

## Système

fiducie - écriture ouverture

int fd = open(chemin, flag, mode)

char\*  $\xrightarrow{\text{lecture}}$

écriture :  
les 2 :

ne pas oublier de fermer : close(fd)

read(fd, buf, taille-but)

$\uparrow$   
void\*

retourne le nombre écrit  $\leq$  taille-but

write (fd, buf, n)

lseek (fd, offset, whence)

décalage  $\uparrow$

SEEK.SET  
SEEK.CUR  
SEEK.END

## Inoent

```
res = lstat(argv[1], &st);  
//avec lstat test_tlen => lien symbolique  
if(res < 0){  
    perror("stat");  
    exit(1);  
}  
else{  
    printf("n° inoent %ld\n", st.ino);  
    printf("Type de fichier : %s");  
    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("pipe/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

srand (time (null));  
rand () % MAX;

## Répétition

Se déplace dans les dossiers

```
int count(char *path) {  
    DIR *dptr = NULL;  
    struct dirent *d;  
    char *npath;  
    assert(path != NULL);  
    if( (dptr_ptr = opendir(path)) == NULL ) return 0;  
    int count0;  
    while( (d = readdir(dptr_ptr)) )  
    {  
        if (strcmp(d->d_name, ".") == 0 ||  
            strcmp(d->d_name, "..") == 0) continue;  
        switch (d->d_type) {  
            case DT_REG: //pour compter les fichiers normaux  
                count++;  
                break;  
            case DT_DIR:  
                npath = malloc(strlen(path) + strlen(d->d_name) + 2);  
                // sprintf(npath, "%s/%s", path, d->d_name);  
                count += count(npath + 1); //compte le dossier + son contenu  
                // pour rester à la surface, enlever l'appel à countf  
                free(npath);  
                break;  
            default:  
                break;  
        }  
    }  
    closedir(dptr_ptr);  
    return count;  
}
```

## Processus

Comment créer n fils  
tant que son père ne réup  
pas ses infos (ou ne termine  
pas) => zombie  
wait(0)



## Ping-Pong

```
#include <stdio.h>  
#include <signal.h>  
pid_t pid;  
void handler(int sig){  
    kill(pid, SIGUSR1);  
    signal(SIGUSR1, handler);  
}  
else{  
    printf("ping\n");  
    signal(SIGUSR1, handler);  
    alarm(2);  
    pause();  
}  
void handlerf(int sig){  
    if(sig == SIGALRM){  
        kill(pid, SIGUSR1);  
        signal(SIGUSR1, handlerf);  
    }  
    else{  
        printf("ping\n");  
        signal(SIGALRM, handlerf);  
        alarm(2);  
        pause();  
    }  
}  
int main(void){  
    signal(SIGUSR1, handler);  
    if(pid == 0){  
        // quand il reçoit ça ] on modifie le processus  
        signal(SIGUSR1, handlerf);  
        kill(getppid, SIGUSR1);  
        pause();  
    }  
}
```

## Tube a 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 pipelin(pipedf(2));

- crée et ouvre un tube anonyme - donc alloue :
  - stocke ces descripteurs dans pipedf : lecture dans pipedf(0), écriture dans pipedf(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 l'ha pas de nom, on ne peut pas le récupérer 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\_lue = min(taille, TAILLE\_BUFFER) octets sont extraits et copiés dans buf
- si le tube est vide, le comportement dépend du nombre d'écritures (i.e. de descripteurs en écriture sur le tube) :
- renvoie nb\_lue = 0 si le nombre d'écritures 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'écritures)

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)

```
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 pipzint(pipedf(2), int flags), avec le flag O\_NONBLOCK
  - après 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=ENNO

• App' à table

Lo tous les fils arrivent

Lo fils-n arrive à n appel

piste

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

pid_t pid;

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

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

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



l'un à la suite : tous les clés n change  
signal, puis mettez en pause, puis envoi du signal

terminalisation du processus  
- SIGINT, SIGTERM, SIGKILL...  
terminalison + génération d'un fichier core  
- SIGQUIT, SIGSEGV...  
signal ignoré  
- SIGCHLD, SIGWINCH  
suspension du processus  
- SIGSTOP, SIGTSTP  
copie 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 5SIGSTOP 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'arrêter-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

```
int main(int nbargs, char ** args) {
    pid_t pid;
    pid = fork(); //essai de creation fils
    if( pid == -1 )
        fprintf(stderr, "impossible de creer le fils %d\n", errno);
    if ( pid == 0 ) { //fils
        printf("pid=%d : ", getpid());
        printf("je suis le fils de %d\n", getppid());
        int n;
        //appel bloquant
        scanf("%d", &n);
        exit(1);
    }
    else { //pere
        int n, erreur;
        pid_t fils;
        //attente du fils
        fils = wait(&erreur);
        fprintf(stdout, "pid=%d : ", getpid());
        printf("je suis le pere de %d", pid);
        printf("...et je suis le fils du shell %d\n", getppid());
        //appel bloquant
        scanf("entree 1 : %d", &n);
        printf("mon fils %d est mort avec %d\n", (int) fils, erreur);
    }
    return EXIT_SUCCESS;
}
```