wait & exit :

- Permet d'envoyer un int
- il faut ajouter #include <stdlib.h>
- le premier processus (qui va envoyer l'entier) doit appeler la fonction exit :
    - exit(int valeur_à_envoyer)
- le deuxieme processus (qui va lire l'entier) doit appeler la fonction wait (ce processus restera bloquer jusqu'à que le premier processus exécute exit) :
    - wait(int* adresse_de_la_variable)
- la valeur lu par le wait doit être divisé par 256 si on veut la valeur envoyé par exit

Pipe :

- Permet d'envoyer n'importe quel type de variable
- Etapes :
    - Créer un tableau tab de deux cases.
    - Appeler la fonction pipe et lui donné le tableau : pipe (tab)
    - Maintenant le tableau tab représente un tube qui relie les deux processus
        - La première case du tableau est utilisée pour la lecture. t[0]
        - La deuxième case du tableau est utilisée pour l'écriture. t[1]
    - write(tab[1],adresse_variable,sizeof(variable)) pour mettre la valeur de la variable dans l'entrée de la pipe
    - read(tab[0],adresse_variable,sizeof(variable)) pour prendre la valeur de la variable qui a été entrée dans la pipe
    - Remarques :
        - La synchronisation est faite automatiquement : le rédacteur écrira toujours avant que le lecteur lit.
        - Un tableau ne peut être utiliser que pour communiquer entre deux processus
        - Le processus qui écrit ne peut pas lire des informations, et inversement. Il faut donc créer deux tubes si on veut que les processus établissent réellement un dialogue.
        - Lorsque nous utilisons un tube pour faire communiquer deux processus, avant de lire il est important de fermer la sortie du tube (la deuxième case du tableau) et avant d'écrire il faut faire pareil pour l'entrée du tube.

A travers les fichiers :

- il faut ajouter #include <fcntl.h> (FILE CONTROL pour ne pas oublier , contient les constantes O_)
- ouverture du fichier avec open (retourn un entier) :
    - int n= open(char* "nom_ou_chemin" , int flags, mode_t mode) ;
        - ou
    - int n =open(char* "nom_ou_chemin",int flags) ;
        - flags :
            - O_CREAT : créer s'il n'existe pas
            - O_TRUNC : ecraser son contenu

- O_RDONLY : ouvrir en mode lecture seule
- O_WRONLY : ouvrir en mode écriture seule
- O_RDWR : ouvrir en mode lecture/écriture
- O_APPEND : écrire à la fin de fichier (le mettre à jour)
- Plusieurs flag peuvent être combiner avec le caractère pipe '|' :

▪ t_mode :

- 00ugo : les deux premiers 0 ne changent , pas u et g et o sont des chiffres qui represente respecitevement les droit de l'utilisateur , groupe et autres (le 1er 0 est obligatoire , il veut dire que le nombre est en octal)
- les valeurs que ces chiffres peuvent prendre :
  - 0 : aucun droit
  - 1 : droit d'execution
  - 2 : droit d'ecriture
  - 4 : droit de lecture

- écriture dans le fichier :
  - write(int le_fichier_ouvert_avec_open,const void* buffer,size_t nbre_de_car_à_ecrire)
- lecture depuis le fichier :
  - read(int le_fichier_ouvert_avec_open,void* buffer ,size_t nbre_de_car_à_lire)

Fork :

- très bien expliqué et résumé ici

Threads :

- pour créer un thread : pthread_create(pthread_t *nom_thread, pthread_attr_t * attr, void * (* start_routine) (void *), void * arg) : pthread
  - premier argument : le nom du thread qui doit être déjà declarer (de type pthread_t)
  - pthread_attr_t : le mode de lancement du thread (généralement on garde le mode par défaut en donnant NULL)
  - start_routine : cette fonction doit accepter un seul paramètre au maximum (donc 0 ou 1), la fonction que le thread va exécuter , il est préférable que le type de retour et de l'argument de cette fonction soit de type void*.
  - arg : l'argument à envoyer à la fonction s'elle a un parametre on lui envoie (void*) variable , qui doit être de préference un pointeur pour que le cast soit clair .
- pthread_join(pthread_t nom, void** variable)
- void pthread_exit (void * retval)
- pour exécuter un code C contenant les fonctions de la bibliothéque pthreadd il faut ajouté -lpthread à la fin de la commande gcc -o executable code.c -lpthread