gedit: permet de créer, ouvrir et éditer des fichiers de texte de manière interactive. Si le fichier spécifié n'existe pas , gedit créera un nouveau

compilation : Gcc -o hello hello.c (le -o sert à nommer l'exécutable)
exécution : ./hello
Gcc -c Hello.c (il va seulement compiler le fichier et créer un fichier hello.o qui n'est pas exécutable)
Gcc hello.o (c'est une commande qui crée un exécutable nommé par défaut A.out)

argc c'est le nombre d'éléments passés en paramètres (le nom du fichier c'est le premier élément de la table cad argv[0])

Argv[] c'est un tableau de pointeurs qui pointent vers les paramètres donnés dans l'instruction d'exécution du fichier avec le nom du fichier comme premier élément du tableau

---

access() c'est une fonction du langage C déclarée dans <unistd.h>
sert a vérifier les droits d'accès à un fichier ou un répertoire
`int access(const char *path, int mode);`
`const char *path` : le nom ou le chemin du fichier ou le répertoire qu'on veut vérifier ses droits d'accès
`int mode` : c'est un ensemble de drapeaux qui spécifient les opérations à vérifier
la fonction renvoie 0 si toutes les opérations spécifiées par le mode sont autorisées et -1 en cas d'erreur
si une opération n'est pas autorisée , la variable globale `errno` est généralement définie pour indiquer la nature de l'erreur

le rôle des drapeaux suivants de l'appel système access()
F_OK : Il vérifie si le fichier existe. Si l'utilisateur a cette permission, le fichier est considéré comme accessible.
R_OK : vérifie si l'utilisateur a le droit de lire les données du fichier. Si cette permission est accordée, la fonction retourne 0, indiquant que le fichier est accessible en lecture.
W_OK : vérifie si l'utilisateur a le droit d'écrire (modifier ou supprimer) dans le fichier. Si cette permission est accordée, la fonction retourne 0, indiquant que le fichier est accessible en écriture.
X_OK : vérifie si l'utilisateur a le droit d'exécuter le fichier. C'est principalement utilisé pour les fichiers exécutables. Si cette permission est accordée, la fonction retourne 0, indiquant que le fichier est accessible en tant qu'exécutable.

3)programme qui verifie l'existance d'un fichier passe en arguments:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

int main(int argc , char *argv[]){
if (access(argv[1],F_OK)){
printf("le fichier %s existe\n",argv[1]);
}else
```

```
printf("le fichier n'existe pas\n");
return 0;
}
```

`open()` est utilisée pour ouvrir un fichier existant ou en créer un nouveau.
Elle est déclarée dans l'en-tête `<fcntl.h>` sous Unix

```
int open(const char *pathname, int flags, mode_t mode);
```

`pathname` : C'est le chemin du fichier qu'on veut ouvrir ou créer.
- `flags` : Ce sont des indicateurs (drapeaux) qui spécifient le mode d'ouverture du fichier. Ces drapeaux sont souvent combinés à l'aide de l'opérateur logique OR (`|`).
- `mode` : (optionnelle) Il est généralement utilisé lorsque on crée un nouveau fichier (`O_CREAT` est spécifié dans les drapeaux).Il définit les permissions du fichier créé. Les constantes symboliques telles que `S_IRUSR` (lecture pour le propriétaire), `S_IWUSR` (écriture pour le propriétaire), etc., sont utilisées pour définir ces permissions.

La fonction `open()` retourne un descripteur de fichier (un nombre entier) en cas de succès et -1 en cas d'échec, auquel cas la variable globale `errno` est généralement définie pour indiquer la nature de l'erreur.

1) le rôle des drapeaux suivants de l'appel système open()
O_RDONLY : Ouvre le fichier en mode lecture seule
O_RDWR : Ouvre le fichier en mode lecture/écriture. Ce drapeau permet de lire à partir du fichier et d'écrire dedans.
O_WRONLY : Ouvre le fichier en mode écriture seule.
O_APPEND : Si le fichier existe, ce drapeau permet d'écrire à la fin du fichier plutôt qu'au début. Utile pour ajouter des données à un fichier existant sans les écraser.
O_CREAT: S'il n'existe pas, ce drapeau crée le fichier .Si le fichier existe déjà , il est ouvert normalement.
O_TRUNC : Si le fichier existe, ce drapeau le tronque (le vide) à zéro octet lors de l'ouverture. Il est souvent utilisé avec `O_WRONLY` ou `O_RDWR` pour effacer le contenu existant du fichier lors de son ouverture.

Ces drapeaux peuvent être combinés en utilisant l'opérateur de bits OR (`|`) lorsque vous appelez la fonction `open()`. Par exemple, pour ouvrir un fichier en lecture/écriture et créer le fichier s'il n'existe pas, vous pourriez utiliser `O_RDWR | O_CREAT`.

programme qui lit les noms des fichiers entrés en paramètres
si le fichier existe il va afficher son contenu car par car
puis passe a l'autre fichier et fait le même traitement: (comme la commande cat)

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>// pour open();
#include <fcntl.h>//pour O_RDONLY

int main(int argc , char *argv[]){
   char buf; //pour enregistrer l'info lue
   int fd; //identifiant d'un fichier prend la val de sortie de open()
   if (argc==1){
```

```
        printf("manque d'arguments!\nUsage: %s[filename] ...\n", argv[0]);
    }else{
        for(int i=1;i<argc ;i++){
        fd=open(argv[i],O_RDONLY);
        if(fd==-1)
           printf("le fichier %s n'existe pas !\n",argv[1]);
        else{//ouverture reussie
            while(read(fd,&buf,1) !=0) write(1,&buf,1);
          // read lit un car du fichier retourne 0 si fin du fichier
         //write affichage du caractère sur la sortie standard
               printf("\n");
               close(fd);}}}
    return EXIT_SUCCESS;}
```

pour l'executer : `gcc printchar.c -o printchar`
`./printchar [filename]`
cat printchar.c //cela permet d'afficher le contenu du fichier printchar

un code qui lit le contenu de chaque fichier s'il existe et affiche le nombre de caractères pour chaque fichier

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
int main(int argc , char *argv[]){
char buf; //pour enregistrer l'info lue
int fd; //identifiant d'un fichier
int cpt; //compteur du nombre de caracte`res dans chaque fichier
if (argc==1){//si pas de fichier
printf("manque d'arguments!\nUsage: %s[filename] ...\n", argv[0]);
}else{

for(int i=1;i<argc ;i++){//une boucle qui parcours tous les fichiers donne´s
cpt=0;
fd=open(argv[i],O_RDONLY);//ouvrir le fichier nume´ro i
if(fd==-1)

printf("le fichier %s n'existe pas !\n",argv[1]);
else{
while(read(fd,&buf,1) !=0) cpt++; // parcourir le fichier caracte`re par
caracte`re
et les compter
printf("le fichier %s contient %d caracteres", argv[i],cpt);
printf("\n");
close(fd);}
}}
return EXIT_SUCCESS;}
```

exemple open()

```
#include <fcntl.h>
#include <stdio.h>
```

```c
#include <stdlib.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <unistd.h>
int main() {
   const char *nomFichier = "exemple.txt";
   int descripteurFichier;
   // Ouvrir le fichier en écriture seule, créer s'il n'existe pas
   descripteurFichier = open(nomFichier, O_WRONLY | O_CREAT | O_TRUNC,
S_IRUSR | S_IWUSR);
     if (descripteurFichier == -1) {
      perror("Erreur lors de l'ouverture du fichier");
       exit(EXIT_FAILURE);
   } else {
       printf("Fichier ouvert avec succès (descripteur de fichier : %d).\n",
descripteurFichier);

       // Vous pouvez effectuer des opérations de lecture/écriture sur le
fichier ici

       // Fermer le fichier lorsqu'il n'est plus nécessaire
       close(descripteurFichier);
   }
   return 0;
}
/*Dans cet exemple,
le programme ouvre un fichier appelé "exemple.txt" en écriture seule.
S'il n'existe pas, il est créé, et le fichier est tronqué s'il existe déjà.
Les permissions du fichier sont définies pour permettre la lecture et
l'écriture par le propriétaire du fichier.
Après avoir effectué les opérations nécessaires, le fichier est fermé à
l'aide de la fonction `close()`.*/
```

exemple access()

```c
#include <stdio.h>
#include <unistd.h>
//a tester au calme juste pour savoir plus de access()
int main() {
   const char *fichier = "exemple.txt";
   // Vérifier si le fichier existe
   if (access(fichier, F_OK) == 0) {
       printf("Le fichier existe.\n");
   } else {
       perror("Erreur lors de l'accès au fichier");
   }
   // Vérifier si le fichier est lisible
   if (access(fichier, R_OK) == 0) {
       printf("Le fichier est lisible.\n");
   } else {
       perror("Erreur lors de la lecture du fichier");
   }
   // Vérifier si le fichier est inscriptible
   if (access(fichier, W_OK) == 0) {
```

```
        printf("Le fichier est inscriptible.\n");
    } else {
        perror("Erreur lors de l'écriture dans le fichier");
    }
    // Vérifier si le fichier est exécutable
    if (access("programme", X_OK) == 0) {
        printf("Le programme est exécutable.\n");
    } else {
        perror("Erreur lors de l'exécution du programme");
    }
    return 0;
}
//Dans cet exemple, le programme vérifie l'existence, la lecture, l'écriture et l'exécution d'un fichier. La fonction perror() est utilisée pour afficher un message d'erreur détaillé en cas de problème.
```

---

programme qui prend un fichier en parametre et le creer s'il n'existe pas sinon il envoi un message d'erreur

```
#include <stdio.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <stdlib.h>
int main (int argc, char *argv[]){ int fd;
if(argc !=2){
 //on doit passer juste un seul et unique parame`tre pour pouvoir continuer l'exe´cution de programme sinon y'a un exit(1).

printf("usage %s <file name#>\n", argv[0]);
exit(1); }

mode_t mode =S_IRUSR | S_IWUSR | S_IRGRP | S_IWGRP | S_IROTH | S_IWOTH ;

//mode_t est un type pre´de´fini dans la bibliothe`que <sys/types.h> permettant de contenir les droits d'acce`s affecte´s a` la variable de ce type la
//readuser|writeuser|readgroup| write group |read others |write others.
//affecter a` "mode" les droits d'acce`s pour l'utilisateur et le groupe et other

fd= open(argv[1], O_WRONLY | O_EXCL | O_CREAT ,mode );
 //ouvrir le fichier passe´ en argument en mode e´criture seule(O_WRONLY)
//cre´er le fichier s'il n'existe pas (O_CREAT) avec les droits d'acce`s donne´es en "mode"
 //sinon retourner -1 avec O_EXCL

if (fd==-1){ perror("open");
//donner l'erreur exacte dans le cas de l'e´chec d'ouverture de fichier
exit(-1); }
return EXIT_SUCCESS; }
```

```
//me^me si on a donne´ les propres droits d'acce`s pour le nouveau fichier
cre´é il va pas le prendre en compte
 //car le umask est plus prioritaire par rapport au programme exe´cute
```

perror("open")donne le type d'erreur d'ouverture
par ex: « open:file exists»

umask la valeur du masque par défaut des droits d'accès : `$ umask`
0022 == 000 010 010 (0 si le droit n'est pas masqué,

1 si le droit est masqué)
le système d'exploitation utilise le masque(0022) pour empêcher l'affectation de certains droits d'accès à un fichier meme si on declare nos proporesdroits d'acces dans le programme

il va faire le calcul: **mode AND !umask**

Mode : rw- rw- rw- (666)
Mode : 110 110 110 (002)
Umask : 000 010 010 (022)
!Umask : 111 101 101 (755)
Mode AND !Umask :110 100 100 (644)

rw- r-- r- -

stat() permet d'obtenir des informations sur un fichier spécifié en paramètre telles que sa taille, ses permissions, son propriétaire, son groupe, et d'autres attributs stat() retourne 2 valeurs:

- (0 succès et -1 échec)
- les infos sur le fichier retourne dans une structure prédéfinie struct stat

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

struct stat {

st_dev (id de disque )
 st_mode (droit d'acce`s pour le fichier)

st_size (la taille de fichier )
 st_uid (identifiant de user )
 st_gid (identifiant de group)
 st_ino (identifiant de la structure stat « inode number »)
 st_nlink ( nombre de lien physique car sous linux on peut donner le me^me
noms au plusieurs fichiers)
 st_rdev (device id sous linux imprimante par exemple est un fichier )

st_blksize (la taille d'un bloc pour les fichiers spe´ciaux )
 st_blocks (nombredublocalloue´)
 st_atim (le temps de dernie`re ouverture )            st_mtim (le temps du
dernie`re modification )
 st_ctim (le temps de dernier changement de statuts ) }
```

un programme qui prend le chemin d'un fichier en tant qu'argument en ligne de commande, utilise l'appel système stat pour obtenir des informations sur ce fichier, puis affiche plusieurs détails comme: le nom du fichier, les droits d'accès, le type de fichier, les droits d'accès de l'utilisateur en mode symbolique, le propriétaire du fichier, et la taille du fichier.(simulation de la commande ls -l)

```
#include <stdio.h>

#include <stdlib.h>

#include <sys/types.h>

#include <sys/stat.h>

#include <fcntl.h>

#include <pwd.h>

#include <time.h>

int main (int argc, char *argv[]){

  struct stat StattusBuffer;

   if (stat(argv[1], &StattusBuffer)==-1){

       perror(argv[1]);

       exit(1);

   }

   printf("%c",StattusBuffer.st_mode & S_IRUSR ? 'r':'-');
printf("%c",StattusBuffer.st_mode & S_IWUSR ? 'w':'-');

   printf("%c ",StattusBuffer.st_mode & S_IXUSR ? 'x':'-');
printf("%c",StattusBuffer.st_mode & S_IRGRP ? 'r':'-');

printf("%c",StattusBuffer.st_mode & S_IWGRP ? 'w':'-');

printf("%c ",StattusBuffer.st_mode & S_IXGRP ? 'x':'-');
printf("%c",StattusBuffer.st_mode & S_IROTH ? 'r':'-');
printf("%c",StattusBuffer.st_mode & S_IWOTH ? 'w':'-');

   printf("%c ",StattusBuffer.st_mode & S_IXOTH ? 'x':'-');

   printf("%s  ",getpwuid(StattusBuffer.st_uid)->pw_name);

   printf("%s  ",getpwuid(StattusBuffer.st_gid)->gr_name);

   printf("%lld  ",StattusBuffer.st_size);

   printf("%s  ",ctime(&StattusBuffer.st_mtime));

   printf("%s  ",argv[1]);
```

```
return EXIT_SUCCESS;}
```

programme C qui copie le contenu d'un fichier source vers un fichier destination, en simulant la commande cp.

vcbhjk

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/stat.h>
int main (int argc , char **argv){
   char buf;
   int fd,fd1;
   struct stat statusBuffer;
   //verifier nbr arguments
   if (argc != 3){
       printf("erreur de nombre d'arguments , entrez deux fichier
seulement\n");
       return -1;
   }
   stat (argv[1], &statusBuffer);
   mode_t mode = statusBuffer.st_mode; //on sauvegarde kes modes de
permissions de l'executable
   //pour l'utiliser dans la creation du fichier
   //verifier si le fichier existe
   fd=open(argv[1], O_RDONLY); //ouvrir en lecture seule
   if (fd== -1){
       printf("le fichier %s n'existe pas \n" , argv[1]);
       return -1;
   }
   //ouvrir le fichier en ecriture seule sinon creer le fichier
   fd1=open(argv[2], O_WRONLY | O_CREAT ,mode);
   if(fd1==-1){
       printf("erreur d'ouverture ou de creation\n");
       close(fd);
       return -1;
   }
   //copier le contenu du fichier 1 dans le fichier 2
   while (read(fd,&buf ,1)!=0)
       write(fd1,&buf,1);
   printf("copie termine\n");
   return 0;
}
```

un programme C qui affiche le contenu d'un répertoire passé comme paramètre. Si aucun répertoire n'est spécifié, le programme affichera le contenu du répertoire courant.Utilisez les fonctions `opendir`, `readdir` et `closedir` . ayants les options :

− l'option -r qui permet d'afficher uniquement les répertoires, et

− l'option -f qui permet d'afficher uniquement les fichiers réguliers.

Utilisez la fonction `getopt` afin de récupérer les options de la ligne de commande.

```
#include <stdio.h>
```

```c
#include <sys/types.h>
#include <dirent.h>
#include <unistd.h>
int main( int argc , char *argv[]){
   DIR *rep;
   struct dirent *ent ;
   int opt=getopt(argc, argv, "rf");
   if(opt != -1) //s'il y a une option
   {
       if(argc>2)
          rep=opendir(argv[2]);
       else
          rep=opendir(".");
       while ((ent=readdir(rep))!=NULL)
        {
           if ((opt=='r') && (ent-> d_type == DT_DIR))
               printf("%s\n", ent->d_name);
           else
             if((opt=='f') && (ent->d_type== DT_REG))
                printf("%s\n", ent->d_name);
        }
   }
   else
   {
       //s'il y a pas d'option
       if (argc>1)
         rep=opendir(argv[1]);
       else
         rep=opendir("."); //"." represente le repertoire courant
   while ((ent=readdir(rep))!=NULL)
      printf("%s\n", ent->d_name);}
   closedir(rep);
   return 0;}
```

- on doit d'abord disposer des privilèges du superutilisateur (sudo) pour monter des systèmes de fichiers
- la commande mount : affiche les points de montage actuels sur le système, avec les systèmes de fichiers correspondants. On peut voir quels périphériques sont montés et à quels emplacements dans l'arborescence du système.
- avant de retirer la clé USB on souhaite la démonter:

`umount <point_montage>`(on trouve le point de montage en exécutant `mount`)
ex: `umount /dev/sdb1` (pour démonter une partition il faut utiliser le point de montage)

- Le nom du périphérique associé à la clé USB pourrait ressembler à "/dev/sdX", où "X" est une lettre qui représente la clé USB (par exemple, "/dev/sdb" ou "/dev/sdc").

remarque: Lorsqu'on retire la clé USB, le système n'a plus d'informations sur cette clé, et donc l'entrée correspondante n'apparaît plus dans la sortie de la commande.

- La commande mount permet aussi d'attacher un système de fichiers (comme une partition de disque ou un périphérique de stockage tq la clé USB) à un point spécifié

dans l'arborescence du système, rendant son contenu accessible à partir de cet emplacement:
`mount [-t type] [-o options] source destination`
type : spécifie le type de système de fichiers.
options : spécifie des options de montage telles que les permissions, les options de lecture/écriture…
source : le périphérique ou le point de montage source.
destination : le point de montage où on veut attacher le système de fichiers.
⇒ `mount -t vfat /dev/sdb1 /media/usb`

- Créez un fichier dans le répertoire /media/usb en redirigeant la sortie de la commande cal vers un fichier : `cal > /media/usb/fich`
- démonter la clé USB : `umount /media/usb`
  Après avoir démonté la clé USB, le contenu du répertoire /media/usb n'est plus accessible car le fichier qu'on a créé reste physiquement sur la clé USB. Le répertoire /media/usb sur le système de fichiers principal est vide après le démontage.
- il faut que la clé USB soit démontée avant de procéder au formatage:
  Formater la clé USB avec le système de fichiers ext2 à l'aide de la commande mkfs.ext2. : `mkfs.ext2 /dev/sdX`
  La commande `mkfs` (Make File System) est utilisée pour formater un périphérique de stockage en créant un système de fichiers sur celui-ci
  si on monte la cle USB sur ce point de montage avec le nouveau systeme de fichiers le contenu de la cle sera endomage car on a change le systeme de fichier (de FAT a ext2)
- `mount | grep /media/usb2`
  cette commande permet d'afficher les informations sur le point de montage /media/usb2 avec le type de système de fichiers utilisé
- le rôle de la commande `dumpe2fs` : afficher les informations détaillées sur le système de fichiers ext2, ext3 ou ext4 d'un périphérique. Elle fournit des informations telles que la taille du bloc, l'utilisation de l'espace disque, l'état du système de fichiers…

on peut Utiliser la commande dumpe2fs afin de confirmer que l'étiquette de la clé USB est bien mon nom : `dumpe2fs /dev/sdb1 | grep "Volume name"`
Cette commande affiche le nom du volume (étiquette) associé au système de fichiers ext3 de la clé USB.

- `dumpe2fs /dev/sdb1 | grep "Journal size"`

Cette commande affiche la taille du journal associé au système de fichiers ext3 de la clé USB.

- reformater la clé USB avec le système de fichier vfat : `mkfs.vfat -n KRIMAT /dev/sdb1`

Le message d'erreur indique que le périph est occupé , car le périph est toujours monté . on doit toujours démonter le périph avant de le reformater

```
umount /media/usb2
mkfs.vfat -n KRIMAT /dev/sdb1
```

- `fdisk -l` : affiche les informations sur les disques et partitions actuellement détectés sur le système telles que la taille, le type de système de fichiers, le nom du périphérique associé

Les options de commande fdisk :

`-Fdisk /dev/nom_periph` :pour interagir avec la table de partitions du periphérique nom_periph (contient des info sur la maniere dont le periph est divise en differentes partitions)

-fdisk -d :supprimer une partition

-fdisk - l : lister les types de partitions supportés

-fdisk –m :afficher le fichier d'aide

-fdisk –n :créer une nouvelle partition

-fdisk – p :afficher la table de partitions pour ce périphérique

-fdisk –q : quitter fdisk sans sauvegarder les changements

-fdisk –w : sauvegarder les changements et quitter fdisk

pour lancer le programme de partitionnement de la clé USB :`fdisk /dev/sdb1`

cela lancera l'interface de partitionnement fdisk pour le periph spécifiée permettant de créer , modifier , ou supprimer des partitions sur la clé usb

1) pour supprimer les partitions existantes sur la clé USB:
   - enter 'd' pour supprimer une partition
   - sélectionner la partition à supprimer
   - répéter ces étapes pour chaque partition existante
2) creer une partition primaire de taille 100 Mo , formatée en FAT32 :
   - enter n pour créer une nouvelle partition
   - sélectionner p pour une partition primaire
   - choisir le point de depart et la taille

```
Commande (m pour l'aide) : n
Type de partition
   p   primaire (0 primaire, 0 étendue, 4 libre)
   e   étendue (conteneur pour partitions logiques)
Sélectionnez (p par défaut) : p
Numéro de partition (1-4, 1 par défaut) : 1
Premier secteur (2048-7864255, 2048 par défaut) :
Dernier secteur, +secteurs ou +taille{K,M,G,T,P} (2048-7864255, 7864255 par défa
ut) : +100M

Une nouvelle partition 1 de type « Linux » et de taille 100 MiB a été créée.

Commande (m pour l'aide) :
```

   - entrer t pour changer le type de partition
   - sélectionner le numéro de la partition qu'on vient de créer
   - choisir le type de système de fichiers FAT32 (code hexadécimal 'b')

```
1b  W95 FAT32 masqu 70  DiskSecure Mult bb  Boot Wizard mas fd  RAID
1c  W95 FAT32 masqu 75  PC/IX           bc  Acronis FAT32 L fe  LANst
1e  W95 FAT16 masqu 80  Minix ancienne  be  Amorçage Solari ff  BBT
Type de partition (taper L pour afficher tous les types) : b

Type de partition « Linux » modifié en « W95 FAT32 ».
```

3) créer une partition étendue de taille égale à l'espace libre restant de la clé usb:
   - entrer n pour créer une nouvelle partition
   - sélectionner e pour une partition étendue

- choisir le point de départ de la taille en fonction de l'espace restant

```
Commande (m pour l'aide) : n
Type de partition
   p   primaire (1 primaire, 0 étendue, 3 libre)
   e   étendue (conteneur pour partitions logiques)
Sélectionnez (p par défaut) : e
Numéro de partition (2-4, 2 par défaut) :
Premier secteur (206848-7864255, 206848 par défaut) :
Dernier secteur, +secteurs ou +taille{K,M,G,T,P} (206848-7864255, 7864255 par dé
faut) :

Une nouvelle partition 2 de type « Extended » et de taille 3.7 GiB a été créée.

Commande (m pour l'aide) :
```

4) Créer une première partition logique de taille de 100Mo:
   - entrer n pour créer une nouvelle partition
   - sélectionner l pour une partition logique
   - choisir le point de départ de la taille (en fonction de l'espace disponible dans la partition étendue)
5) Créez une deuxième partition logique de taille égale à l'espace libre restant de votre clé USB:
   - entrer n pour créer une nouvelle partition
   - sélectionner l pour une partition logique
   - choisir le point de départ de la taille (en fonction de l'espace disponible dans la partition étendue)
6) Utilisez l'option w de la commande fdisk pour mettre à jour la table de partitions

```
Commande (m pour l'aide) : w
La table de partitions a été altérée.
Appel d'ioctl() pour relire la table de partitions.
Échec de relecture de la table de partitions.: Argument invalide

Le noyau continue à utiliser l'ancienne table. La nouvelle sera utilisée lors du
 prochain démarrage ou après avoir exécuté partprobe(8) ou kpartx(8).
```

7) Formatez la première partition logique de votre clé USB avec le système de fichiers ext2: `mkfs.ext2 /dev/sdXn` avec n = numéro de partition
   `ex: mkfs -t ext2 /dev/sdb1p5`
   les partitions primaires il y a 4 :sdb1p1 , sdb1p2 , sdb1p3 , sdb1p4
   les partitions logiques commencent par sdb1p5