université de BORDEAUX

Collège Sciences et technologies

Année	2020-2021	**Type**	Entrainement
Licence	Informatique		
Code UE	4TIN503U	**Épreuve**	Programmation Système
Date	13/11/2020	**Documents**	Non autorisés
Début	14h	**Durée**	1h

À la fin du sujet, vous trouverez un memento vous rappelant la syntaxe de quelques appels système utiles.

1 Questions de cours (échauffement)

Question 1 Que se passe-t-il lorsqu'un processus tente une écriture dans un tube depuis lequel plus aucun processus ne possède de descripteur de fichier ouvert en lecture ? Et dans la situation symétrique (lecture depuis un tube sur lequel plus aucun processus ne possède de descripteur ouvert en écriture) ?

Question 2 Expliquez précisément la différence entre `stdout` et `STDOUT_FILENO`.

2 Fichiers

On développe une application de gestion de stocks pour une petite boutique. La liste des articles existants est mémorisée dans un fichier d'inventaire, sous la forme d'une suite d'enregistrements de type `struct article` :

```
struct article {
  unsigned ref;       // référence
  char name[16];      // nom de l'article
  unsigned nb_items; // nombre d'exemplaires en stock
  off_t skip;         // réservé pour usage interne
};
```

Un fichier d'inventaire a donc une taille multiple de `sizeof(struct article)`. Au fil du temps, certains articles peuvent cesser d'être commercialiés, auquel cas ils sont effacés dans le fichier en positionnant le champ `ref` à zéro. Pour fixer les idées, voici comment on pourrait créer artificiellement un fichier d'inventaire contenant 4 articles référencés, et 3 emplacements non utilisés (seuls les champs `ref`, `name` et `nb_items` nous intéressent pour le moment).

```
{
  struct article t[] = {{1, "stylo", 100, 32},
                        {0, "", 0, 0},
                        {41, "gomme", 50, 64},
                        {0, "", 0, 0},
                        {0, "", 0, 0},
                        {25, "bloc note uni", 20, 0},
                        {26, "bloc note ligne", 20, 0}};
  ... // open
  write (fd, t, sizeof (t));
  ... // close
}
```

Une fonction `display` affiche les articles référencés dans un fichier d'inventaire. Voici son implémentation :

```
void display (char *filename)
{
  struct article a = {0, "", 0, 0};
  int fd           = open (filename, O_RDONLY);

  printf ("Inventaire :\n");
  while (read_next (fd, &a) != -1)
    printf ("  Reférence %2d, %-16s quantité en stock : %d\n", a.ref, a.name, a.nb_items);
  close (fd);
}
```

Son exécution sur le fichier d'exemple précédemment évoqué produit la sortie suivante :

```
Inventaire :
  Reférence  1, stylo            quantité en stock : 100
  Reférence 41, gomme            quantité en stock : 50
  Reférence 25, bloc note uni    quantité en stock : 20
  Reférence 26, bloc note ligne  quantité en stock : 20
```

Question 1 Donnez le code de la fonction `read_next (int fd, struct article *ptr)` qui range à l'adresse `ptr` les informations du prochain article valide dans le fichier (en renvoyant 0) ou qui renvoie -1 si la fin de fichier est atteinte sans trouver d'article valide. On suppose que la position courante du fichier `fd` est un multiple de `sizeof (struct article)` avant l'appel.

Question 2 On s'intéresse maintenant au champ `skip`, qui contient le nombre d'octets « à sauter » pour atteindre le prochain article valide dans le fichier. Dans notre exemple d'inventaire, remarquez que la valeur de ce champ vaut 32 (= `sizeof (struct article)`) pour un article suivi d'un emplacement libre, et 64 pour un article suivi de 2 emplacements libres consécutifs. Pour simplifier, on suppose que le premier article du fichier est toujours valide.

Donnez le nouveau code de la fonction `read_next` en utilisant ce champ atteindre plus rapidement le prochain article. Notez (en regardant le code de `display`) que le paramètre `ptr` pointe toujours sur une structure déjà initialisée.

Question 3 Donnez le code de la fonction `void add_new_ref (char *filename, int ref, char *name)` qui ajoute un nouvel article dans le fichier en utilisant le premier emplacement libre trouvé. Le champ `nb_items` sera initialisé à zéro. On ne demande pas de vérifier qu'un article de même référence existe déjà. Il faut juste prêter attention à bien mettre à jour les champs `skip` impactés par cet ajout.

3 Processus

On souhaite disposer d'une fonction `void run_at (unsigned secs, char *cmd, char *arg)` qui crée un processus chargé d'exécuter le programme `cmd` (avec un seul paramètre `arg`) au bout de `secs` secondes. Voici typiquement comment on voudrait pouvoir l'utiliser :

```
int main ()
{
  run_at (2, "ls", "-l");
  run_at (4, "echo", "coucou");
  sleep (10);
}
```

Dans ce cas, on devrait voir apparaître le résultat de `ls -l` sur la sortie standard au bout d'environ 2 secondes, puis le résultat de `echo coucou` 2 secondes plus tard.

Question 1 Donnez le code de la fonction `run_at`. Note : on ne se soucie pas de la récupération des zombies.

Question 2 Ajoutez un 4^e paramètre `int fd` à la fonction `run_at` et redirigez la sortie d'erreur de la commande vers ce descripteur, supposé ouvert et accessible en écriture.

Question 3 On suppose qu'un *tube* est créé au début de la fonction `main` et que son descripteur en écriture est passé en paramètre à tous les appels à `run_at`.

Remplacez `sleep (10)` par une boucle qui lit depuis le tube et qui ajoute tous les caractères lus à la fin d'un fichier `"run.log"`. La boucle de lecture ne doit pas être infinie...

Memento

```
int open(const char *pathname, int flags, mode_t mode);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);
off_t lseek(int fd, off_t offset, int whence);
/* whence = SEEK_SET ou SEEK_CUR ou SEEK_END */
```

```
int dup2(int oldfd, int newfd);
int execlp(const char *file, const char *arg0, ...);
int execvp(const char *file, char *const argv[]);
int pipe(int fildes[2]);
pid_t wait(int *stat_loc);
```