

GCC

```
gcc -Wall -Wextra -g hello.c -o hello
```

-Wall : Ajout de warnings de base

-Wextra Ajout de warnings supplémentaires

-g : Ajout des infos de debug

-o : nom de l'exécutable final.

Compilation

La compilation passe par 4 étapes majeures

Pre-Processing

- suppression des commentaires
- Inclusion des fichiers .h dans le fichier .c
- Traitement des directives qui commencent par #

Compiling

- Vérification de la syntaxe
- Compilation en langage assembleur

Assembling

Transformation de l'assembleur en code machine.

Linking

Lier tous les fichiers entre eux :

- les différentes classes
- la lib C...

Arguments d'un programme

argc : nombre d'arguments qui sont passés.

argv : tableau de chaines de char (char** ou char *[])

argv a une taille de argc + 1 car il possède NULL en dernière position.

Processus

Un processus est un programme en cours d'exécution.

Il possède :

- un identifiant ==> **PID**
- Des I/O
- Un parent et un ou plusieurs enfants.

Fork

Clonage d'un processus.

Il possède les mêmes I/O mais pas le même PID.

La fonction **fork** prend 1 paramètre et renvoie un **pid_t**.

L'exécution du programme qui à été fork ne démarre pas depuis le début.

Lors du fork, le père récupère le PID du fils, le fils reçoit 0 et le père reçoit -1 si le fork n'a pas pu cloner.

Processus Zombie

Un processus qui c'est terminé normalement ou anormalement dont on a pas encore regardé l'état de retour.

`wait` peut attendre la fin d'un des fils, le premier qui se termine. Renvoie son identité et son statut de retour.

processus orphelin

Un processus ne peut pas être orphelin, si le père meurt, il change automatiquement de père.

Sleep

La méthode **sleep** permet d'éteindre un programme pendant un temps **minimum** pas le temps exact.

Makefile

1 seul fichier

```
CFLAGS=-Wall -Wextra -g
```

La commande make suivie du nom du programme sans le .c .

Réalise la commande cc -Wall -Wextra -g nomdufichier.c -o nomdufichier .

règles semi-explicites

On met les CFLAGS comme au dessus puis on définit ce qu'on veut compiler.

```
nomdufichier :nomdufichier.c
```

plusieurs fichiers

```
CFLAGS=-Wall -Wextra -g
```

```
hello: hello.o fonction.o
fonction.o : fonction.c fonction.h
hello.o: hello.c fonction.h
```

```
clean:
```

```
rm -rf *.o hello #lors de compilation complexes, ne pas supprimer les .o
```

Lorsqu'on utilise la commande **make**, la première règle est lancée (ici **hello**), si on veut clean, il faut faire la commande **make clean** .

Attention

Les warnings sont générés au moment de la compilation du fichier, si un fichier n'est pas modifié donc pas recompilé, les warnings ne seront pas re-générés.

Toujours tout recompiler pour vérifier les warnings en fin de développement.

make all

on peut créer une règle all (peu importe le nom) qui crée les différents exécutables qui sont dans le make.

```
all: hello fork
```

règles explicites

```
$@ #cible
$< #le premier fichier indiqué en dépendance
$^ #l'ensemble des dépendances
```

dans gcc, le -c permet de générer les fichiers objet (.o)

Exec

Lorsqu'on lance une commande dans un terminal, deux commandes se lancent :

- fork pour se dupliquer
- exec pour exécuter le bon programme.

famille de fonction

exec existe en 6 fonctions, regroupés en 2 familles :

- execl
- execv

La fonction **exec** quelle qu'elle soit, retourne seulement si il y a eu une erreur

execl

Possède une liste d'arguments terminée par NULL (même si il n'y a qu'un seul argument aka le nom du programme).

```
int error = execlp("ls", "ls", "-l", NULL);
```

Pipe

Fichiers

Tout processus lance au moins 3 fichiers :

- Entrée standard : 0
- Sortie standard : 1
- Sortie d'Erreur : 2

Lors d'un fork, le même fichier est ouvert avec le même état noyau (ils utilisent le même espace mémoire pour y accéder). Ils ne peuvent pas accéder au fichier au même moment.

ouverture de fichier

`open(avec des paramètres) => renvoie le file descriptor associé`

`close(avec des paramètres)`

Des flags qui peuvent être utile :

- O_RDONLY
- O_WRONLY
- O_RDWR
- O_CREATE => Crée le fichier si il n'existe pas
- O_APPEND => Amène à la fin du fichier
- O_TRUNC => efface le contenu du fichier avant de l'ouvrir

Pour utiliser plusieurs flags, on peut les concaténer avec un ou binaire

`O_WRONLY | O_CREAT`

read / write

read et write prennent 3 paramètres :

- int fd => le descripteur de fichier (0,1,2 pour les fichiers natifs et plus si un fichier a été ouvert)

- un buffer, de type void qui doit être un pointeur peu importe son type
- et un count qui correspond au nombre d'octets à lire (la fonction renvoie la quantité lue ou écrite)

Tubes anonymes (Pipe)

la fonction pipe prend en entrée un tableau de 2 int (int pipefd[2])

Ce tableau contient les descripteurs de fichiers de la sortie et de l'entrée du tube.

Le tube fonctionne sur un système **FIFO**.

Il faut fermer les extrémités que l'on utilise pas.

Dans un tube, l'extrémité entrante est tube[1] et l'extrémité sortante est tube[0]

Lors d'un read dans tube[0] , la données à été utilisée et disparait du tube.

redirection

```
int dup2(int oldfd, int newfd);
```

On duplique le descripteur de fichier oldfd dans newfd .

Si on fait

```
int dup2(tube[1], 1);
```

La sortie standard est remplacée par l'entrée du tube (on a plus la possibilité d'écrire sur la sortie standard).

tube nommé

```
int mknod(const char *filename, mode_t mode);
```

Crée un fichier de type tube sur le système.

Deux processus indépendants peuvent ainsi communiquer.

Pour supprimer le tube :

```
int unlink(const char *filename);
```

Semaphore

Introduit par Dijkstra en 1965. Utilisé pour synchroniser des processus.

Des Semaphores peuvent venir de processus différents.

Un **Semaphore** c'est :

- Un entier **E(s)** (positif) --> quantité d'opérations disponibles.
- Une file d'attente **F(s)** --> Liste d'attente des opérations qui ont fait P(s) quand E vault 0.
- Deux opérations :
 - P(s) --> décrémente E si possible (sinon va dans **F(s)**).
 - V(s) --> réincrémente F (pas besoin d'être passé par P(s) pour réincrémenter le compteur).
- P et V sont des opérations **atomiques** (opération qui ne peut être interrompue)

Mutex

Mutual Exclusion :

- Protéger une ressource critique (un seul process doit pouvoir y accéder pendant autant de temps que nécessaire)

Initialisation d'un sémaphore avec **un** jeton lors de son initialisation.

Chaque P doit être suivi par un V et inversement.

Utilisation

Gestion d'un parking avec barrière

Gestion d'accès à une BDD

...

Cas du parking :

On crée un semaphore avec n places disponibles, chaque entrée réalise une opération P et chaque sortie une opération V.

Cas bdd

Un réacteur doit être le seul à accéder à la base

Un lecteur peut y accéder en même temps de d'autres lecteurs (pas écriveurs)

Exemple des philosophes

Il y a n philosophes et n fourchettes, chaque philosophe à besoin de 2 fourchettes pour manger.

En C

```
int semget(key_t key, int nsems, int semflg);
```

key_t est l'identifiant (int)

nsems doit toujours valoir 1 (dit le nombre de semaphore créé)

semflg correspond aux droits.

Mémoire partagée

Permet de partager une donnée entre plusieurs processus.

Il faut protéger la lecture / écriture avec des sémaphores.

Il faut attacher la mémoire partagée avec shmat .

Lorsqu'on en a plus besoin, il faut la détacher avec shmdt .

shmat revoie un pointeur générique void * pour avoir la possibilité de retourner tout type de pointeur.

Thread

Une nouvelle file d'exécution au sein du même processus.

Mémoire partagée

Changement de contexte plus rapide (pas de déchargement de mémoire dans le processeur)

```
pthread_create(pthread_t *restrict thread ...)
```

prend en paramètre :

- un pthread_t
- une fonction void*
- et un argument void *

Pour attendre la fin d'un thread, il faut utiliser `pthread_join` avec le `pthread_t` et le retour.

Exit ferme de process, si on veut quitter un thread, il faut `pthread_exit`.

Semaphore

Pour créer : `sem_create`

`P` --> `sem_wait`

`V` --> `sem_post`

pour détruire : `sem_destroy`

mutex dans les pthread

`pthread_mutex_create`

`pthread_mutex_lock(pthread_mutex_t)`

`pthread_mutex_unlock(pthread_mutex_t)`

`pthread_mutex_destroy`

barrière

bloque les threads jusqu'à ce que les n se finissent.

Signal

kill

La fonction `kill` envoie un signal à un process.

On doit donner le pid. Si il est supérieur à 0, on envoie au bon process, si il vaut 0, envoie au process courant et à tous es enfants.

alarm

envoie un signal **SIGALRM** au pour de n secondes.

Si `alarm` est appelé 2 fois, la deuxième reset le premier et renvoie le temps qu'il restait avant l'envoi du signal.