Introduction

Présentation du cours ASR2

- On se concentre sur le S et R de ASR
- Prérequis : C et dans une moindre mesure ASR1.
- Organisation : TP/TD le jeudi à 8h00 en salle Europe (001), cours le mercredi à 13h30 (amphi B).
- Projet(s) avec rendus intermédiaires (50% de la note)
- Examen final (50% de la note) : en salle machine (rendu papier + rendu binaire)
- Questions, suggestions, bugs : `michael.rao@ens-lyon.fr`

Références

Livres :

- Jean-Marie Rifflet et Jean-Baptiste Yunès, « UNIX : programmation et communication », Dunod, 2003.
- Andrew S. Tanenbaum et Herbert Bos, « Modern Operating Systems, 4th Ed », Pearson 2015.

Les pages du manuel (RTFM !)

- `man <terme>` dans un terminal
- Attention : il y a plusieurs sections !

Ce qu'il y aura dans le cours

Le cours est divisé en 3 grandes parties (à peu près équi-réparties)

- Programmation système
  - Descripteurs de fichiers, gestion de la mémoire, gestions des processus, signaux et communication inter-processus.
- Programmation multithreads
  - programmation en threads POSIX, un peu de théorie sur l'interblocage et l'ordonnancement
- Réseau
  - les couches TCP/IP, mécanismes de routage

À côté : un peu de shell, de deguggage, de bonnes pratiques et de sécurité.

Buts et non-buts

- Comprendre ce qui se passe entre la machine (le matériel, la partie A de ASR) et un processus (que vous allez programmer dans la vraie vie). Dans cette superposition de couches, le système d'exploitation (OS = Operating System) joue un rôle important.

Couches : Matériel / OS / (librairies) / Processus

- On cherche aussi à comprendre ce qui se passe entre les processus (communication, synchronisation, réseau...)
- Connaitre par coeur toutes les fonctions système n'est pas un but. Savoir se servir de la documentation pour trouver ce que l'on cherche en est un.
- On ne cherche pas, à notre niveau, à savoir concevoir et programmer un OS, juste à savoir opérer avec lui.
- On programmera en C, sous GNU/Linux. On essayera de suivre au maximum la norme POSIX.

Un OS : pourquoi ?

Que fait un système d'exploitation ?

- interface avec le matériel
- gestion de la mémoire
- gestion des périphériques
- gestion des interruptions
- gestion des système de fichiers
- ...

Ces taches sont très dépendantes du matériel, et souvent fastidieuses. L'OS est là pour nous simplifier la vie.

Que fait un système d'exploitation « moderne » ?

- multi-tâches : plusieurs processus peuvent tourner en même temps
- multi-utilisateurs : il peut y avoir plusieurs utilisateurs différents qui l'utilisent.
- gestion utilisateurs, groupes d'utilisateurs, et droits.
- gestion du réseau

- essayer d'être "compatible" avec les autres OS : respect de normes.

Systèmes "Unix"

Introduction d'Unix (1969-, K. Thompson & D. Ritchie...)

- Rompt avec les OS propriétaires, et les programmes monolithiques.
- Introduit conjointement avec le C (1969-, D. Ritchie & B. Kernighan)
- Rapidement, plusieurs branches de développement : Unix, BSD, et système propriétaires (Xenis, AIX, System V...)
- Assez rapidement, une volonté de normalisation : norme POSIX (1988-)
- Philosophie : être modulaire . Préférer des programmes simples qu'on peut composer via les entrées/sorties standards.

GNU / Linux

- GNU : "GNU is Not Unix" : projet d'OS libre (1983- R. Stallman)
- Logiciel libre : code source disponible, que l'on peut modifier et redistribuer (licence GPL "GNU Public Licence", ou similaire)
- Linux : Un noyau (cœur de l'OS) Unix libre.
- Autres Unix libres : Minix, *BSD, Hurd...

Écosystème...

Un "écosystème" vivant dans un ordinateur :

- ensemble de processus en exécution
- interagissant entre eux
- utilisant les mémoires
- éventuellement utilisant des périphériques

Les processus n'interagissent pas directement avec le matériel, ils passent par l'OS pour y avoir accès.

Mémoires

Deux types importantes de mémoire :

- la mémoire vive, "volatile"
  - non pérenne
  - accès très rapide.
- la mémoire "non volatile"
  - pérenne
  - disque dur mécanique, SSD, clef USB...
  - accès (beaucoup) plus lent

La mémoire non volatile est organisée sous forme arborescente, avec des fichiers et des répertoires : l'arborescence des fichiers.

Plan approximatif des cours :

- Arborescence de fichiers / Shell
- Programmation système : bases, entrées sorties
- Mémoire
- Processus
- Communication inter-processus.
- Threads
- Réseau
- ...

Arborescence de fichiers et Shell

Shell

Shell : interface textuelle entre l'humain et l'OS.
Votre premier objectif : maîtriser le shell

Fonctionnement d'un shell

- prompt : attente d'une commande

`commande [argument1] [argument2] ...`

- on entre une commande, éventuellement avec arguments, et on appuie sur "entrée".
- le shell exécute la commande, puis rend la main quand la commande est terminée.
- pour quitter "proprement" un shell : `exit` (ou ctrl+d sur certains shells).

Shell / terminal

Un shell se lance au travers d'un terminal.

Il existe plusieurs shells différents. Un des plus courant sous GNU/Linux est `bash`.

Permet de composer facilement des processus via les redirections d'entrée/sorties.

On peut faire des scripts en shell.

L'arborescence des fichiers

Les fichiers sont organisés sous forme arborescente.

- La racine : /
- Deux principaux types de fichiers :
  - les fichiers standards = fichiers réguliers
  - les répertoires (ou dossiers).
  - (Il en existe d'autres : liens, fifo... à suivre)
- Chaque fichier possède :
  - un propriétaire et un groupe propriétaire
  - un ensemble de droits (lecture/écriture/exécution) pour l'utilisateur, le groupe, et le reste du monde.
  - une date de création, de modification, de lecture.
- Fichiers commençant par un point : fichiers cachés.

L'arborescence des fichiers

Répertoires spéciaux :

- `..` : répertoire parent
- `.` : répertoire courant

Chemin de la racine à un fichier : chemin absolu

- `/home/mrao/Documents/cours.pdf`

Chemin du répertoire courant à un fichier : chemin relatif

- `Documents/cours.pdf`
  = `./Documents/cours.pdf`

Organisation typique sous Unix/Linux

- /home : les répertoires des utilisateurs
- /root : le répertoire "home" du super-utilisateur
- /bin et /usr/bin les programmes (les "binaires") ;
- /sbin et /usr/sbin : les binaires système
- /lib et /usr/lib : librairies
- /usr : ressources système
- /etc : fichiers de configurations
- /dev : fichiers spéciaux (ressources, périphériques)
- /tmp : un répertoire pour les fichiers temporaires
- /var : données variables
- ...

Les shell : premiers pas...

Le shell permet de naviguer dans l'arborescence de fichiers, modifier les droits, faire des opérations simples.

- `ls` : liste les fichiers
  - option `-a` : affiche également les fichiers cachés
  - option `-l` : format long (droits, taille, propriétaire...)
- `cd` *`rep`* : entrer dans le répertoire *rep*
  - `cd ..` : retour au répertoire parent

Autres commandes de base

- cat : affiche un fichier
- rm : efface un fichier
- cp : copie un fichier
- mv : déplace (renomme) un fichier
- mkdir/rmdir : crée/efface un répertoire
- ...

Un peu plus sur les droits des fichiers

```
mrao@meshuggah:~/test$ ls -la
total 32
drwxr-xr-x  4 mrao users 4096 dec.  29 22:40 .
drwx------ 61 mrao users 4096 janv.  9 17:25 ..
-rwxr-xr-x  1 mrao users 6656 dec.  29 13:37 programme
-rw-r--r--  1 mrao users  173 dec.  29 13:37 programme.c
drwxr-xr-x  2 mrao users 4096 dec.  29 13:39 sousrep
mrao@meshuggah:~/test$
```

premier champ : un sous ensemble de drwxrwxrwx

- d : répertoire
- 1er triplet (rwx) : droits pour l'utilisateur (ici, mrao)
- 2eme triplet : droits pour les utilisateurs du groupe (users)
- 3eme triplet : droits pour le reste du monde
- r : droit de lecture
- w : droit d'écriture
- x : droit d'exécution (pour les répertoires : droit d'entrer)

Pour changer les droits : `chmod`

Liens

Unix supporte des liens. Il y a deux types de liens, fondamentalement différents.

- Lien symbolique : un "pointeur" vers un autre fichier.
  Il s'agit d'un type de fichier spécial.
  Commande shell : `ln -s`. Appel système : `symlink`

- Lien physique : fichier correspondant à la même zone sur le disque qu'un autre.
  Commandes shell : `ln`, `link`. Appel système : `link`

Autres fichiers spéciaux

- Fichier périphérique (device file).
  Correspond à un périphérique
  Généralement situé dans `/dev/`

- Fichiers tubes (ou fifo).
  Pour créer un fichiers tube : `mkfifo`
  (On reparlera de tubes au moment de la communication inter-processus.)

Un peu plus sur les système de fichiers

- L'arborescence des fichiers est un "patchwork" de systèmes de fichiers.
- Un système de fichier correspond généralement à une partition sur un disque sur.
- plusieurs types de systèmes de fichiers : FAT, EXT, NTFS...
- commandes : `mount`, `umount`, `df`

Utilisateurs et groupes

Chaque utilisateur a :

- un nom (une chaîne de caractère)
- un numéro (UID = User IDentifier)
- un ou plusieurs groupes
- un répertoire HOME, généralement : `/home/<user>`
- un mot de passe, stocké de façons hashée.

`root` est le "super-utilisateur". Il a tous les droits. Son UID est 0.

Chaque groupe a un numéro (GID = Group IDentifier).

Les processus

Chaque processus a :

- un numéro (le PID = Process IDentifier)
- un père, généralement le processus qui l'a lancé.
- un utilisateur (généralement, celui qui l'a lancé)
- une zone mémoire qui lui a été attribué. Il peut en demander plus au système.
- certains processus peuvent être en attente.
- une entrée standard, et une sortie standard et une sortie erreur.
- À sa fin, un processus renvoie un code retour : un entier, généralement 0 s'il n'y a pas d'erreur, et $\neq 0$ sinon.

Lancer des commandes/processus dans un shell

- Certaines commandes sont interprétés directement par le shell, les builtin (comme `cd`). D'autres correspondent à des programmes exécutables (généralement situés dans `/bin/` ou `/usr/bin/`).
- S'il le trouve, il l'exécute (le processus se lance). Sinon il renvoie un message d'erreur.
- Pour lancer un programme dans le répertoire courant il faut spécifier le répertoire avant le nom du programme.
- Les programmes sont cherchés dans les répertoires listés dans la variable d'environnement PATH

Entrées/sorties standard

Quand un processus s'exécute dans un terminal :

- l'entrée standard est (par défaut) l'entrée du terminal (le clavier)
- la sortie standard est (par défaut) affichée dans le terminal.
- la sortie erreur est (par défaut) affichée dans le terminal.

Le shell permet de facilement rediriger ces entrées/sorties.

Rediriger les E/S standards

- `commande > ficher` : redirige la sortie standard de la commande vers le fichier (écrase le fichier)
- `commande >> fichier` : redirige la sortie standard de la commande vers le fichier (rajoute à la fin du fichier)
- `commande 2> fichier` : redirige la sortie erreur de la commande vers le fichier
- `commande1 | commande2` : la sortie standard de commande1 sera regirigée vers l'entrée standard de commande2

Rediriger les E/S standards

- `tee` *`fichier`* : copie entrée standard sur la copie standard et *fichier*
- `commande < fichier` : l'entrée standard sera lue depuis le fichier
- `commande << EOF` : le shell va lire l'entrée standard, jusqu'à ce qu'il lise EOF. Ce qui est lu sera envoyé dans l'entrée standard de commande.

Quelques commandes utiles

- `sleep x` ; attend x seconde (utile pour les scirpts)
- `echo` : affiche les arguments
- `less` : permet de se déplacer dans le texte
- `head/tail` : affiche le début/fin de l'entrée
- `sort` : trie les ligne
- grep : afficher les lignes correspondant à un motif donné
- `sed` : fait des recherches / remplacements
- `awk` : un truc qui fait mieux que grep / sed, mais encore plus compliqué.

Voir/gérer les processus :

Commandes shell pour voir/gérer les processus :

- `ps` affiche la liste des processus
  exemple : `ps faux`
- `top` affiche la liste des processus dynamiquement
- `kill` *`pid`* : tue un processus de PID *pid* (on en reparlera dans la partie "Signaux")

Variables du shell

- Le shell manipule des variables.
- Exemples : HOME, USER, PATH
- Affecter une variable :
  - `VARIABLE=affectation`
- déréférencer une variable : la faire précéder par $
  - Ex : pour afficher une variable : `echo $VARIABLE`
- `set` : affiche toutes les variables

Certaines variables sont persistantes : les variables d'environnement. Elles seront transmises aux fils

- `export` : exporte la variable (les rend persistantes)
- `env` : affiche toutes les variables d'environnement.

Variables spéciales du shell

- $? : code retour de la précédente commande
- $$: PID du shell
- $! : PID du dernier processus lancé en arrière plan
- ~ : interprété par le shell comme le répertoire HOME
- ~user : interprété par le shell comme le répertoire HOME de l'utilisateur user

Jokers et echappements

- `*` dans un nom de fichier : n'importe quelle chaîne de caractère
- `?` : exactement un caractère

Pour qu'un caractère spécial ne soit pas interprété par le shell

- `\` : exemple `echo \*`
- `'` : exemple `echo '*'`
- `"` : exemple `echo "*"`
  - le shell interprète les $ et certains \dans des chaînes entre "

Le caractère "espace" peut être aussi echappé, pour ne pas séparer les arguments

Processus en arrière plan

- `commande &` lance un processus, mais le shell n'attend pas la fin du processus pour rendre la main.
- ctrl + z : stoppe un processus
- `jobs` : liste les taches (jobs) en cours d'exécution dans le shell
- `bg` : passe une tache en arrière plan (similaire à &)
- `fg` : passe une tache en premier plan (le shell rend la main au job)
- `%i` : identifie le job numéro *i* du shell.

nohup

- Si on termine un shell, tous ses jobs seront arrêtés.
- Pour qu'un processus survive aux déconnexions, à la mort de son père, on peut le lancer précédé de la commande `nohup`

Scripts : enchaînement et composition des commandes

- *commande1* ; *commande2*
  execute commande1, puis commande2
- (*listedecommandes*)
  crée un groupement de commande
- *commande1* `*commande2*`
  la sortie de commande2 est donnée en argument à commande1

  Sur certains shells, on peut également faire ceci :
  commande1 $(*commande2*)

Scripts : enchaînement et composition des commandes

- *commande1* `&&` *commande2*
  execute commande1, puis commande2 si commande1 réussi (i.e. renvoie 0)
- *commande1* `||` *commande2*
  execute commande1, puis commande2 si commande1 échoue
- `if` *condition* `; then` *commande2* `; else` *commande3* `;fi`

Scripts : tests

`test` *expression* permet de tester une expression conditionnelle.

Note : sur certains shells, c'est équivalent à `[` *expression* `]`

expression construite avec `(` `)` `&&` `||` `!` et des expression élémentaires.

Expression élémentaire (exemples) :

- tester si un fichier existe : `-e` *fichier*
- tester si un fichier est un répertoire : `-d` *fichier*
- tester si un fichier est un fichier régulier : `-f` *fichier*
- tester si un fichier est lisible : `-r` *fichier*
- tester si deux chaînes de caractères sont égales : *chaine1* `=` *chaine2*
- tester si expression numériques sont égales : *chaine1* `-eq` *chaine2*

Scripts : évaluer une expression

`expr` *`expression`* permet d'evaluer une expression

expression construite avec `( ) + - * / % = >=` ... et des expressions élémentaires (entiers...)

Attention aux echappements : `expr \( 2 + 3 \) \* 5`

Sous bash on peut utiliser directement `$((`*`expression`*`))`

Scripts : boucles

`while` *`condition`* `; do` *`commandes`* `; done`

- Ex : `while true ; do date ; sleep 1; done`

`for` *`v`* `in` *`liste`*`; do` *`commandes`* `; done`

- entre do et done, v est une variable
- Ex :
  `for f in *wav; do lame $f ‘basename $f wav‘mp3 ; done`
- `seq` *`a b`* : tous les entiers entre a et b.

Voir également : `break`, `continue`, `until`, `case`

Scripts : fichiers scripts

```
#!/bin/bash
for i in "$*" ; do
  echo $i
done
```

- #! : shebang : dit au système quel interpréteur utiliser
- $1 : 1er argument, $2 : 2eme argument ...
- $* : tous les arguments
- $# : nombre d'arguments

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;
}
```

Il y a d'autres façons de voir les variables d'environnement

- `extern char ** environ;`
- `setenv`, `getenv`...

Exercice 0.5

Que fait le code suivant ?

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main() {
  char buffer[100];
  int r,fd;
  fd=open("hello.txt",O_RDONLY);
  if(fd<0) {perror("open");return 1;}
  r=read(fd,buffer,100);
  if(r<0) {perror("open");return 1;}
  buffer[r]=0;
  printf("%s",buffer);
  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 fonctions 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);
```

Descripteur de fichier

- Un descripteur de fichier est un entier ≥ 0
- Chaque processus possède sa table de descripteurs de fichiers
- Attention : la table a une taille limitée
- Chaque entrée de la table pointe sur un fichier ouvert (ou autres "choses" qu'on peut lire et/ou écrire, comme des terminaux, des connections réseau...)
- Des entrées différentes (d'un même processus, ou de processus différents) peuvent pointer sur le même fichier ouvert.
- Un fichier peut être ouvert plusieurs fois

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`, `ftruncate`
- 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

```c
#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 n'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 données statiques
  - 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)

Processus

Les processus : Rappels (?)

Un processus :

- a un numéro (le PID = Process IDentifier)
- a un père.
- a un propriétaire (réel et effectif)
- a un état : en exécution, en sommeil, stoppé ou zombie
- a un niveau de priorité
- un répertoire courant
- (certains processus) peuvent être “légers” (“threads”) (on y reviendra dans quelques cours)
- une table de descripteurs de fichiers
- une table de pages
- renvoie un code retour (un entier entre 0 et 255)

Voir les processus

- Pour voir tous les processus tournant, avec leur lien de parenté : `ps faux`
- Pour voir en temps réel (utilisation CPU, mémoire...) : `top`
- Toutes les informations dans le système procfs (`/proc/`)

État d'un processus

Un processus peut être :

- actif (i.e. en exécution)
- prêt (en attente d'exécution)
- suspendu (par exemple avec ctrl+z)
- en sommeil : il attend un évènement
  - sleep, pause...
  - attente sur une I/O
- zombi

Ordonnancement

Deux processus ne peuvent pas s'exécuter en même temps sur un même coeur.

L'OS découpe le temps en petits bouts, et fait tourner les processus les uns après les autres.

L'OS choisi l'ordre, en essayant de respecter la priorité des processus (voir `nice`)

Passage d'un processus à un autre : commutation de contexte (context switch).

- assez lent...
- transparent pour nous

Gestion de processus : syscalls

- `getpid()` renvoie le PID du processus courant
- `getppid()` renvoie le PID du père
- `getuid()` renvoie l'UID de l'utilisateur processus courant
- `geteuid()` renvoie l'UID de l'utilisateur effectif du processus courant
- Ces UIDs peuvent être différents si le binaire est en "setuid"
- `setuid()` et `seteuid()` permettent de changer les utilisateurs, si on a les droits !

- Obtenir/changer le répertoire courant : `getcwd()` / `chdir()`
- Obtenir le temps CPU consommé (en mode utilisateur et système) : `times()`
- Modifier le masque de création de fichiers : `umask()`
- Pour voir/changer les "limites" d'un processus : `ulimit`, `getrlimit`, `setrlimit`

Priorité d'un processus

- Chaque processus a une priorité :
  - généralement, un nombre entre -20 et 19, et par défaut : 0.
  - plus le nombre est élevé, moins le processus aura du temps de calcul.
- Il est possible de lancer un processus avec une priorité plus basse avec `nice`
- Il est possible de diminuer la priorité d'un de ses processus en exécution :
  - Commande : `renice`
  - Appel système : `nice`
- Seul `root` a le droit d'augmenter une priorité.

Comment lancer un processus ?

Il faut différencier le fait de :

- créer un nouveau processus : le processus appelant continue de vivre, et un nouveau processus naît
- exécuter un binaire spécifié : il n'y a pas de nouveau processus, l'ancien processus est "écrasé" par le nouveau

Créer un nouveau processus (création) se fait avec `fork`

Exécuter un binaire (recouvrement) se fait avec `exec...`

Créer un nouveau processus qui est l'exécution d'un binaire se fait avec la combinaison de `fork` et `exec...`.

`int system(const char *command)`

(Pas un appel système. Donné à titre informatif.)

Un moyen simple de lancer un processus depuis un programme est d'utiliser `system` (dans <stdlib.h>).

`system` lance un shell (`/bin/sh`) qui exécutera `command`. Une fois la commande terminée, la fonction retournera le code retour de la commande.

Exemple : `system("ls");`

Les exec*

```
#include <unistd.h>
int execl(const char *path, const char *arg, ...);
int execlp(const char *file, const char *arg, ...);
int execle(const char *path, const char *arg, ...,
    char * const envp[]);
int execv(const char *path, char *const argv[]);
int execvp(const char *file, char *const argv[]);
int execvpe(const char *file, char *const argv[],
    char *const envp[]);
```

- Elles ne créent pas un nouveau processus : elles remplacent (recouvrent) le processus courant par l'exécution du fichier en argument.
- En particulier, le PID, L'UID, les fichiers ouverts sont conservés (sauf si option O_CLOEXEC)
- Si l'exécution se fait normalement, elles ne retournent jamais !

`pid_t fork(void)`

`fork()` (dans <unistd.h>) est la commande pour lancer un nouveau processus.

Elle duplique le processus courant, pour créer un processus fils.

- Dans le père, elle renvoie le PID du fils (et rien ne change)
- Dans le fils (le nouveau processus) :
  - elle retourne 0
  - le fils aura un nouveau PID
  - son père sera le processus père

`pid_t fork(void)`

- `fork` retourne donc deux fois, une fois dans le père, une fois dans le fils
- Tout se passe comme si toute la mémoire du processus appelant fork est copiée.
- (En pratique, le système copie seulement si c'est nécessaire.)
- Les deux processus sont concurrents. On ne peut pas dire lequel des deux retournera en premier.

Exemple : fork

```c
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    /*...*/
    return 0; /* fin du fils */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

fork et descripteurs de fichiers

- Lors d'un fork, la table des descripteurs du processus est copiée.
- Les descripteurs des deux processus (père et fils) référencient les mêmes fichiers ouverts par le système (comme après un `dup`)
- En particulier, si un des deux processus modifie le curseur d'un descripteur (read/write/lseek...), cela affectera le curseur du même descripteur de l'autre processus

fork et descripteurs de fichiers

```
int main()
{
  int fd=open("sortie.txt",O_CREAT | O_RDWR
      ,0644);
  if(fork()==0) {
    write(fd,"A",1);
    close(fd);
    return 0;
  }
  write(fd,"B",1);
  close(fd);
  return 0;
}
```

sortie.txt : `AB` ou `BA`

Exemple : fork + exec

```c
#include <unistd.h>

int main()
{
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'
        execlp a echoue */
  }
  /* si on est ici, on est le pere */
  /*...*/
  return 0; /* fin du pere */
}
```

`pid_t wait(int *ptr)`

`wait` : attend jusqu'à ce qu'un processus fils termine.

Plus précisément :

- Si un processus fils termine avant l'appel de `wait` de son père, il devient zombi.
- Si un processus n'a pas de fils : `wait` renvoie -1.
- Si un processus a un fils zombi : `wait` renvoie le PID du fils zombi, et efface ce processus de la liste des processus.
  - Si `ptr` n'est pas NULL, `wait` copie le "statut" dans l'entier pointé par `ptr` (voir man).
- Si un processus a des fils, mais pas pas de fils zombi : `wait` attend jusqu'à ce qu'un fils devienne zombi, puis idem.

Attente d'un processus particulier :

```
pid_t waitpid(pid_t pid, int *status, int options);
```

Exemple : fork + exec + wait

```c
#include <unistd.h>

int main()
{
  int status;
  if(fork()==0) {
    /* si on est ici, on est le fils */
    execlp("xeyes","xeyes",NULL);
    return 1; /* si on se trouve ici, c'est qu'
        execlp a echoue */
  }
  /* si on est ici, on est le pere */
  wait(&status);
  return 0; /* fin du pere */
}
```

Exercice

Que fait le code suivant

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

int main() {
  int status;
  if (fork())
    wait(&status);
  else
    if (!fork())
      execlp("xeyes","xeyes",NULL);
  return 0;
}
```

`pid_t setsid(void)`

Un processus peut se détacher de son père en appelant `setsid()`.

Plus précisément, cette fonction sert à créer une nouvelle session.
Le processus appelant devient leader de cette session.

Communication inter processus : avant goût

IPC = Inter Processus Communucation

Comment faire communiquer des processus ?

- via les entrées sorties : pas dynamique
- fichier standards : archaïque, lent, problèmes de synchronisation
- fichiers tubes
- Signaux : information très limitée (mais ça sert à plein de niveau)
- partage de mémoire
- par un canal réseau ...

[C : Pointeurs sur fonctions]

- Une fonction dispose également d'une adresse mémoire
- C'est son point d'entrée , i.e. l'adresse mémoire où commence la liste des instructions en langage machine
- Il est possible de manipuler les adresses des fonctions en C, et d'avoir des pointeurs sur des fonctions
- Il est obligatoire de savoir manipuler les pointeurs sur fonctions pour gérer les signaux et les threads...

Syntaxe en C

Le type d'un pointeur sur fonction doit contenir les types des paramètres de la fonction, et le type de retour.

- les paramètres n'ont pas besoin d'avoir de nom :
- le compilateur doit juste savoir quel type empiler sur la pile

Pour déclarer un pointeur sur une fonction :
`type_retour (*nompointeur) (liste_arguments...);`

Syntaxe en C

Exemple :

```
int (*fct) (int);
```

déclare fct comme étant un pointeur sur une fonction prenant en argument un entier, et revoyant un entier

Appeler une fonction pointée se fait de la même manière que pour une fonction normale.

Exemple

```
int carre(int x) {return x*x;}

int cube(int x) {return x*x*x;}

void iter(int (*fct)(int)) {
  int i;
  for(i=1;i<=10;i++)
    printf("%d : %d\n",i,fct(i));
}

void main() {
  int (*x)(int);
  x=carre;
  iter(x);
  iter(cube);
}
```

avec `typedef`

On peut simplifier les choses avec `typedef` :

- `typedef int (*typefctintint) (int);`

Définit `typefctintint` comme étant le type pointeur sur une fonction int $\rightarrow$ int ;

- `typefctintint fct=carre;`

Exemple 1 : atexit

`atexit` enregistre une fonction qui sera appelée à la fin (normale) du processus (après un `exit`, ou au retour du `main`)

```
#include <stdlib.h>
int atexit(void (*function)(void));
```

Exemple 2 : qsort

`qsort` est une fonction de la libc effectuant un *quick sort*.

```
#include <stdlib.h>
void qsort(
   void *base,
   size_t nmemb,
   size_t size,
   int (*compar)(const void *, const void *)
   );
```

On doit passer en argument l'adresse de la fonction de comparaison (`compar`) que `qsort` doit utiliser.

Les signaux

Les signaux : introduction

- Les signaux permettent de notifier des évènements à un processus.
- Il s'agit d'un moyen de communication limité :
  - ponctuel
  - unique information : le numéro du signal, un entier entre 1 et (généralement) 64.
- Mais très important sous Unix.

Les signaux : introduction

Beaucoup de mécanismes sous Unix utilisent des signaux. Par exemple :

- `ctrl + c` (arrêt d'une tache)
- `ctrl + z` (mise en pause d'une tache)
- Tuer un processus par `kill`
- Erreur de segmentation
- Division par 0...

Signaux standard

- SIGTERM (15) : Signal de fin (signal par défaut de `kill`)
- SIGINT (2) : Terminaison depuis le clavier (ctrl + c)
- SIGKILL (9) : Tuer un processus (on ne peut pas le contourner)
- SIGSTOP (19) : Arrêt (pause) du processus (ctrl + z)
- SIGCONT (18) : Continuer si en pause
- SIGALRM (14) : Temporisation `alarm` écoulée.
- SIGUSR1 (10) : Signal utilisateur 1.
- SIGUSR2 (12) : Signal utilisateur 2.
- SIGCHLD (17) : Fils arrêté ou terminé

Les erreurs :

- SIGFPE (8) : Erreur mathématique virgule flottante.
- SIGPIPE (13) : Écriture dans un tube sans lecteur.
- SIGSEGV (11) : Référence mémoire invalide.
- SIGILL (4) : Instruction illégale.
- ...

Signaux générés

Un signal est généré par un évènement :

- Envoi d'un signal par un autre processus
- Action sur le terminal (ctrl+c, ctrl+z...)
- Erreur (arithmétique, de segmentation ...)
- Minuterie
- Arrêt ou terminaison d'un fils...

Lorsque le signal est délivré à un processus, une action se produit :

- action par défaut
- signal ignoré
- effectuer une action choisie : handler

Un signal généré, mais pas (encore) délivré, est pendant

Si le même signal est généré plusieurs fois, on est pas sûr qu'il sera délivré le même nombre de fois

Envoyer un signal

Depuis le shell : `kill -sig pid`, où :

- *sig* est le signal : le nom (KILL, STOP,CONT...) ou numérique
- *pid* est le PID du processus à qui on lance le signal

Appels systèmes :

```
#include <sys/types.h>
#include <signal.h>
int kill(pid_t pid, int sig);
int raise(int sig);
unsigned int alarm(unsigned int s)
```

`kill` envoie le signal `sig` au processus `pid`.
`raise` envoie le signal `sig` au processus courant.
`alarm` envoie le signal SIGALRM s secondes plus tard. (Si $s = 0$, annule l'alarme)

Réception : comportement par défaut

À la réception d'un signal, un comportement par défaut est définit. Celui-ci peut être :

- Terminer le processus
  - KILL, TERM, ALARM, INT, FPE, PIPE, USR1, USR2...
- Terminer le processus avec fichier `core`
  - ILL, SEGV
- Signal ignoré
  - CHLD
- Suspension du processus
  - STOP
- Continuation du processus
  - CONT

Il est possible d'ignorer ce comportement par défaut, ou d'en définir un autre, pour tous les signaux, sauf SIGSTOP et SIGKILL

Ensemble de signaux

`sigset_t` est un type pour un ensemble de signaux. Une variable de ce type peut être manipulé par les fonctions suivantes.

```
#include <signal.h>
int sigemptyset(sigset_t *set);
int sigfillset(sigset_t *set);
int sigaddset(sigset_t *set, int signum);
int sigdelset(sigset_t *set, int signum);
int sigismember(const sigset_t *set, int signum
    );
```

Masquer des signaux

```
#include <signal.h>
int sigprocmask(int how, const sigset_t *set,
    sigset_t *oldset);
```

- how :
  - SIG_SETMASK : nouveau masque = set
  - SIG_BLOCK : nouveau masque = ancien masque $\cup$ set
  - SIG_UNBLOCK : nouveau masque = ancien masque $\setminus$ set
- Masquer un signal ne veut pas dire l'ignorer.
- Si un signal masqué est généré, il reste pendant, sauf si le comportement par défaut est de l'ignorer.

Lister les signaux pendants

```
#include <signal.h>
int sigpending(sigset_t *set);
```

Copie dans set la liste des signaux pendants.

C'est particulièrement utile si des signaux sont masqués (et non ignorés).

Changer le comportement : `signal`

On peut demander à exécuter une fonction (handler) en cas de réception d'un signal.

L'ancienne interface Unix (non POSIX) est la suivante. Donnée à titre indicatif (car plus simple à comprendre). Ne pas utiliser.

```
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t
    handler);
```

`handler` est soit :

- SIG_IGN : ignore le signal
- SIG_DFL : action par défaut
- l'adresse d'une fonction prenant un entier
  - à la réception d'un signal, la fonction sera appelée, avec comme argument le numéro du signal.

Changer le comportement : `signal`

```c
void handler(int i)
{
    printf("signal recu : %d\n",i);
}

int main()
{
    signal(SIGUSR1,handler);
    signal(SIGUSR2,handler);
    sleep(10000);
    return 0;
}
```

sigaction

L'interface à utiliser pour manipuler les handlers est `sigaction`

```
struct sigaction {
 void      (*sa_handler)(int);
 void      (*sa_sigaction)(int, siginfo_t *,
    void *);
 sigset_t    sa_mask;
 int         sa_flags;
};

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

`sigaction`

- sa_handler : le handler (comme `signal`)
- sa_flags : options (void man)
- sa_mask : liste des signaux à masquer en plus, le temps que le handler s'exécute
- Si `act` n'est pas NULL : nouveau handler à installer
- Si `oldact` n'est pas NULL : l'ancien handler est copié dans la structure pointée
- On peut utiliser `sa_sigaction` à la place de `sa_handler` pour avoir un comportement plus fin (voir man).

Attente de signaux

```
#include <unistd.h>
int pause(void);

#include <signal.h>
int sigsuspend(const sigset_t *mask);
```

- `pause` met le processus en pause, jusqu'à ce qu'un signal (n'importe lequel) arrive.
- Problème : un signal non masqué peut arriver avant l'appel à pause(), et être "perdu"...
- `sigsuspend` change temporairement le masque des signaux masqués, et attend jusqu'à ce qu'un signal arrive.

Signaux et appels systèmes

- Certains appels systèmes peuvent être interrompus par un signal.
  - Dans ce cas, l'appel système échoue, et le code retour (errno) sera EINTR
- Lors d'un fork, les signaux pendants ne sont pas hérités (le masque et les handlers, si)
- Lors d'un exec, les handlers ne sont pas hérités (le masque et les signaux pendants, si)

Communication Inter Processus (IPC) : Tubes

Principe

- À partir de maintenant, on veut faire communiquer plusieurs processus.
- Un tube est un moyen de le faire.

Note :

- Faire communiquer des processus sur une même machine par tubes peut sembler archaïque, et pas très efficace (comparé à la mémoire partagée).
- Mais : les communications réseau (par sockets) se feront de manière similaire
- les principes/fonctions expliqués dans ce chapitre seront toujours valables.

Principe

- Tube : canal de communication FIFO (First In First Out)
- Utilise 2 descripteurs de fichiers : un pour l'écriture (l'entrée), et un pour la lecture (la sortie)
- L'écriture dans l'entrée sera mise en attente dans un tampon
- La lecture dans la sortie lira les données du tampon, dans l'ordre (FIFO).
- La lecture et l'écriture se font comme pour les fichiers réguliers : read et write
- Il n'y a pas de "tête" : lseek est impossible !

Principe

Par exemple, lorsque l'on exécute :

```
cat fichier.txt | grep password
```

- il y a deux processus crées : un pour `cat` et un pour `grep`
- ils communiquent via un tube
- la sortie standard de `cat` sera le côté "écriture" du tube
- l'entrée standard de `grep` sera le côté "lecture" du tube

Principe

Mais cela ne se fait pas dans l'ordre "création processus", puis "création tubes"...

```
cat fichier.txt | grep password
```

- le shell crée un tube
- le shell lance deux nouveaux processus (deux `fork()`) : un pour `cat` et un pour `grep`
- la sortie standard de `cat` est ecrasée par le côté "écriture" du tube (via par exemple `dup2`)
- l'entrée standard de `grep` est écrasée par le côté "lecture" du tube
- les fils se recouvent (`exec...`) en cat et grep.

Principe

```
cat file.txt | grep passwd | sed 's/.*passwd=\(\w*\).*/\1/'
```

```
$ lsof
...
cat   5223   mrao   0u   CHR   136,1   0t0   4       /dev/pts/1
cat   5223   mrao   1w FIFO    0,10   0t0   27854   pipe
cat   5223   mrao   2u   CHR   136,1   0t0   4       /dev/pts/1
...
grep  5224   mrao   0r FIFO    0,10   0t0   27854   pipe
grep  5224   mrao   1w FIFO    0,10   0t0   27856   pipe
grep  5224   mrao   2u   CHR   136,1   0t0   4       /dev/pts/1
...
sed   5225   mrao   0r FIFO    0,10   0t0   27856   pipe
sed   5225   mrao   1u   CHR   136,1   0t0   4       /dev/pts/1
sed   5225   mrao   2u   CHR   136,1   0t0   4       /dev/pts/1
```

Créer un tube (par syscall)

```c
#include <unistd.h>

int pipe(int pipefd[2]);
```

- Ouvre les 2 descripteurs de fichier associés a un nouveau tube
- Prend en argument un tableau de deux entiers :
- Renvoie dans `pipefd[0]` la sortie du tube (le descripteur en lecture)
- Renvoie dans `pipefd[1]` l'entrée du tube (le descripteur en écriture)

Exemple : pipe

```
main() {
   int fd[2],r;
   char buffer[10];

   pipe(fd);

   r=write(fd[1],"hello",5);
   assert(r==5);

   r=read(fd[0],buffer,10);
   assert(r==5);

   buffer[r]=0;
   printf("recu: %s\n",buffer);
}
```

Exemple : pipe + fork

Créer un tube nommé ("fichier tube")

- Un autre moyen de créer un tube est de créer et ouvrir un "tube nommé"
- Il s'agit d'un fichier spécial (non "régulier")
- Commande shell pour créer un tube nommé : `mkfifo`.
- Appels systèmes : `mkfifo` ou `mknod`.

Quand un fichier tube est ouvert en lecture, et ouvert par un autre processus en ecriture, le comportement sera le même qu'un tube crée par `pipe`

Mode "flot" (stream)

Mode flot : les envois successifs d'informations s'additionnent.
Il n'y a pas de "séparations" entre elles.

Exemple :

- `write(in,"ABC",3)`
  - le tube contient "ABC"
- `write(in,"123",3)`
  - le tube contient "ABC123"
- `read(out,bf,4)`
  - renvoie 4, et bf contient "ABC1"
  - le tube contient "23"
- `read(out,bf,4)`
  - renvoie 2, et bf contient "23"
  - le tube est vide
- `read(out,bf,4)`
  - bloque jusqu'à ce qu'un processus écrive dans le fifo...

Nombre de lecteur et écrivains

- Un tube peut avoir un nombre de lecteur (ou d'écrivain) différent de un.
- Si un tube a 0 lecteur : l'écriture échouera (signal SIGPIPE)
- Si plus d'un lecteur : premier arrivé, premier servi
- Si un tube a 0 écrivain (et le tube est vide), la lecture renverra 0 (i.e. comme pour un fin de fichier)

- Comme toujours, on ferme les descripteurs qui ne servent plus.

Caractère bloquant

- Par défaut, la lecture dans un tube vide sera bloquant
- Il est possible de rendre la lecture non bloquante, en changeant l'option O_NONBLOCK du descripteur de fichier
- Dans ce cas, la lecture dans un tube vide échouera (retour -1), avec errno = EAGAIN
- Attention, un tube a aussi une capacité limitée (PIPE_BUF=4096).
  Quand un tube est plein, une écriture sera également bloquante.

Manipuler un descripteur de fichier : fcntl

`fcntl` permet de manipuler les descripteurs de fichiers.

Elle permet (entre autres) de changer les options (modes) des descripteurs de fichiers. En particulier :

- O_NONBLOCK : caractère non bloquant d'un descripteur

Pour passer un descripteur en mode non bloquant :

```
int flags = fcntl(fd, F_GETFL, 0);
fcntl(fd, F_SETFL, flags | O_NONBLOCK);
```

Attente sur plusieurs descripteurs de fichiers : select

`select` permet d'attendre (avec un temps limite) sur un ensemble de descripteurs de fichiers en un seul appel.

Pour utiliser `select`, il faut au préalable manipuler une structure qui représente un ensemble de descripteurs de fichiers. Cela se fait via les primitives suivantes :

```
#include <sys/select.h>

void FD_CLR(int fd, fd_set *set);
int  FD_ISSET(int fd, fd_set *set);
void FD_SET(int fd, fd_set *set);
void FD_ZERO(fd_set *set);
```

Attente sur plusieurs descripteurs de fichiers : select

```
#include <sys/select.h>

int select(int nfds, fd_set *readfds, fd_set *
    writefds, fd_set *exceptfds, struct timeval
    *timeout);
```

- nfds : le plus grand descripteur de fichiers à vérifier +1
- readfds : l'ensemble des descripteurs à vérifier en lecture
- writefds : l'ensemble des descripteurs à vérifier en ecriture
- execptfds : l'ensemble des descripteurs à vérifier en exception
- timeout : temps maximal à attendre.

À sa sortie, `select` modifie les ensembles de telle façon qu'il ne reste que les descripteurs de fichiers sur lesquels il y a quelque chose à lire ou écrire.

Attente sur plusieurs descripteurs de fichiers : poll

`poll` permet également d'attendre sur un ensemble de descripteurs de fichiers, mais plus finement.

```
#include <poll.h>

int poll(struct pollfd *fds, nfds_t nfds, int timeout);

struct pollfd {
 int    fd;          /* file descriptor */
 short  events;      /* requested events */
 short  revents;  /* returned events */
};
```

- fds : un table de pollfd à surveiller,
- nfds : taille de fds
- timeout : temps maximum (en millisecondes)
- cmdevents et `revents` sont des conjonctions de :
  - POLLIN : il y a quelque chose à lire
  - POLLOUT : il est possible d'y écrire
  - POLLERR : il y a une erreur
  - POLLHUP : pipe ou socket fermé de l'autre côté

Un premier pas vers les communications réseau

Une socket est un point de communication où il est possible d'envoyer et de recevoir des informations.
On en reparlera longuement au moment de la programmation réseau

Les sockets communiquent par pair. Il y a plusieurs moyens de les faire communiquer (différents protocoles réseau, ou en local).

On peut créer une paire de socket en communication locale, qui fonctionnera similairement deux tubes :

- l'entrée de la 1ere socket sera l'entrée du 1er tube et la sortie de la 2eme socket sera la sortie du 1er tube
- l'entrée de la 2eme socket sera l'entrée du 2eme tube et la sortie de la 1ere socket sera la sortie du 2eme tube

Un premier pas vers les communications réseau

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

int socketpair(int domain, int type, int
    protocol, int sv[2]);
```

Crée 2 sockets associées.
Pour le faire via une communication locale :

- domain = AF_UNIX
- protocol = 0
- type :
  - SOCK_STREAM : communication par flot (comme pour les tubes)
  - SOCK_DGRAM : communication en mode paquet
- sv : un tableau de 2 entiers, pour le renvoi des 2 descripteurs de fichiers (les 2 sockets)

mode paquet (DGRAM)

Au contraire du mode flot (STREAM), chaque information envoyée constitue une entité indivisible.
Exemple :

- `write(in,"ABC",3)`
  - la file de messages contient "ABC"
- `write(in,"123",3)`
  - la file contient "ABC","123"
- `read(out,bf,10)`
  - renvoie 3, et bf contient "ABC"
  - la file contient "123"
- `read(out,bf,10)`
  - renvoie 3, et bf contient "123"
  - la file vide
- `read(out,bf,4)`
  - bloque jusqu'à ce qu'un processus écrive dans le socket...

Autres IPC

D'autres moyens de communication inter-processus existent (POSIX et SysV).

Nous n'ont parlerons pas, car les mécanismes sont similaires à des mécanismes déjà vus (pipe/socket), ou que l'on verra plus tard (threads)

Ce sont :

- Les files de messages (POSIX : `man mq_overview`)
  - Similaire aux sockets en mode paquet (DGRAM)

Autres IPC

- La mémoire partagée (POSIX : `man shm_overview`)
  - Un segment mémoire est partagé entre plusieurs processus. C'est un moyen de communication très rapide (au sein d'une même machine), mais il faut faire attention aux synchronisations.
- Les sémaphores (POSIX : `man sem_overview`)
  - Il s'agit de mécanisme de synchronisation (exclusion mutuelle). On parlera de sémaphores et mutex en même temps que les threads.

Pour voir les mécanismes System V : `man svipc`

Bonnes pratiques, débogage et optimisation

On va voir :

Quelques bonnes et mauvaises pratiques de programmation

Outils de débogage :

- gdb
- valgrind

Outils de "profilage" :

- gprof
- gcov

Options utiles de gcc

Les "bugs"

Des bugs (cachés ou non) peuvent avoir de conséquences fâcheuses :

- plantages (aléatoires), pertes de données
- "exploitations" : porte d'entrée aux problèmes de sécurités

Lorsqu'un "bug" arrive :

- c'est (généralement) votre faute !

S'il un programme s'exécute sans "bug" :

- cela n'implique pas que vous avez bien programmé !
- les bugs peuvent être "non déterministes" ("Heisenbug"...)

Les bugs dans un code

Mieux vaut prévenir que guérir :

- adopter de bonnes pratiques de programmation
- tester régulièrement son code
- détecter les problèmes le plus tôt possible dans le processus de programmation

Mais quand il faut guérir :

- utilisation d'outils de débogage

Bonnes pratiques

Servent à éviter la confusion, et améliorer la compréhension entre les différents programmeurs. Donc en conséquence, à limiter le risque d'erreurs.

- commenter le code
- avoir indentation correcte
- utiliser des noms de variables/fonction explicites
- "garder le code simple" (KIS) :
- préférer des fonctions courtes
- éviter la redondance de code
- lors de la première version préférez un algorithme simple (et plus lent) à un algorithme complexe (et plus rapide)

Bonnes pratiques

Pour les projets conséquents, ou à plusieurs :

- code "modulaire"
- documentez vos fonctions
- respectez une convention de nommage
- utilisez un utilitaire de versionnage

Note : on peut faire un code correct sans ces pratiques, mais c'est périlleux (e.g : `ioccc`)

Pratiques mauvaises/dangereuses/interdites

Ne pas initialiser les variables

- l'erreur pourra passer inaperçue, car souvent elle sera initialisée à 0 la première fois, mais après ce sera plus aléatoire...

Ne pas tester les codes retours

- lire les manuels des fonctions que vous utilisez

L'utilisation de fonctions réputées dangereuses

- sprintf(), strcpy(), strcat(), vsprintf(), gets() ne vérifient pas si il y a assez de place
- fonctions non ré-entrantes dans un code multithread

Ne pas désallouer/fermer ce qui ne sert plus (mémoire, descripteurs de fichiers...)

Note : avec ces pratiques, un code ne sera pas "correct".

Tester

En cas de projet conséquent, il faut régulièrement :

- tester si le code compile
- tester s'il donne les résultats attendus

Séparer le processus de développement en petites parties. Par exemple, on peut dégager deux processus indépendants :

- le "refactoring" : on ne change pas les fonctionnalités, on ne fait que réorganiser/clarifier/simplifier/optimiser le code.
- l'ajout de fonctionnalités.

Ne pas les faire en même temps, et vérifier après chaque étape.

Tests

- "Test unitaire" : vérifier le bon fonctionnement d'une partie (unité, module) du logiciel.
- Créez et intégrez une batterie de tests qui teste automatiquement chaque partie les unes après les autres
- Les tests doivent être "méchants" : testez sur beaucoup d'entrées, et essayez de couvrir tous les cas

Programmation par contrats

Assertion : expression qui doit être évaluée à vraie à un moment donné

Dans le paradigme "Programmation par contrats", 3 types d'assertions :

- pré-conditions
- post-conditions
- invariants

En C/C++ : on peut tester une assertion avec `assert`

assert

`assert(expr)`

- dans assert.h
- se déactive avec l'option -DNDEBUG
- attention : pas pour tester les codes retours dans un vrai programme !

```
#define mon_assert(expr) {\
  if(!(expr)) {\
    fprintf(stderr,"assert␣%s␣fail␣%s:%s:%d\n",\
                __STRING(expr), __FILE__,\
                __ASSERT_FUNCTION,__LINE__);\
    abort();\
  }\
}
```

Outils d'aide au développement

Utilisation d'environnement de développement

Outils de gestion de version

- subversion (SVN), Git, mercurial...
- possible de faires des branches stable / développement
- il est possible de reprendre une ancienne version pour tracker l'apparition d'un bug.

Tracker les bugs : outils à disposition

voir les choses "suspectes", même sur un code qui semble marcher correctement :

- gcc -Wall
  - un code devrait toujours compiler sans warning !
  - on peut raffiner les tests de warning. ex : "-Wno-sign-compare"
  - on peut (des)activer un test dans le code :

```
#pragma GCC diagnostic ignored "-Wsign-compare"
...
#pragma GCC diagnostic warning "-Wsign-compare"
```

Tracker les bugs : outils à disposition

- gcc -fstack-protector-all
- en C++ : g++ -D_GLIBCXX_DEBUG pour des tests sur les conteneurs de la STL
- Valgrind : passer un coup de valgrind de temps en temps, même sur un code sans suspicion, ne fait pas me mal...

En cas de bug avéré :

- compiler avec les infos de débogage : gcc -g
  - attention, des fois cela fait des choses bizarres avec "-Ox"
- gdb
- valgrind

Valgrind

valgrind détecte (des fois) :

- les variables non initialisées
- les fuites mémoires
- les dépassements de tableaux

Il y a (rarement) des faux positifs (dans certaines librairies). Mais en général : si il y a un warning, c'est qu'un truc n'est pas bon dans votre code. Càd, un truc à corriger au plus tôt !

Principe (idée) : exécute le code dans un processeur virtuel. Exécution 10 à 30x plus lente...

Valgrind : utilisation

- Compiler avec l'option -g (rajout des symboles de débogage dans le fichier binaire)
- Exécuter la commande, précédée de `valgrind`
- Les avertissement seront envoyés sur la sortie erreur :

```
==21068== Invalid write of size 8
==21068==    at 0x400A6F: add(char const*, elm_t*) (vector.
    cpp:28)
==21068==    by 0x400AF7: main (vector.cpp:39)
==21068==  Address 0x5a81c88 is 8 bytes inside a block of
    size 16 free'd
==21068==    at 0x4C2A30B: operator delete(void*) (
    vg_replace_malloc.c:575)
...
```

Autres outils de la suite Valgrind

`valgrind -tool=<toolname>`

- memcheck (par défaut) : reporte les problèmes d'accès mémoire (non alloué, non initialisé, inaccessible), les fuites mémoires, double-free...
- massif : profilage de tas
- cachegrind, callgrind : profilage de cache
- helgrind , DRD : déboguer programmes multithreadés

gdb

gdb est le déboguer par défaut de la suite GNU
Permet, entre autres :

- d'exécuter jusqu'à un ou des points d'arrêts
- d'exécuter pas à pas
- de regarder l'état des variables, pile, registres...

Comment ça marche (idée, sur x86) :

- gdb a accès à tout l'espace mémoire du processus qu'il débogue
- quand on met un point d'arrêt sur une ligne, gdb remplace la première instruction machine correspondante à la ligne par une instruction "INT 3" (opcode : 0xCC)
- l'exécution de "INT 3" provoque une interruption, qui rend la main à gdb (qui peut remettre l'instruction initiale à la place de INT 3)

gdb : lancement

Compiler le programme à déboguer avec l'option "-g"

- attention, ça fait souvent des choses bizarres avec -Ox

Lancer le gdb :

- `gdb ./executable`
- si arguments : `gdb --args ./executable arguments...`

Dans l'interface de gdb :

- `run` : lance l'exécution

Attacher un programme en cours d'exécution :

- lancer `gdb`
- `attach pid`

- `gdb -tui` : avec une interface textuelle

gdb : lister le code

- `run` : lance l'exécution
- `ctrl+c` : stoppe l'exécution
- `cont` : continue l'exécution
- `list` : lister le code (à la position courante)
- `list` *`fct`* : lister le code depuis le début de la fonction *fct*
- `list` *`fichier:fct`* : lister le code depuis le début de la fonction fct dans le fichier *fichier*
- `list +`, `list -` : avancer (reculer) dans le fichier
- `step` : avance d'un pas
- `next` : avance d'un pas (sans entrer dans les fonctions)
- `finish` : avance jusqu'à la fin de la fonction courante

gdb : points d'arrêts

Point d'arrêt (breakpoint) : arrête le processus quand il atteint une ligne

- `break` *`fct`* : rajoute un point d'arrêt au début de la fonction *fct*
- `break` *`n`* : rajoute un point d'arrêt à la ligne *n*
- `break` *`fichier:ligne`* ou `break fichier:fct`
- possibilités de point d'arrêts conditionnels
- `info breakpoints` : lister les points d'arrêts

Retirer un point d'arrêt :

- `clear` *`fct`*
- `delete` *`nb`*

gdb : variables et "watchpoints"

- `print` *var* : affiche la valeur d'une variable (ou expression)
- `display` *var* : affiche à chaque pas

Modifier une variable :

- `set` *var* `=` *x*

watchpoint : arrête le programme quand une variable est modifiée

- `watch` *var*

gdb : pile et threads

- `backtrace` : affiche la pile d'appels
- `up` / `down` : monter ou descendre dans les *frames*
- `frame` *num* : changer de *frame*

Multithread :

- `info threads` : liste les threads
- `thread` *num* : change le thread courant

gdb : registres et assembleur

- `info registers` : affiche les registres
- `layout asm` : affiche le code assembleur
- `layout src` : affiche le code source

Il existe des interfaces graphiques à gdb...

gdb : raccourcis

- entrée : précédente commande
- r : run
- l : list
- c : continue
- s : step
- n : next
- bt : backtrace
- i : info
- b : breakpoints
- i b : info breakpoints
- ...

Déboguer : aller plus loin

Il est possible d'intégrer des outils de débogage dans son code.

Exemple : `backtrace`

```
void sigsegv(int)
{
  void *bt[DEBUGMEM_MAXBT];
  int sizebt = backtrace (bt,DEBUGMEM_MAXBT);
  char **strings = backtrace_symbols (bt, sizebt);
  for(int i=0;i<sizebt;i++)
    fprintf(stderr,"  %s\n",strings[i]);
  exit(1);
}

...
signal(SIGSEGV, (sighandler_t) sigsegv);
signal(SIGBUS, (sighandler_t) sigsegv);
...
```

Optimiser son code

Là, on suppose que notre code marche bien. On veut l'optimiser :

Options de gcc :

- -Ox
  - -O0 : pas d'optimisation
  - -O1 : optimisations modérées
  - -O2 : pleines optimisations
  - -O3 : comme -O2, en encore plus agressif
  - -Os : optimisation en mémoire (taille de d'exécutable)
- -march=native : compile pour le processeur de la machine
- -ffastmath : active certaines optimisations sur les flottants (ne respecte plus la norme IEEE 754)
- ...

Optimisations de gcc : passer des variables en registres, rendre des fonctions *inline*, dérécursiver, déboucler, réorganisation des instructions...

À savoir :

- les malloc/free (new/delete), c'est plutôt lent. Préférer d'autres méthodes d'allocations en cas de grosse demande
- les realloc peuvent être très lents (déplacement en mémoire)
- les appels systèmes, c'est très lent

En C++ : Certains conteneurs sont plus lents que d'autres :

- utiliser le conteneur le plus adapté
- array, c'est bcp plus rapide que vector
- remplir un vecteur avec un push_back, c'est lent

Certaines choses rendent l'inlinisation impossible :

- les accesseurs séparés dans un autre fichier source
- les fonctions membres `virtual`...

Outils de profilage

Profilage : analyse dynamique de l'exécution d'un code.

Outils :

- gprof
- gcov
- C++ : g++ -D_GLIBCXX_PROFILE

`https://gcc.gnu.org/onlinedocs/libstdc++/manual/profile_mode.html`

gprof

- Calcule le temps passé dans chaque fonction, et le graphe d'appel.
- Le compilateur rajoute du code, qui va générer un fichier `gmon.out` contenant les informations de profilage.
- Inconvénient : le code ne doit pas être optimisé (-Ox) , sinon cela peut faire des choses bizarres
  - cela ne dit pas vraiment le temps passé dans chaque fonction quand ce sera optimisé, mais cela donne néanmoins de bonnes approximations
- Note : le code devient notablement plus lent

`gprof` : utilisation

- Compiler avec l'option -pg
  - attention, souvent cela fait des choses bizarres avec -Ox !
- Lancer le programme normalement. Il va générer le fichier `gmon.out`
- Une fois terminé, lancer `gprof executable`
- L'affichage est en 2 parties
  - Le temps passé dans chaque fonction
  - le graphe d'appel

gcov

- Teste la "couverture". Pour chaque ligne, affiche le nombre de fois que la ligne a été exécutée
- Compiler avec `-fprofile-arcs -ftest-coverage`
- Exécuter le code.
- Exécuter `gcov` *`fichier_source`*
- Il va générer un fichier texte *fichier_source.gcov*