_TRUNC, S_IRUSR|S_IWUSR);
  // traitement des erreurs

  close(1);//supprimer avec dup2
  if (dup(fd) < 0) {//avec dup2 : dup2(fd, 1)
    //erreurs
  }
  close(fd);
  printf("Sur la sortie standard\n");
}
```

→Les descripteur de fichier ouverts sont conserver par fork. Ils sont aussi conserver par `execve` (sauf ceux ouverts avec O_CLOEXEC)

→on peut donc utiliser ça pour rediriger les flux standards d'un nouveau processus

```
int main(void) {
  int entree = open("/etc/motd", O_RDONLY);
  assert(entree >= 0);
  int sortie = creat("/tmp/motd.copy",  S_IRUSR|S_IWUSR);
  assert(sortie >= 0);
  close(STDIN_FILENO);
  if(dup(entree) < 0 ){
    //erreurs
  }
  close(STDOUT_FILENO);
  if (dup(sortie) < 0) {
    //erreurs
  }
  close(entree);
  close(sortie);

  execlp("cat", "cat", (char*)NULL);
  perror("execlp")
}
```

Signaux

un signal est une info élémentaire envoyée à un processus pour lui signaler un événement:

- fin d'un fils
- erreur arithmétique
- demande d'arrêt
- demande de suspension, de reprise
- ...

un signal peut être envoyé:

- par le système d'exploitation en cas de fautes (par ex: SIGSEGV)
- par un autre processus grâce à la fonction int kill(pid_t pid, int sig);
- par l'utilisateur avec la commande `$kill [options] PID [...]`

Ça peut :

- terminer le processus (ex: SIGTERM, SIGINT[ctrl+C], SIGHUP, SIGKILL, ...)
- terminer le processus avec un fichier core (ex: SIGSEGV, SIGABRT, SIGQUIT[ctrl+/])
- être ignoré (ex: SIGCHILD, ...)

- suspendre le processus (ex: SIGSTOP, SIGTSTP[ctrl+Z], ...)
- continuer l'exécution d'un processus (SIGCONT)

Le comportement peut être changé (sauf pour SIGKILL et SIGSTOP)

On peut :

- ignorer le signal(SIG_IGN)
- retrouver le comportement par défaut(SIG_DFL)
- exécuter une fonction utilisateur

Pour changer le comportement à réception d'un signal on peut utiliser les fonctions `signal(2)` ou `sigaction(2)`

signal

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

sigaction

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

cf. exemple après les processus

Processus

Création

```
pid_t fork(); // créer un processus, -1 = erreur, 0 = fils, reste = père
```

Attente

Attente de la fin d'un processus

```
pid_t wait(int *status);
```

Pour vérifier le code de retour du processus, on peut utiliser les macros :

- `WIFEXITED(status)` : vrai si le processus s'est terminé normalement, dans ce cas `WIFEXITED(status)` donne le code de retour
- `WIFSIGNALED(status)` : vrai si le processus s'est terminé anormalement (par un signal). Dans ce cas, `WIFSIGNALED(status)` donne le N° de signal

Exemple sigaction et processus

```
int mon_system(const char *commande)
{
    /* commande NULL : rien à faire */
    if (commande == NULL)
        return 1;

    /* ignore les signaux SIGINT et SIGQUIT ;
     * sauvegarde des anciens traitements pour une restauration ultérieure */
    struct sigaction sa_ign, sauv_int, sauv_quit;
    sa_ign.sa_handler = SIG_IGN;
    sa_ign.sa_flags = 0;
    sigemptyset(&sa_ign.sa_mask);
    sigaction(SIGINT, &sa_ign, &sauv_int);
    sigaction(SIGQUIT, &sa_ign, &sauv_quit);

    /* création du processus fils */
    pid_t fils = fork();
    if (fils == 0) {            /* processus fils */
        /* restauration de la gestion des signaux SIGINT et SIGQUIT pour le
         * processus fils */
        sigaction(SIGINT, &sauv_int, NULL);
        sigaction(SIGQUIT, &sauv_quit, NULL);
        /* exécution de la commande */
        execl("/bin/sh", "sh", "-c", commande, (char *)NULL);
        _exit(127);             /* execl() a échoué */
    }

    int statut;
    if (fils == -1) {
        statut = -1;            /* fork() a échoué */
    } else {
        /* attente du processus fils et récupération du statut */
        while (waitpid(fils, &statut, 0) == -1) {
            if (errno != EINTR) { /* waitpid() a échoué */
                statut = -1;
                break;
            }
        }
    }

    /* restauration de la gestion des signaux SIGINT et SIGQUIT */
    sigaction(SIGINT, &sauv_int, NULL);
    sigaction(SIGQUIT, &sauv_quit, NULL);

    /* le statut du processus fils est retourné */
    return statut;
}
```

```
void verification_system(const char *commande)
{
    int a = system(commande);
    int b = mon_system(commande);
    fprintf(stderr, "%s: \"%s\" (%d, %d)\n",
            a == b ? "OK  " : "FAIL", commande ? commande : "(null)", a, b);
}

void verification_system_limit(int resource, int valeur, const char *commande)
{
    struct rlimit oldrl, newrl;
    if (getrlimit(resource, &oldrl) == -1) {
        perror("getrlimit");
        return;
    }
    newrl = oldrl;
    newrl.rlim_cur = valeur;
    if (setrlimit(resource, &newrl) == -1) {
        perror("setrlimit");
        return;
    }
    verification_system(commande);
    setrlimit(resource, &oldrl);
}

int main(int argc, char *argv[])
{
    if (argc >= 2) {
        for (int i = 1; i < argc; i++)
            verification_system(argv[i]);
    } else {
        const char *cmds[] = {
            "",                     /* empty command */
            "true",                 /* successful command */
            "false",                /* failing command */
            "ls / > /dev/null",     /* another command */
            "exec 2>/dev/null; plop", /* non-existent command */
            "kill -HUP $$",         /* killed by SIGHUP */
            "kill -INT $$",         /* killed by SIGINT */
            "kill -QUIT $$",        /* killed by SIGQUIT */
            "kill -INT $PPID",      /* send SIGINT to main process */
            "kill -QUIT $PPID",     /* send SIGQUIT to main process */
            NULL                    /* NULL command */
        };

        for (int i = 0; i == 0 || cmds[i - 1] != NULL; i++) {
            verification_system(cmds[i]);
        }
        verification_system_limit(RLIMIT_NPROC, 0, ": failed fork");
        verification_system_limit(RLIMIT_AS, 0, ": failed exec");
    }
}
```

Thread :

Exemple :

```
/*
 * Cinq threads font en même temps des mises à jour sur une donnée partagée.
 * Sans précaution, on voit que le résultat ne correspond pas au résultat
 * attendu.
 *
 * Une solution pour résoudre le problème est d'utiliser un verrou (mutex) pour
 * protéger la section critique (la partie du code qui accède à la donnée
 * partagée).
 *
 * Un mutex est de type pthread_mutex_t, s'acquiert par pthread_mutex_lock() et
 * se relâche par pthread_mutex_unlock().  Cf. les lignes 27, 35 et 37
 * ci-dessous.
 */

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

#define NTHREADS 5

struct param {
    long *p;
    long n;
    long i;
};

/* pthread_mutex_t verrou = PTHREAD_MUTEX_INITIALIZER; */

/* La fonction de calcul des threads.
 */
void *calcul(void *arg)
{
    struct param *param = arg;
    for (long i = 0; i < param->n; i++) {
        /* pthread_mutex_lock(&verrou); */
        *param->p += param->i;
        /* pthread_mutex_unlock(&verrou); */
    }
    return NULL;
}

int main(int argc, char *argv[])
{
    long v = 0;                 /* la variable partagée */
    struct param param = {      /* les paramètres des threads */
        .p = &v,
        .n = 100000,
        .i = 3,
    };
```

```c
    if (argc > 1)
        param.n = atol(argv[1]);
    if (argc > 2)
        param.i = atol(argv[2]);
    if (argc > 3 || param.n == 0 || param.i ==0) {
        fprintf(stderr, "Usage: %s [n [i]]\n", argv[0]);
        return EXIT_FAILURE;
    }

    printf("# n = %ld, i = %ld, nthreads = %d\n", param.n, param.i, NTHREADS);

    /* Lance les NTHREADS threads. */
    pthread_t threads[NTHREADS];
    for (int i = 0; i < NTHREADS; i++) {
        if (pthread_create(&threads[i], NULL, calcul, &param) != 0)
            perror("pthread_create");
    }

    /* On attend que les threads lancés ci-dessus se terminent. */
    for (int i = 0; i < NTHREADS; i++)
        pthread_join(threads[i], NULL);

    printf("Résultat ..........: %ld\n", v);
    printf("Résultat théorique : %ld\n", NTHREADS * param.n * param.i);

    return EXIT_SUCCESS;
}
```