Memento primitives système Unix

Si non précisé, retour de fonctions : 0 = Succès, -1 = Erreur

Types utilisés (liste non exhaustive) pour assurer la portabilité et la lisibilité :

- `clock_t` (entier long non signé) : nombre de tops horloge
- `dev_t` (entier long) : numéro (mineur et majeur) de périphérique
- `gid_t` (entier long) : identificateur de groupe
- `ino_t` (entier long non signé) : numéro d'inode
- `key_t` (entier long) : clé utilisée pour les IPC System V
- `mode_t` (entier court non signé) : permissions associées à un fichier
- `off_t` (entier long) : déplacement dans un fichier
- `pid_t` (entier long) : identificateur de processus
- `sigset_t` (tableau) : bitmap utilisée pour les signaux POSIX
- `size_t` (entier non signé) : nombre d'octets
- `socklen_t` (entier long) : taille de socket
- `ssize_t` (entier) : nombre d'octets
- `time_t` (entier long) : nombre de secondes depuis le 01/01/1970
- `uid_t` (entier long) : identificateur de processus

1 Manipulation i-nodes

Informations générales :

```
int stat(const char *nom_f, struct stat *ptr)
int fstat(int desc, struct stat *ptr)
int lstat(const char *nom_f, struct stat *ptr)
```
`nom_f` est un lien symbolique

Test accès et droits d'accès

```
int access(char *nom_f, int type)
```
type = R_OK, W_OK, X_OK, `F_OK` (test existence du fichier)

```
int chmod(const char *path, mode_t mode)
```
change les droits, ex : chmod("monfichier ", S_IRWXU|S_IRGRP|S_IXGRP|S_IROTH) ou chmod(" monfichier ", 0754)

Liens

```
int link(char *nom_f1, char *nom_f2)
```
f1 pas repértoire, f2 non existant et même disque logique que f1

```
int symlink(char *nom_f1, char *nom_f2)
```
idem mais lien symbolique (=> nouvel i-node, compteur de liens pas incrémenté, contient simplement chemin d'accès).

```
int unlink(char *nom_f)
```
destruction lien et fichier si dernier lien.

2 Primitives de manipulation de fichiers

```
int open(char *nom_f, int mode, .../*mode_t droits*/)
```

mode : O_RDONLY, O_WRONLY, O_RDWR, O_CREAT, O_TRUNC (si fich existe, destruction), O_APPEND (écriture autom. en fin), droits = S_IRUSR, ..., S_IWGRP, ..., S_IXOTH. retourne un descripteur

```
int creat(char *nom_f, mode_t droits)
```
= open(nom_f, O_WRONLY|O_CREAT|O_TRUNC, droits)

```
int close(int desc)
int read(int desc, void *ptr, int nb_octets)
int write(int desc, void *ptr, int nb_octets)
```
renvoie nb octets réellement lus

renvoie nb octets réellement écrits

`int` **`dup`**`(int desc)` cherche plus petit descripteur libre et le fait pointer sur même fichier que desc.
`int` **`dup2`**`(int desc1, int desc2)` = close(desc2); dup(desc1)
`int` **`lseek`**`(int desc, int offset, int origine)`

origine = SEEK_SET, position courante = SEEK_CUR, fin de fichier = SEEK_END

`int` **`fcntl`**`(int desc, int commande, .../*param*/)` manipulation directe des info du descripteur : ouverture, fermeture, mode accès fichier, verrouillage (ex rajout de O_APPEND : fcntl(desc, F_SETFL, fcntl(d, F_GETFL)|O_APPEND)), F_GETxx retourne info, F_SETxx modif.

Fonctions parcours de répertoire : opendir, readdir, closedir, rewinddir , seekdir , telldir , scandir

bibliothèque entrée standard C :
int *fileno*(FILE *) (renvoie desc associé), *fopen*, *fdopen* (desc en param et non nom_f), *freopen* (réouvre, utile pour redirection), *tmpfile* (création fich. temp.), *tmpnam* (associe nom à fich. temp.), *fclose*, *putc*, *fputc* (vraie fonction contrairement à putc qui est une macro et ne peut pas être passée en param.), *fputs*, *fwrite*, *fprintf*, *getc*, *fgetc*, *ungetc* (replace car. dans tampon lect/ecr), *fgets*, *gets* (lit sur entrée standard), *fread*, *fscanf*, *fseek*, *feof*, *setvbuf* et *setbuf* (associe un nouveau tampon lect/ecr au fich.), *fflush* (vide le tampon)

3 Processus

`pid_t` **`getpid, getppid`**`()` id du proc, de son père
`uid_t` **`getuid, geteuid`**`()` propriétaire réel/effectif
+ **setuid**, **getgid**, **getegid**, **setgid**.
`int` **`chdir`**`(const char *dirname)` change le répertoire courant (cwd : current working directory)
`void` **`exit`**`(int state)` sort du processus, state utilisé par wait (= $? dans un shell).
`char *`**`getcwd`**`(char *buf, int taille)` place dans buf répertoire de travail.
`clock_t` **`times`**`(struct tms *buf)` => buf.tms_utime : nb clics horloge en mode utisateur, buf.tms_stime : idem en mode noyau, nb clics par seconde = CLK_TCK)
`mode_t` **`umask`**`(mode_t masque)` droits de création des fichiers crées (voir open).
`pid_t` **`fork`**/**`vfork`**`()` création proc. fils, renvoie 0 chez le fils et le pid du fils chez le père, fork : données père et fils séparées (recopie), vfork pas de recopie de données (optimisation si suivi de exec).
`pid_t` **`wait`**`(int *ptr_status)` attend la fin d'un fils quelconque dont le pid est retourné, si ptr_status ≠ NULL, alors contient état de la terminaison du proc.
`pid_t` **`waitpid`**`(pid_t pid, int *ptr_status, int options)` attend un proc dont le pid est spécifié.
`int` **`execl, execlp`**`(char *ref, char *arg, ..., NULL)` recouvrement (héritage des descripteurs et fichiers ouverts) ex : execl("/bin/ls", "ls", "-l", "/", NULL). execl -> chemin complet (pas de recherche du fichier), execlp -> utilise PATH (recherche du fichier).
`int` **`execle`**`(char *ref, char *arg, ..., NULL, char **arge)` idem mais possibilité de préciser les variables d'environnement.
`int` **`execv, execvp`**`(char *ref, char *argv[])` idem exec/execlp mais avec les param. dans un tableau (sur le même schéma il existe **execve**).
`getrlimit`/**`setrlimit`** : consulter/modifier les ressources du processus (taille max de pile, nb fichiers ouverts, etc)

4 Les tubes

Un tube (fifo) se maniple comme un fichier (fstat, fcntl).
`int` **`pipe`**`(int p[2])` ouverture deux desc. sur un fichier interne, temporaire, ecriture dans p[1] et lecture dans p[0].
`int` **`mkfifo`**`(char *ref, mode_t mode)` création du tube nommé = fichier dans repértoire, se supprime avec `unlink`, `rm`. Sa particularité reste qu'il s'agit d'une fifo.

5 Les signaux

Construction d'un ens. de signaux

`int` **`sigemptyset, sigfillset`**`(sigset_t *p_ens)` ensemble vide et ensemble de tout les signaux

`int` **`sigaddset, sigdelset, sigismember`**`(sigset_t *p_ens, int sig)`

Fonctions sur signaux

`int` **`kill`**`(pid_t pid, int sig)` envoie d'un signal, si sig = 0 => test de l'existence du processus.

`int` **`raise`**`(int sig)` standard C, envoie d'un signal au processus courant.

`int` **`sigaction`**`(int sig, struct sigaction *p_action, struct sigaction *p_old_action)` définit quelle fonction appeler (définie dans la structure) pour le signal sig.

`int` **`sigprocmask`**`(int op /* SIG_SETMASK */, sigset_t *set, sigset_t *old_set)` masquage (non prise en compte) des signaux de set (débloquage de tous les signaux si set est vide), si old_set ≠ NULL alors recupération des anciens masques.

`int` **`sigpending`**`(sigset_t *set)` retourne dans set l'ensemble des signaux reçu par le processus et qui n'ont pas été pris en compte (pas encore eu le temps ou masquage).

`int` **`pause`**`()` attend un signal sans savoir lequel est reçu.

`int` **`sigsuspend`**`(sigset_t *set)` attend un signal en ignorant ceux de set.

`int` **`alarm`**`(int secondes)` envoie le signal SIG_ALARM après un laps de temps.

6 Terminal

Rappel : ens des proc partionné en sessions. un terminal est associé à au plus une session et reciproquement. une session est partionnée en n groupes de proc. (1 groupe en premier plan et n-1 groupe en arrière plan). Chaque session a un proc leader et chaque groupe a un proc leader. Dans les fonctions, si pid = 0 alors désigne proc. appelant.

int **isatty**(int desc) 1 si desc associé à un terminal et 0 sinon.

char ***ttyname**(int desc) chemin d'accès du terminal.

pid_t **getpgrp**() rend pid du processus leader du groupe auquel le processus appartient

pid_t **getpgrp2**(pid_t pid) idem mais demande pour le proc. pid.

pid_t **setpgid**(pid_t pid1, pid_t pid2) rattache proc. pid1 à groupe du proc. pid2, renvoie pid

pid_t **setsid**() crée une nouvelle session sans terminal dont le proc appelant est le leader et le seul membre.

pid_t **tcgetsid**(int desc) renvoie pid leader session rattachée au terminal desc.

pid_t **getsid**(pid_t pid) renvoie leader session du proc pid

pid_t **tcgetpgrp**(int desc) renvoie leader groupe en premier plan de la session rattachée au terminal desc.

pid_t **tcsetpgrp**(int desc, pid_t id_grp) force le groupe du proc id_grp à devenir le groupe de premier plan de la session dont desc est le terminal.

7 Les IPC (Inter Process Communication)

Files de messages :

`int msgget``(key_t key, int msgflg)` : création/récupération si déjà créée

int **msgctl** (int msqid, int cmd, struct msqid_ds *buf) : selon cmd, récup. informations ou destruction

int **msgsnd** (int msqid, struct msgbuf *msgp, size_t msgsz, int msgflg) écriture d'un message

`ssize_t` **`msgrcv`** `(int msqid, struct msgbuf *msgp, size_t msgsz, long msgtyp, int msgflg)` : lecture

Mémoire partagée

`int` **`shmget`**`(key_t clé, int size, int shmflg)` : création/récupération si déjà créée

`int` **`shmctl`**`(int shmid, int cmd, struct shmid_ds *buf)` : contrôle divers

`char *`**`shmat`**`( int shmid, char *shmaddr, int shmflg )` : attachement à une adresse

`int` **`shmdt`**`( char *shmaddr)` : détachement

Sémaphores

`int` **`semget`** `(key_t key, int nsems, int semflg)` : création/récupération si déjà créés

`int` **`semctl`** `(int semid, int semno, int cmd, union semun arg)` : contrôle (init., récup. param, destruction, etc.)

`int` **`semop`** `( int semid, struct sembuf *sops, unsigned nsops )` : modification valeur

8 Les sockets

int **socket**(int domaine, int type, int protocole) domaine = AF_UNIX, AF_INET, type = SOCK_DGRAM, SOCK_STREAM, protocole = IPPROTO_UDP, IPPROTO_TCP, 0 => prot. par défaut. Renvoie un desc. et a principalement comme effet de définir le protocole (algorithme qui sera utilisé pour envoyer les messages). Se ferme avec close.

int **bind**(int desc, struct sockaddr_xx *ptr, int lgr) on donne une adresse à la socket, si internet (AF_INET) alors ptr de type sockaddr_in et on donne adr IP + num port, si interne au système UNIX alors ptr de type sockaddr_un et on donne juste le nom d'un fichier qui se alors créé.

struct hostent ***gethostbyname**(char *nom_machine) recuperation de l'adresse IP d'une machine, celle-ci doit généralemennt être ensuite placée dans une structure sockaddr_in..

int **sendto/recvfrom**(int desc /*socket locale*/, void *message, int lgr, int option /* 0 */, struct sockaddr_xx *ptr /* adresse destinataire/émetteur */, int lgr_adr) envoie/recoit message.

+ sendmsg/recvmsg pour l'envoi de fragments de message (utile ?).

int **connect**(int desc, struct sockaddr_xx *ptr, int lgr) permet d'associer l'adresse du destinataire à une socket et d'utiliser alors read/write ou send/recv plutôt que sendto/recvfrom.

Construction d'un ensemble de descripteur

FD_ZERO(fd_set *ptr_set) ens. vide FD_CLR, FD_SET, FD_ISSET(int desc, fd_set *ptr_set) suppression/ajout/test d'existence d'un desc à un ens.

int **select**(int nb_desc, fd_set *ptr_lect, fd_set *ptr_ecr, fd_set *ptr_execption, struct timeval *ptr_temp) nb_desc = num max + 1 des desc des ens. en param, les autres champs peuvent avoir la valeur NULL. select est bloqué pendant ptr_temp (infini si NULL) ou jusqu'à ce que l'un des descripteurs ait reçu un évènement. Ex : on attend en lecture sur les descripteurs 3 et 5 => select(6, A, NULL, NULL, NULL) avec A = {3, 5} puis arrivé d'un msg pour 3 => A ne contient plus que 3 au retour du select (à tester avec FD_ISSET)

Mode connecté :

int **listen**(int desc, int nb_connexion) déclare au système local que la socket est prêt à recevoir jusqu'à nb_connexion (retour 0 ou -1 suivant succès ou échec).

int **accept**(int desc, struct sockaddr *ptr, socklent_t ***longueur**) fonction bloquante, répond à un connect côté client. desc est la socket à l'écoute et ptr contient l'adresse de la socket qui a réalisé le connect (retour du descripteur de la socket).

9 Thread

Threads

`int` **`pthread_create`**`(pthread_t * thread, pthread_attr_t * attr, void * (*start_routine)(void *), void * arg)` : crée un thread

`void` **`pthread_exit`**`(void *`*`retval`*`)` : termine le thread courant

`int` **`pthread_join`**`(pthread_t th, void **thread_return)` : attend la mort d'un thread

manipulation des attributs passés à pthread_create : **`pthread_attr_init`**, **`pthread_attr_setdetachstate`**, **`pthread_attr_setschedpolicy`**, **`etc..`**

Mutex

```
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutexattr_t
*mutexattr)
int pthread_mutex_lock(pthread_mutex_t *mutex))
int pthread_mutex_trylock(pthread_mutex_t *mutex)
int pthread_mutex_unlock(pthread_mutex_t *mutex)
int pthread_mutex_destroy(pthread_mutex_t *mutex)
```