Programmation système en C : entrées sorties

Contexte

- À partir de maintenant, on fait du C
- But de ce "chapitre" :
  - se familiariser avec les appels système
  - se familiariser avec les descripteurs de fichiers

Les appels système

- Un appel système : le processus appelle directement une fonction du noyau.
- En interne : cela se fait par un mécanisme spécial (interruption)
- En pratique, ce sont des fonctions que l'on appelle (comme à l'accoutumé en C)
- Attention : un appel système est plutôt lent !
- Pour voir les appels système d'un processus : `strace` ou `ltrace -S`

Codes retour et erreurs

- Les appels système renvoient un code retour
- Il faut toujours vérifier si cela a marché !
- les codes erreurs sont généralement retournés dans la variable externe `errno`
- `perror` permet d'afficher de manière compréhensive une erreur système.
- regardez la page du manuel des la fonction que vous utilisez pour comprendre le code retour !

Quelques rappels en C

```c
#include <stdio.h> /* pour printf() */
int main(int argc, char **argv)
{
  /* argc     : nombre d'argument dans la ligne
     de commande
                (y compris l'executable)
     argv[i] : le ieme argument  */

  int i;
  printf("le nombre d'argument est %d\n",argc);
      /* affiche sur la sortie standard */
  for(i=0;i<argc;i++)
    printf(" l'argument %d est %s\n",i,argv[i])
      ;

  return 0; /* code de retour */
}
```

Quelques rappels en C

En fait, `main` peut accepter un 3ème argument, qui sera l'ensemble des variables d'environnement

```
int main(int argc, char **argv, char **env)
   int i;
   for(i=0;env[i]!=NULL;i++)
      printf("%d) %s\n",i,env[i]);
   return 0;
}
```

`printf` est une fonction de la bibliothèque standard du C (libc/glibc), une couche entre l'OS et nos programmes.

- Il y a deux niveaux de gestion des E/S et fichiers : bibliothèque standard ou par appel système.
- `fprintf` utilise un appel système (`write`) pour afficher la chaîne de caractères

Exemple : écriture dans un fichier via la bibliothèque standard vs fonctions système

- bibliothèque standard : fopen, fwrite (ou fprintf), fclose
- système : open, write, close

Ouvrir un fichier

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags)
```

- ouvre le fichier au chemin `pathname`
- renvoie un entier, le descripteur de fichier
- renvoie -1 si l'ouverture échoue (fichier non trouvé, pb de droits...)
- les autres fonction d'accès prennent en paramètre ce descripteur de fichier.
- Pour fermer un descripteur : `close(descripteur)`.
- Toujours fermer quand on s'en sert plus !

Ouvrir un fichier

- `flags` : conjonction de :
  - O_RDONLY, O_WRONLY, ou O_RDWR (lecture, écriture ou les 2)
  - O_CREAT : crée le fichier (s'il n'existe pas)
  - O_APPEND : rajoute à la fin du fichier (positionne à la fin du fichier)
  - O_TRUNC : tronque le fichier à la taille 0.

- `open` peut prendre un 3eme argument : le mode (droits "rwx" pour "ugo", en octal) si un fichier est crée

Exemple :

```
int fd=open("file.txt",O_WROLNY | O_CREAT | O_TRUNC, 0644);
```

Entrées/sorties standards

Un processus possède à l'origine 3 descripteurs de fichiers ouverts :

- 0 : ouvert en lecture : l'entrée standard
- 1 : ouvert en écriture : la sortie standard
- 2 : ouvert en écriture : la sortie erreur

Lire dans un fichier

```
#include <unistd.h>
ssize_t read(int fd, void *buf, size_t count);
```

- `fd` : descripteur de fichier
- `buf` : pointeur vers la zone mémoire où seront copiées les données
- `count` : nombre maximum d'octets à lire
- retour : nombre d'octets lus, -1 si erreur

Similairement : `write` pour écrire

Se déplacer dans un fichier

```
#include <sys/types.h>
#include <unistd.h>
off_t lseek(int fd, off_t offset, int whence);
```

- `offset` : de combien d'octets on se déplace (positif ou négatif) depuis :
  - (si `whence=SEEK_SET`) le début du fichier
  - (si `whence=SEEK_CUR`) la position courante
  - (si `whence=SEEK_END`) la fin du fichier
- retour : position dans le fichier

(Pour connaître la taille d'un fichier `lseek(fd,0,SEEK_END)`)

Dupliquer les descripteurs

```
#include <unistd.h>
int dup(int oldfd);
int dup2(int oldfd, int newfd);
```

- `dup` duplique le descripteur `oldfd`, et renvoie un nouveau descripteur
- `dup2` copie le descripteur `oldfd` dans `newfd`

Exemple :

```
int fd=open("sortie.txt",O_CREAT|O_WRONLY,0644);
dup2(fd,1);
```

Autres fonctions pour gérer les fichiers

- pour tronquer un fichier à la position courante : `truncate`
- pour créer un fichier : `open` ou `create`
- pour supprimer un fichier : `unlink`
- pour créer un lien dur : `link`
- pour renommer un fichier : `rename`

Scanner les répertories

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

DIR *opendir(const char *name);
struct dirent *readdir(DIR *dirp);
int closedir(DIR *dirp);

struct dirent {
  ino_t          d_ino; /* inode number */
  off_t          d_off; /* see man */
  unsigned short d_reclen; /* length */
  unsigned char  d_type; /* type of file */
  char           d_name[256]; /* filename */
};
```

Informations sur un fichier

```
int stat(const char *pathname, struct stat *buf);

struct stat {
    dev_t st_dev; /* device containing file */
    ino_t st_ino; /* inode number */
    mode_t st_mode; /* protection */
    nlink_t st_nlink; /* number of hard links */
    uid_t st_uid;  /* user ID of owner */
    gid_t st_gid;  /* group ID of owner */
    dev_t st_rdev; /* device ID (if special file) */
    off_t st_size; /* total size, in bytes */
    blksize_t st_blksize;
      /* blocksize for filesystem I/O */
    blkcnt_t st_blocks;
      /* number of 512B blocks allocated */
    struct timespec st_atim; /* last access */
    struct timespec st_mtim; /* last modification */
    struct timespec st_ctim; /* last status change */
};
```

Autres

Autres appels système qui peuvent servir (et ne sont pas le sujet de prochains cours)

- `time` : renvoie le temps Unix, i.e. le nombre de secondes depuis le 1er Janvier 1970.
- `exit` : termine le programme
- `nanosleep` : endort le processus pour un temps déterminé (void aussi `sleep` et `usleep`)

Pour trouver l'appel système si on a la commande shell : regarder la fin du man. (Notamment : gestion des droits et fichiers spéciaux...)

La mémoire

Les différentes mémoires vives

Une machine possède différent type de mémoire vive :

- La “mémoire principale” (RAM). Taille de l'ordre de Giga-octet (ordinateur/smartphone actuel) au Tera-octet (grosses machines de calcul).
- Les registres. Il s'agit de mémoires directement implantées dans l'unité de calcul du processeur.
- Les mémoires caches.
  - Pour accélérer les accès mémoires.
  - Gérées par le CPU.
  - Transparentes pour l'utilisateur / l'OS.
  - On en reparlera plus.
- Le “swap”.

Mémoire principale

- La mémoire principale est un tableau d'octets (=8 bits).
- Une adresse mémoire est un index (un "numéro") de case mémoire.
- Un pointeur : une variable qui contient une adresse mémoire.

Sur une machine 64 bits :

- Une adresse mémoire est un entier de 64 bits
- Théoriquement, 2^{64} octets accessibles = 17179869184 Go...

Adressage sans abstraction

Dans les "vieux" ordinateurs (-286, DOS) (et dans certains modes des ordinateurs actuels) :

- Si processus accède à la donnée à l'adresse i, il accède à la donnée à l'adresse i dans la RAM :
  Le processus "voit" directement la mémoire physique.

Deux processus ne peuvent pas utiliser la même zone mémoire, sans interférer.

Problèmes :

- Un processus voit la mémoire des autres processus
- les processus peuvent empiéter les uns sur les autres.
- La mémoire d'un processus doit correspondre à une zone mémoire physique (ex : utilisation de "swap" impossible)

Virtualisation de la mémoire

Mémoire virtuelle : il n'y a pas une correspondance directe entre l'espace d'adressage d'un processus et la mémoire physique.

- La mémoire vue par un processus est formée d'un ensemble de pages mémoires.
- La RAM est découpée en zones de même taille.
- Une translation (au niveau du processeur) a lieu pour convertir les adresses virtuelles en adresse physique, via la table des pages

Virtualisation de la mémoire

Avantages :

- Le processus peut organiser la mémoire comme il le veut (chaque processus a sa table)
- Des zones mémoires peuvent être partagées entre différents processus
- Déplacement possible de zones mémoires (swap...)
  - une interruption a lieu si le processus veut accéder à une zone qui ne correspond à rien dans la table des pages.

Le mode noyau et mode utilisateur

Sous Unix, il y a deux modes de fonctionnement :

- Le mode "utilisateur" : mémoire virtualisée, accès matériel impossible (autrement que via les syscalls).
  Tous vos processus seront dans ce mode.

- Le mode "noyau"
  - Le noyau voit (et peut gérer) toute la mémoire physique. Il peut modifier les tables des pages.
  - + de privilèges (accès au matériel...)

Appel système

- Un appel système : passage du mode utilisateur au mode noyau
- Via une sorte d'interruption : le processus fait basculer le processeur du mode utilisateur en mode système.
- Chaque syscall a un numéro.
- On ne peut donc pas appeler n'importe quelle fonction du noyau, seulement celles qui ont été prévues...

Différentes zones mémoire d'un processus :

- les instructions :
  - le code du programme (en langage machine)
  - les librairies qu'il utilise (libc...)
- les données :
  - segment de donnée statique
  - pile (stack)
  - tas (heap)
- non allouées : si on essaye d'y lire ou d'y écrire, il y aura une erreur de segmentation (ou segfault)
- les zones ont également des droits (lecture seule, exécution autorisée...)
- certaines zones peuvent être partagées entre différents processus (c'est un moyen de communiquer inter-processus).

Pile (Call stack)

- Sont stockés dans la pile : les variables locales aux fonctions, les paramètres des fonctions, les adresses de retour.
- Attention au dépassement !
  (On peut augmenter la taille de la pile avec `setrlimit`)

Tas (Heap)

Pour les allocation dynamiques.
En C : géré par la libc via `malloc/free`

Désavantages des malloc/free :

- (des)allocation un peu lent
- une structure allouée prend un peu plus de place en mémoire
- fragmentation
- Il ne faut pas oublier à libérer la mémoire qui ne sert plus (`free`) sinon on aura des fuites mémoires !

On peut changer la taille du tas avec `brk()` ou `sbrk()`.

Gérer différemment la mémoire dynamique

Il existe des mécanismes de ramasse miettes (garbage collector), pour éviter d'avoir à désallouer la mémoire.

On peut faire ses propres allocateur de mémoire

Exemple : "memory pool", si on alloue beaucoup d'objets de la même taille t

- on alloue un grand tableau de n cases de taille t
- une "allocation" renvoie l'adresse d'une nouvelle case
- les zones libres sont gérées par une liste chainée.

Demander des nouvelles zones mémoires

`mmap` permet de mapper de nouvelles zones mémoires

- On peut mapper soit un fichier (via un descripteur), soit une zone vierge
- Deux modes possible : "shared" ou "private"
  - private : on a notre propre copie en mémoire
  - shared : la copie est partagée
- On doit spécifier les droits (read, write, exec)
- On peut spécifier l'adresse.

On peut libérer une zone avec `munmap`.

Format et chargement des binaires

Le format des exécutables sous la plupart des Unix est ELF (Executable and Linkable Format)

- Un fichier ELF est composé de plusieurs sections, qui vont correspondre à des zones mémoires ("text" pour les instructions, "data"...)
- Pour voir les différentes sections : `objdump`
- Ces sections seront "chargées" en mémoire via mmap.

- Les librairies (glibc...) seront chargées à l'exécution, par la librairie "ld".
- "ld" cherche les librairies dans les répertoires listés dans LD_LIBRARY_PATH, /lib et /usr/lib
- Il est possible de forcer "ld" à choisir une autre librairie (LD_PRELOAD)

Suite :

- Signaux
- IPC : Tubes
- Threads