Système

fichier - écriture ouverture

int fd = open (chemin, flag, mode)

char* ——— chemin

flag :
lecture :
écriture :
les 2 :

ne pas oublier de fermer : close(fd)

read(fd, buf, taille_buf)

void *

retourne le nombre écrit ≤ taille_buf

write (fd, buf, n)

lseek (fd, offset, whence)

décalage

SEEK_SET
SEEK_CUR
SEEK_END

Inoeud

```c
res = lstat(argv[1], &st);

//avec lstat test_lien => lien symbolique
if(res < 0){
   perror("stat");
   exit(1);
}
else{
   printf("n° inoeud :%ld \n", st.st_ino);

   printf("Type de fichier :  ");

   switch (st.st_mode & S_IFMT) {
       case S_IFBLK:  printf("périphérique de bloc\n");      break;
       case S_IFCHR:  printf("périphérique de caractère\n"); break;
       case S_IFDIR:  printf("répertoire\n");               break;
       case S_IFIFO:  printf("FIFO/tube\n");                break;
       case S_IFLNK:  printf("lien symbolique\n");          break;
       case S_IFREG:  printf("fichier ordinaire\n");        break;
       case S_IFSOCK: printf("socket\n");                   break;
       default:       printf("inconnu ?\n");                break;
   }
}
```

Random

```c
srand(time(NULL));
rand() % MAX;
```

créer des zombies

Répertoire

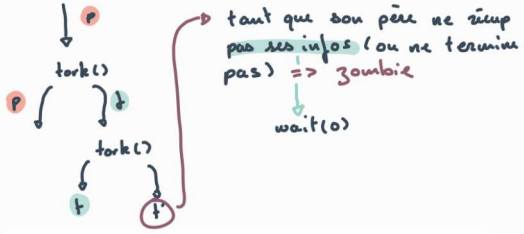
Se déplacer dans les dossiers

```c
int countF(char *path) {
    DIR *dir_ptr = NULL;
    struct dirent *direntp;
    char *npath;
    assert(path != NULL);
    if( (dir_ptr = opendir(path)) == NULL ) return 0;

    int count=0;
    while( (direntp = readdir(dir_ptr)))
    {
        if (strcmp(direntp->d_name,".")==0 ||
            strcmp(direntp->d_name,"..")==0) continue;
        switch (direntp->d_type) {
            case DT_REG: //pour compter les fichiers normaux
                ++count;
                break;
            case DT_DIR:
                npath=malloc(strlen(path)+strlen(direntp->d_name)+2);
                // sprintf(npath,"%s/%s",path, direntp->d_name);
                count += countF(npath)+1; //compte le dossier + son contenu
                // pour rester à la surface, enlever l'appel à countF
                free(npath);
                break;
        }
    }
    closedir(dir_ptr);
    return count;
}
```

Processus

- Comment créer n fils




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

main()
{
        int pid;
        int i = 1;
        int nbre  = 5;   //** Nombre de zombies à créer
        int vcrea = 2;   //** Vitesse de création des zombies, en seconde
        int vdest = 5;   //** Vitesse de destruction des zombies, en seconde
        int tpsZ  = 20;  //** Temps d'attente pour observer les zombies, en second
        int tpsP  = 5;   //** Temps d'attente pour observer le père, en seconde

        for (i; i <= nbre; i++)
        {
                pid = fork();

                if (pid == 0) // processus fils
                {
                        printf("* Zombie %d dit : Ceeeervau.....\n", i);
                        exit(1);
                }
                else // processus pere
                {
                        sleep(vcrea);
                }
        }

        printf ("-----------------------------------------------------------------\n");
        printf ("  Vous avez %d processus zombies.     \n", i-1);
        printf ("  Ils errent pendant %d sec, observable avec ps -aux. \n",tpsZ);
        printf ("-----------------------------------------------------------------\n");

        // Temps d'attente, en seconde, pour observer les zombies
        sleep(tpsZ);

        // Le père fait ensuite appel à wait() et récupère ces fils terminés
        for (i; i > 1; i--)
        {
                sleep(vdest);
                wait(0);
                printf("* Le zombie %d a disparu \n", i-1);
        }

        // Le père reste encore x secondes pour l'observer
        sleep(tpsP);
        // Fin du script, le père se termine
}
```

Tube & Signal

- un mécanisme de communication entre processus,
- manipulable presque comme un fichier ordinaire – descripteur, read, write...
- la lecture est destructrice : tout octet lu est consommé et retiré du tube
- flot continu de caractères : pas de séparation entre 2 écritures successives
- fonctionnement de type fifo, unidirectionnel : un tube a une extrémité en écriture et une en lecture
- capacité limitée (donc notion de tube plein)
- par défaut, les opérations sur les tubes sont bloquantes
- un tube est auto-synchronisant : impossible de lire un caractère avant qu'il ne soit écrit !

int pipe(int pipefd[2]);

- crée et ouvre un tube anonyme – donc alloue :
- stocke ces descripteurs dans pipefd : lecture dans pipefd[0], écriture dans pipefd[1],
- renvoie 0 en cas de succès, -1 en cas d'échec (si la table de descripteurs du processus ou la table des ouvertures de fichiers est pleine)

le tube créé n'est accessible que via ces 2 descripteurs – comme il n'a pas de nom, on ne peut pas le réouvrir avec open.
⇒ seuls les descendants du processus qui a créé un tube anonyme peuvent donc y accéder, en héritant des descripteurs.

- si le tube n'est pas vide et contient taille octets, nb_lus = min(taille, TAILLE_BUF) octets sont extraits et copiés dans buf
- si le tube est vide, le comportement dépend du nombre d'écrivains (i.e. de descripteurs en écriture sur le tube) :
- renvoie nb_lus = 0 si le nombre d'écrivains est nul
- sinon, par défaut la lecture est bloquante : le processus est mis en sommeil jusqu'à ce que quelque chose change (contenu du tube ou nombre d'écrivains)

le caractère bloquant permet la synchronisation d'un lecteur sur un écrivain... mais peut également provoquer des auto- ou interblocages

Tubes nommés

de même nature que les tubes anonymes, mais avec une existence dans le SGF :

- création et ouverture séparées
- accessibles par des processus non nécessairement apparentés
- accessibilité contrôlable
- persistants (enfin... pas leur contenu)

Création

```c
int mkfifo(const char *pathname, mode_t mode);
```

- Suppression avec unlink(), renommage avec rename(), changement des droits avec chmod()... comme les autres entrées de répertoires

Parfois le comportement bloquant par défaut des tubes n'est pas adapté ; on peut le modifier :

- à l'ouverture d'un tube nommé, avec le flag O_NONBLOCK
- (sous Linux) à la création/ouverture d'un tube anonyme par
  - int pipe2(int pipefd[2], int flags), avec le flag O_NONBLOCK
- après l'ouverture, en modifiant les flags du descripteur avec
  - int fcntl(int fd, int cmd, ... /*arg */)

comportement d'une ouverture non bloquante :
• en lecture, elle réussit immédiatement ;
• en écriture, elle réussit seulement en présence d'un lecteur ; sinon elle échoue, avec errno=ENXIO

terminaison du processus
- SIGINT, SIGTERM, SIGKILL...
terminaison + génération d'un fichier core
- SIGQUIT, SIGSEGV...
signal ignoré
- SIGCHLD, SIGWINCH
suspension du processus
- SIGSTOP, SIGTSTP
reprise du processus
- SIGCONT

Principaux signaux

- problèmes matériels : SIGBUS, SIGSEGV, SIGILL, SIGFPE 2
- événements « externes » : SIGCHLD 3 , SIGPIPE 1
- job-control :
  - SIGTERM, SIGKILL pour terminer 1
  - SIGSTOP, SIGTSTP pour suspendre 4
  - SIGCONT pour reprendre 5
  SIGSTOP et SIGKILL ne peuvent être ni ignorés ni captés
- événements liés au terminal :
  - SIGHUP : déconnexion 1
  - SIGINT 1 , SIGTSTP 4 , SIGQUIT 2 : ctrl-C, ctrl-Z, ctrl-\
  - SIGTTIN, SIGTTOU 4 : tentative de lecture/écriture par un processus à l'arrière-plan
  - SIGWINCH 3 : redimensionnement
- auto-notification : SIGABRT 2 , SIGALRM 1
- sans signification prédéfinie : SIGUSR1, SIGUSR2 1

pause() : permet de bloquer un processus jusqu'à la recep° d'un signal

```c
int main(int argc, char const *argv[]) {
    pid_t pid;
    pid = fork(); // essai de création fils

    if(pid == -1)
        fprintf(stderr, "impossible de créer le fils %d\n", errno);

    if(pid == 0) { // Fils
        printf("PID = %d\n", getpid());
        printf("Je suis le fils de %d\n", getppid());
        int n;

        // Appel bloquant
        scanf("Valeur pour le fils %d\n", &n);
        exit(1);
    } else { // Père
        int n, erreur;
        pid_t fils;

        // Attente du fils
        fils = wait(&erreur);

        fprintf(stdout, "PID = %d ", getpid());
        printf("Je suis le père de %d", pid);
        printf("...et je suis le fils du shell %d\n", getppid());

        // Appel bloquant
        scanf("Entrez 1 : %d ", &n);
        printf(" mon fils %d est mort avec %u\n", (int)fils, erreur);

    }

    return EXIT_SUCCESS;
}
```

- App' à table
  - Lo tous les fils arrivent
  - Lo fils-n arrive à n appel

piste

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

pid_t pid;

void handler(int sig){
 if(sif == SIGALRM){
   kill(pid, SIGSR1);
   signal(SIGUSR1, handler);
 }
 else{
   printf("Pong \n");
   signal(SIGALRM, handler);
   alarm(2);
 }
 pause();
}

void handlerf(int sig){
 if(sig == SIGALRM){
   kill(pid, SIGSR1);
   signal(SIGUSR1, handlerf);
 }
 else{
   printf("Ping \n");
   signal(SIGALRM, handler);
   alarm(2);
 }
 pause();
}

int main(void){
 signal(SIGUSR1, handler);
 pid = fork();
 if(pid == 0){
   signal(SIGUSR1, handlerf);
   kill(getppid, SIGUSR1);
 }
   pause();
}
```

ac² père

for ....

l'un à la suite : tous les créer a changer signal, puis mettre en pause, puis envoie du signa

Ping - Pong

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

int main(void){
  int fd[2];
  int fd2[2];
  int cpt = 5; //nombre de passes
  char buf[52];
  char buf2[52];

  if (pipe(fd) == -1 || pipe(fd2) == -1) {
    perror("pipe");
    exit(1);
  }
    int r = fork();
  while(cpt >= -1){
    switch (r) {
    case -1 :
        perror("fork");
        exit(1);
    case 0 : // fils renvoie pong
        printf("Fils : %d \n", cpt);
        close(fd[0]); //fermer en lecture
        close(fd2[1]); //fermer en écriture
        read(fd2[0], buf2, 52);
        printf("%s\n", buf2);
        if(cpt == 0){
            cpt--;
            write(fd[1], "Fils : Pong. \n Dehors, j'ai gagné" , 52);
            break;
        }
        else if(cpt == -1){
            cpt--;
            write(fd[1], "Fils : Bravo… une petite revanche?" , 52);
            close(fd2[0]);
            break;
        }
        else if(cpt > 0){
            cpt--;
            write(fd[1], "pong" , 52);
        }
        break;
    default : // père envoie ping
        printf("Père : %d \n", cpt);
        close(fd[1]); //fermer en écriture
        close(fd2[0]); //fermer en lecture
        if(cpt == 0){
            cpt--;
            write(fd2[1], "Père : Ping. \n Dehors, j'ai gagné" , 52);
            close(fd2[1]);
            break;
        }
        else if(cpt == -1){
            cpt--;
            write(fd2[1], "Père : Bravo… une petite revanche?" , 52);
            break;
        }
        else if (cpt > 0){
            cpt--;
            write(fd2[1], "ping" , 52);
            read(fd[0], buf, 52);
            printf("%s\n", buf);
        }
    }
  }
}
```

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

pid_t pid;

void handler(int sig){
 if(sif == SIGALRM){
   kill(pid, SIGSR1);
   signal(SIGUSR1, handler);
 }
 else{
   printf("Pong \n");
   signal(SIGALRM, handler);
   alarm(2);
 }
 pause();
}

void handlerf(int sig){
 if(sig == SIGALRM){
   kill(pid, SIGSR1);
   signal(SIGUSR1, handlerf);
 }
 else{
   printf("Ping \n");
   signal(SIGALRM, handler);
   alarm(2);
 }
 pause();
}

int main(void){
 signal(SIGUSR1, handler);
 pid = fork();
 if(pid == 0){
   signal(SIGUSR1, handlerf);
   kill(getppid, SIGUSR1);
 }
   pause();
}
```

pour le processus père

quand il reçoit ça fait

on modifie le processus fils