U. Paris Diderot Programmation système avancée le 25 juin 2019 8h30-11h30

Documents autorisés :

(1) trois feuilles manuscrites format A4 recto/verso de notes personnelles, chaque page doit porter votre nom,

(2) les deux polys du cours.

Le sujet comporte 4 pages.

Votre code doit être écrit de façon **lisible**, *avec des indentations et des accolades appropriées permettant de voir la fin des blocs de code (fin de boucles, etc.). Vous pouvez, si vous le jugez utile, écrire des fonctions auxiliaires.*

Il est inutile d'écrire les includes. Une gestion d'erreurs, même minimaliste, sera appréciée (mais ce n'est pas le point le plus important).

Exercice 1

Question 1: On exécute le programme suivant.

```
1  int main(int argv, char *argc[]){
2    char buff[1024];
3    char *txt="abcdefghijkmn";
4    char *t1="xxx";
5    char *t2="uuu";
6    int d;
7    d=open("toto", O_RDWR|O_TRUNC|O_CREAT|O_APPEND, S_IRUSR|S_IWUSR);
8    write(d,txt,strlen(txt));
9    lseek(d,-8,SEEK_CUR);
10   read(d,&buff,3);
11   write(STDOUT_FILENO,&buff,2);
12   write(d,t1,strlen(t1));
13   lseek(d,0,SEEK_SET);
14   write(d,t2,strlen(t2));
15   exit(0);
16 }
```

Qu'affiche-t-il sur le terminal ? Quel est le contenu du fichier `toto` **après** l'exécution du programme ?

On suppose qu'on supprime `O_APPEND` dans `open`, ligne 7. Quel sera le contenu du fichier `toto` après l'exécution de programme après cette modification ?

Question 2: On sait que quand on installe un handler pour un signal le handler est hérité par le processus fils à travers de `fork()`.

Mais qu'est-ce qui se passe quand un processus fait `exec` ?

(1) Les handlers que le processus possède sont maintenus.

(2) Tous les handlers du processus disparaissent (on revient au traitement de signaux par défaut).

Choisir la réponse et justifier.

Exercice 2 [signaux] On considère le programme `selmot` suivant :

```c
/*     selmot.c    */
int fdin = 0;
int fdout = 1;
int fdterminal ;

int main(int argc, char **argv){
  if(argc != 3){
    fprintf(stderr, "usage : %s mC mQ\n", basename(argv[0]));
    exit(1);
  }
  // ouvrir le terminal
  fdterminal = open("/dev/tty", O_RDWR);

  size_t len_messC = strlen(argv[1]);
  size_t len_messQ = strlen( argv[2] );
  char c;
  ssize_t len;
  while( 1 ){
    len = read( fdin, &c, 1) ;
    if( len == 0 )
      break;
    /* ecrite sur le terminal soit argv[1] soit argv[2] en fonction
     * de caractere lu sur les descripteur fdin */
    if( c == 'C' ){
      write(fdterminal, argv[1], len_messC );
    }else if( c == 'Q' ){
      write(fdterminal, argv[2], len_messQ );
    }

    if( write( fdout, &c, 1 ) == -1 )
      exit(1);
  }
  if( len == 0 ){
    fprintf(stderr, "fdin closed");
    exit(1);
  }else if(len == -1)
    perror( "\nproblem read");

  exit(0);
}
```

La commande `selmot` implémentée par ce programme prend deux paramètres `mC` et `mQ` et, dans une boucle, elle lit un caractère sur le descripteur 0, écrit le même caractère sur le descripteur 1 et écrit sur le terminal soit le mot `argv[1]` soit le mot `argv[2]` en fonction de caractère lu sur le descripteur 0.

Il faut noter que les descripteurs 0 et 1 ne sont pas forcement lié au terminal (à cause de redirections possibles), pour cette raison dans la ligne 12 on ouvre explicitement le terminal.

Modifier `selmot` pour ajouter le traitement de signaux :

- si le programme `selmot` reçoit le signal `SIGUSR1` alors à la prochaine lecture d'un caractère sur le descripteur 0 on inverse le caractère écrit sur le descripteur 1, si le caractère lu est 'C' alors on écrit 'Q', si le caractère lu est 'Q' on écrit 'C'.
  Cette inversion de caractère doit se produire une seule fois, ensuite le programme revient au comportement normal, c'est-à-dire il copie le caractère lu sur le descripteur 0 vers le descripteur 1.
  La prochaine inversion de caractère aura lieu à la prochaine réception de `SIGUSR1`.
  Notez que le handler n'écrit rien sur le descripteur 1, le handler doit juste positionner un indicateur pour indiquer la réception du signal. C'est dans la boucle `while` quand `selmot` écrit un caractère sur le descripteur 1 qu'il faut inverser le caractère si l'indicateur est positionné.
- à la réception de signal `SIGPIPE` le programme `selmot` doit écrire sur le terminal `"pas de lecteur"` et terminer. (Le signal `SIGPIPE` est envoyé quand un processus essaie d'écrire dans un tube qui n'a pas de lecteurs.)
- le signal `SIGUSR1` peut être reçu pendant l'exécution de `read` (ligne 19) ou `write` (ligne 30). Il faut faire en sorte que
  - soit l'appel système `read`, `write` interrompu par le signal reprenne automatiquement (en installant le handler de `SIGUSR1` de façon appropriée).
  - soit détecter que `read` ou `write` était interrompu par un signal et revenir à `read` ou `write` interrompu. (Remarque : quand `read` ou `write` sont interrompus par un signal ces deux appels système retournent -1 et `errno == EINTR`).

Exercice 3 [redirections] Écrire un programme `bingbang` qui crée deux processus fils, appelons les `fils1` et `fils2` respectivement.

Les deux fils exécuteront le même programme `selmot` donné dans l'exercice précédent, `fils1` doit exécuter

```
./selmot toto titi
```

tandis que `fils2` doit exécuter

```
./selmot tic tac
```

Quand `exec` échoue il faut terminer le processus qui a fait `exec`.

Les deux fils doivent être liés par deux tubes anonymes.

Le premier tube, appelons le `tube1to2`, sert à la communication de `fils1` vers `fils2` : quand `fils1` écrit sur le descripteur 1 il faut que le caractère écrit soit envoyé dans `tube1to2` et quand `fils2` lu sur le descipteur 0 il faut qu'il lise sur `tube1to2`.

Le deuxième tube, appelons le `tube2to1`, sert à la communication de `fils2` vers `fils1` : quand `fils2` écrit sur le descripteur 1 il faut que le caractère écrit soit envoyé dans `tube2to1` et quand `fils1` lu sur le descipteur 0 il faut qu'il lise sur `tube2to1`.

Vous devez fermer tous les descripteurs inutiles chez les deux fils.

Après avoir implémenté le comportement décrit ci-dessus le deux fils sont bloqués sur `read` (les deux tubes sont vides).

Pour démarrer la communication entre deux frères le processus père écrit dans le tube `tube1to2` le caractère `C` (il le fait une seule fois) et il termine.

Exercice 4 [signaux] Un programme exécute une fonction sensible `f()`.

Nous voulons empêcher que l'exécution de `f()` soit interrompue par le signal `SIGUSR1` (nous ne voulons pas que le handler du signal soit exécuté pendant l'exécution de `f()`).

Mais si le signal `SIGUSR1` est effectivement envoyé pendant l'exécution de `f()` il faut que le handler du signal soit exécuté un fois `f()` terminé. Autrement, il faut faire en sorte que l'exécution de handler soit juste retardée jusqu'à la fin de `f()`.

Écrire un fragment de code qui implémente ce comportement :

- bloquer `SIGUSR1` avant l'appel à `f()`,
- exécuter `f()`,
- débloquer le signal `SIGUSR1` après l'exécution de `f()` et si on détecte que `SIGUSR1` a été reçu quand il a été bloqué (signal pendant) alors reenvoyer `SIGUSR1` à soi-même.

Exercice 5 Des processus communiquent par la mémoire partagée, le père (producteur) écrit un entier dans la mémoire, le fils (consommateur) lit l'entier écrit par le père.

Le contenu de la mémoire partagée est décrite par la structure

```
struct memoire{
  sem_t libre;
  sem_t occupe;
  int info;
}
```

où les sémaphores anonymes `libre` et `occupe` indiquent si `info` contient ou non un entier.

Question 1: Écrire une fonction

```
struct memoire *init()
```

qui crée la mémoire partagée pour stocker la structure `struct memoire`. La fonction initialise les deux sémaphores (`libre` vaut 1 et `occupe` vaut 0 initialement). `init` retourne le pointeur vers la mémoire partagé.

Question 2: Écrire la fonction

```
int produire( struct memoire *mem, int value)
```

qui met dans la mémoire partagée la valeur `value` (seulement si mémoire n'est pas occupée). Faire en sorte qu'aucun autre processus ne puisse ni lire ni modifier la mémoire pendant ce temps. La mémoire devient occupée.

Question 3: Écrire la fonction

```
int obtenir( struct memoire *mem, int *pvalue)
```

qui met à l'adresse `pvalue` la valeur stockée dans la mémoire partagée. Faire en sorte qu'aucun autre processus ne puisse ni lire ni modifier la mémoire pendant ce temps.

La mémoire devient libre.
