Question 1

Le BIOS ("Basic Input/Output System") est un programme stocké en mémoire non volatile, sur la carte mère de l'ordinateur. C'est le premier programme à être exécuté au démarrage de l'ordinateur. Il effectue les tests de certains composants de la carte mère, tels que la mémoire et les périphériques de base tels que disques durs et autres périphériques de stockage, équipements réseaux, etc. Il charge en mémoire leurs "firmware", en prévision de leur utilisation éventuelle par le système d'exploitation.

Dans l'ordre qui lui a été fourni en paramètre, il cherche le premier périphérique de stockage présent et charge en mémoire le premier secteur de ce périphérique, à qui il transfère le contrôle de l'ordinateur, en exécutant le code qu'il contient, permettant ainsi le chargement du système d'exploitation.

Question 2

(2.1)

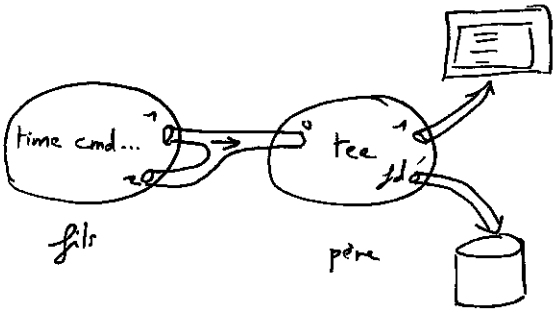
```
int
main (int argc, char * argv[])
{
  int   fd;
  char  c;
  fd = open (argv[1], O_WRONLY | O_CREAT | O_TRUNC, 0666);
  while (read (0 , &c, 1) > 0) {        /* Lit 1 caractère (au plus) */
    write (1, &c, 1);                   /* Copie sur la sortie standard */
    write (fd, &c, 1);                  /* Copie dans le fichier */
  }
  close (fd);
  return (EXIT_SUCCESS);
}
```

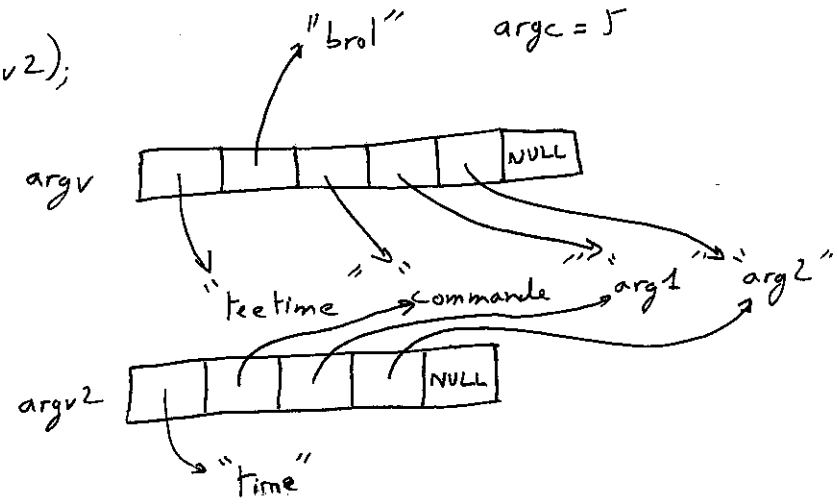
```
int
main (int argc, char * argv[])
{
    int   fd[2];

    pipe (fd);
    if (fork() >0) {   /* père → tee */
        close (fd[1]);
        dup2 (fd[0], 0);
        close (0);
        execlp ("/usr/bin/tee", "tee", argv[1], NULL);
    }
    else {                  /* Fils → time command ... */
        int    i;
        char ** argv2;

        close (fd[0]);
        dup2 (fd[1], 1);
        dup2 (fd[1], 2);
        close (fd[1]);

        argv2 = (char **) malloc (argc * sizeof (char *));
        argv2[0] = "time";
        for (i=1; i< (argc-1); i++)
            argv2[i] = argv[i+1];
        argv2[argc-1] = NULL;
        execvp ("/usr/bin/time", argv2);
    }
}
```





Question 3

(3.1) longjmp ne peut être appelé que dans un contexte de pile plus profond que celui du dernier appel à setjmp. À l'inverse, on ne peut appeler longjmp dans un contexte de pile moins profond que celui du dernier appel à setjmp. En effet, comme longjmp remet le pointeur de pile dans le même état qu'au moment où setjmp a été appelé, il faut que le contenu de la pile n'ait pas été détruit par un dépilage de contexte entre le setjmp et le longjmp.

De même, on ne peut faire un longjmp au sein d'une routine de traitement d'interruption que si la pile se trouvait, au moment de l'appel à la routine de traitement de l'interruption, dans un contexte plus profond que celui dans lequel setjmp a été appelé. Les raisons sont identiques.

(3.2)
```
jmp_buf     env;                /* Variable globale (peut être "static") */

int
essaie (int (*f) (void))
{
  int  e;
  if((e=setjmp (env))==0)       /* Si premier appel à setjmp et pas retour de longjmp*/
      f();
  return (e);
}

void
leve (int e)
{
  longjmp (env, e);
}
```

N.B.: Autres solutions possibles avec sigsetjmp / siglongjmp (ce qui sera mieux pour la question suivante).

(3.3)

```
void
alarme (int sig)
{
  leve (-1);
}

int
essaie_tempo (int (*f)(void), int duree)
{
  int e;
  signal (SIGALRM, alarme);
  alarm (duree);
  if ((e = setjmp (env)) == 0)
    f();
  return (e);
}
```

N.B.: Autres solutions possibles (préférables) avec sigsetjmp/siglongjmp, sigaction.

Question 4

(4.1)

```
void
parallèle (int nbre, void (*f) (void * info))
{
  pthread_t * thrdtab;
  Info *      infotab;
  int         i;

  thrdtab = (pthread_t *) malloc (nbre * sizeof (pthread_t));
  infotab = (Info *)      malloc (nbre * sizeof (Info));

  infotab[0].nbre = nbre;
  infotab[0].rang = 0;
  :
  :
```

```
    for (i=1; i < nbre; i++){
        infotab[i].nbre = nbre;
        infotab[i].rang = i;
        pthread_create (&thrdtab[i], NULL, (void * (*) ()) f, &infotab[i]);
    }
    f(&infotab[0]);

    for (i=1; i < nbre; i++)
        pthread_join (thrdtab[i], NULL);

    free (infotab);
    free (thrdtab);
}
```

(4.2)

```
void
calculer_histo2 (void * info)
{
    int   nbre = ((Info *) info) -> nbre;
    int   rang = ((Info *) info) -> rang;
    int   i, j;

    for (i = rang; i < hauteur; i += nbre) {
        for (j = 0; j < largeur; j++)
            histo[image[i][j]] ++;
    }
}
```

```
void
calcule_histo (int nbre)
{
    memset (histo, 0, 256 * sizeof (unsigned int));
    parallele (nbre, calcule_histo2);
}
```

(4.3) L'instruction d'incrémentation peut poser problème parce qu'elle n'est pas atomique. À cause des accès concurrents au tableau histo, partagé par tous les threads, certaines incrémentations peuvent ne pas être prises en compte. Les valeurs calculées du tableau histo risquent donc d'être inférieures aux valeurs attendues.

Pour corriger ce problème, il faut protéger l'incrémentation au sein d'une section critique, au moyen d'un mutex.

```
pthread_mutex_t    mtx;

void calcule_histo2 (void * info)
{
    int nbre = ((Info *) info) -> nbre;
    int rang = ((Info *) info) -> rang;
    for (i=rang; i < hauteur; i += nbre) {
        for (j=0; j < largeur; j++) {
            pthread_mutex_lock (&mtx);
            histo [image[i][j]] ++;
            pthread_mutex_unlock (&mtx);
        }
    }
}
```

```
void
calculer_histo (int nbre)
{
    memset (histo, 0, 256 * sizeof (unsigned int));
    pthread_mutex_init (&mtx, NULL);
    parallele (nbre, calculer_histo_2);
    pthread_mutex_destroy (&mtx);
}
```