Programmation système avancée
Examen du 5 mai 2021, 13h00 - 15h30

Documents autorisés et consignes

Tous les documents manuscrits ou imprimés sont autorisés, sauf les livres.
Le sujet comporte 5 pages. Tâchez d'écrire de façon lisible, avec des indentations et des accolades appropriées permettant de voir la fin des blocs de code (fin de boucles, etc.).
Il est inutile d'écrire les `#include`.

Exercice 1 :

On sait que les handlers (gestionnaires de signaux) installés par un processus parent sont préservés par le `fork()` (le processus enfant hérite des handlers).
Qu'est-ce qui se passe quand un processus fait un `exec` (fait appel à une fonction de la famille `exec`) ?
Les handlers que le processus possède sont-ils maintenus ? Sinon, tous les handlers du processus disparaissent-ils (le processus revient au traitement pas défaut pour tous les signaux) ?
Justifier la réponse.

Exercice 2 :

On exécute le programme suivant :

```
1 int main(){
2   execl("/bin/echo", "echo", "bonjour", (char *)0);
3   char *txt = "au-revoir\n";
4   write(1, txt, strlen(txt));
5 }
```

Le programme affichera-t-il

(1) `bonjour` ?

(2) `au-revoir` ?

(3) `bonjour` et `au-revoir` ?

(4) rien ?

(J'espère que tout le monde sait ce que fait la commande `echo` ?)

Exercice 3 :

Un fichier texte est composé de lignes, chacune étant une suite de caractères qui termine par le caractère `'\n'` de nouvelle ligne. Par exemple le fichier qui contient les caractères suivants (où `\n` désigne le caractère de la nouvelle ligne et non pas deux caractères `\` et `n`) :

`abc\ndef\nghi\n`

est composé de trois lignes :

(1) abc\n (2) def\n (3) ghi\n

Cependant parfois la dernière ligne est incomplète, il manque le caractère '\n' à la fin. C'est le cas du fichier qui contient :

aze\nrty\nuio

où les lignes aze\n, rty\n terminent correctement avec '\n', mais le caractère de nouvelle ligne manque à la fin de la dernière ligne uio.
Écrire une commande (fonction main) qui prend comme argument un nom de fichier et ajoute un caractère de nouvelle ligne à la fin du fichier si (et seulement si) le dernier caractère du fichier est différent de '\n'.
Votre programme doit utiliser les fonctions de bas niveau : open, read, write, lseek.

Exercice 4 :

On exécute le programme suivant.

```
int main(int argc, char *argv[]){
  int f,g;
  char msga[]="abc";
  char msgb[]="edf";
  if( ( f = open("toto", O_RDWR|O_TRUNC|O_CREAT, S_IRUSR|S_IWUSR) ) == -1 ){
    perror("open");
    exit(EXIT_FAILURE);
  }
  if( ( g = open("gogo", O_RDWR|O_TRUNC|O_CREAT, S_IRUSR|S_IWUSR) ) == -1 ){
    perror("open"); exit(EXIT_FAILURE);
  }

  dup2(f,g);
  write(f, msga, strlen(msga));
  write(g, msgb, strlen(msgb));
  close(f);
  close(g);

}
```

Après l'exécution, le fichier toto contient-il

(1) abc ? (2) edf ? (3) abcedf ? (4) est-il vide ?

Même question pour le fichier gogo.

Exercice 5 :

Dans cet exercice il faut compléter le code de la fonction suivante :

```
int redirection( int desc, char  *arguments[], int fin ){
  int tube[2];


  pid_t pid = fork() ;
```

```
  if( pid < 0 )
    return -1;

  if( pid == 0 ){

    execv( arguments[0], arguments );
    perror( "execv" );
    exit( 1 );
  }

  if( fin )
    return 1;

  return tube[0];

}
```

pour obtenir le comportement suivant :

- l'entrée standard du processus enfant (exécutant le programme spécifié par `arguments`) doit correspondre au fichier dont le descripteur est `desc` ;
- si `fin == 0`, sa sortie standard doit également être redirigée sur un tube anonyme – le vecteur `tube` approprié est déjà défini ; dans ce cas, le processus parent retourne `tube[0]` ;
- si `fin == 1` en revanche, la sortie standard de l'enfant n'est pas redirigée et le processus parent retourne 1.

Inutile de recopier toute la fonction, en utilisant les numéros de lignes il suffit d'indiquer le code à insérer et l'endroit d'insertion.

Exercice 6 :

Il est facile d'ajouter des octets à la fin d'un fichier. Ajouter des octets au début d'un fichier demande un peu de travail. Le but de l'exercice est d'écrire une commande (fonction `main`) telle que

`./a.out prefix file`

ajoute `prefix` au début du fichier `file`. Par exemple si on exécute

```
echo "mon fichier" > f.txt
./a.out   "prefix a ajouter" f.txt
```

le fichier `f.txt` contient

`prefix␣a␣ajoutermon␣fichier`

Le programme implémentera l'algorithme suivant :

(1) rallonger le fichier de `strlen( argv[1] )` octets ;

(2) créer une image mémoire du fichier (`mmap`) ;

(3) déplacer l'ancien contenu du fichier à la fin de la mémoire (`memmove`) ;

(4) copier la chaîne `argv[1]` au début de la mémoire `mmap` ;

(5) effectuer `munmap`.

Exercice 7 :

Il est impossible de poser un verrou sur un tube (nommé ou anonyme, c'est-à-dire `fifo` ou `pipe`). Ceci peut être un problème quand plusieurs processus écrivent dans le tube en parallèle et nous ne voulons pas que les octets écrits se mélangent [1]. Si le nombre d'octets est supérieur à `PIPE_BUF` alors les octets écrits peuvent se mélanger avec les octets écrits par d'autres processus.

Vous devez écrire une fonction :

```
long ecrire_fifo( const char *nom_fifo, const char *buf, size_t len)
```

prenant en paramètre le nom d'un tube nommé, `nom_fifo`, et y écrivant `len` octets qui se trouvent à l'adresse `buf`.

Pour assurer l'atomicité de l'opération `ecrire_fifo`, elle commence par poser un verrou exclusif sur un fichier dont le nom est obtenu en concaténant `".lock"` à `nom_fifo` (le fichier doit être éventuellement créé s'il n'existe pas). Après avoir posé le verrou, la fonction écrit le pid du processus dans le fichier. Après l'écriture dans le tube, la fonction écrit -1 dans le fichier et lève le verrou.

Vous pouvez supposer que vous avez une fonction `char *concat(const char *s, const char *t)` qui retourne la concaténation de chaînes de caractères pointées par `s` et `t`.

Exercice 8 :

On suppose qu'un processus pose un verrou exclusif sur un fichier (verrou BSD `flock`).

Un autre processus effectue l'opération `write` sur le même fichier *sans avoir posé un verrou.*

Est-ce que cette opération

(1) réussit ?

(2) échoue ?

(3) bloque jusqu'à la levée du verrou ?

Exercice 9 :

On considère le programme suivant :

```
char *path;

pid_t lancer(const char *path){

  pid_t pid = fork();

  if( pid < 0 ){

  }

  if( pid > 0 ){

  }

```

1. Le système garantit une écriture atomique uniquement si le nombre d'octets à écrire par un processus est inférieur ou égale à `PIPE_BUF`, la constante définie dans `limits.h`.

```
  execl(path, path, (char *)0);


  return -1;
}



int main(int argc, char **argv){
    path = argv[1];
    enfant = lancer(path);

    while( 1 ){
      sleep(2);
    }

}
```

Compléter la fonction `lancer` de la façon suivante :

(1) si `fork` échoue, alors le processus doit terminer (on suppose que le processus sans enfant n'a pas d'utilité).

(2) le processus parent retournera le pid de l'enfant ;

(3) si `execl` échoue, il faut faire en sorte que les processus parent **et** enfant terminent (vous avez le droit d'envoyer un signal, par exemple `SIGKILL`).

Dans `main`, il faut ajouter le code qui permet de lancer un nouvel enfant quand le processus enfant termine (peu importe pour quelle raison).
Pour cela vous devez installer un handler qui sera activé par un signal `SIGCHLD` envoyé à la terminaison de l'enfant.
Dans le handler :

(1) faites en sorte que l'enfant zombie soit supprimé,

(2) lancez un nouvel enfant (par l'appel à `lancer(path)`).

Programmation système avancée
Annexe

```
void *
mmap(void *addr, size_t len, int prot, int flags, int fd, off_t offset);
MAP_FAILED
PROT_NONE PROT_READ PROT_WRITE  PROT_EXEC
MAP_ANONYMOUS MAP_PRIVATE MAP_SHARED

int munmap(void *addr, size_t len);
int msync(void *addr, size_t len, int flags);
MS_ASYNC        Return immediately
MS_SYNC         Perform synchronous writes
MS_INVALIDATE   Invalidate all cached data


int open(const char *path, int oflag, ...);
ssize_t read(int fildes, void *buf, size_t nbyte);
ssize_t write(int fildes, const void *buf, size_t nbyte);
off_t lseek(int fildes, off_t offset, int whence);
SEEK_SET SEEK_CUR SEEK_END

int dup(int fildes);
int dup2(int fildes, int fildes2);

int flock(int fd, int operation);
LOCK_SH   LOCK_EX   LOCK_NB   LOCK_UN

int fcntl(int fildes, int cmd, ...);
F_GETLK  F_SETLK  F_SETLKW
struct flock {
    off_t       l_start;    /* starting offset */
    off_t       l_len;      /* len = 0 means until end of file */
    pid_t       l_pid;      /* lock owner */
    short       l_type;     /* lock type: read/write, etc. */
    short       l_whence;   /* type of l_start */
};

int pipe(int fildes[2]);
pid_t wait(int *stat_loc);
pid_t waitpid(pid_t pid, int *stat_loc, int options);

int  ftruncate(int fildes, off_t length);
int  truncate(const char *path, off_t length);

int fstat(int fildes, struct stat *buf);
int stat(const char *restrict path, struct stat *restrict buf);
struct stat { /* when _DARWIN_FEATURE_64_BIT_INODE is NOT defined */
        dev_t    st_dev;    /* device inode resides on */
```

```
        ino_t     st_ino;     /* inode's number */
        mode_t    st_mode;    /* inode protection mode */
        nlink_t   st_nlink;   /* number of hard links to the file */
        uid_t     st_uid;     /* user-id of owner */
        gid_t     st_gid;     /* group-id of owner */
        dev_t     st_rdev;    /* device type, for special file inode */
        struct timespec st_atimespec;  /* time of last access */
        struct timespec st_mtimespec;  /* time of last data modification */
        struct timespec st_ctimespec;  /* time of last file status change */
        off_t     st_size;    /* file size, in bytes */
        quad_t    st_blocks;  /* blocks allocated for file */
        u_long    st_blksize;/* optimal file sys I/O ops blocksize */
        u_long    st_flags;   /* user defined flags for file */
        u_long    st_gen;     /* file generation number */
      };

int sigaction(int sig, const struct sigaction *restrict act,
    struct sigaction *restrict oact);

    struct sigaction{
      void (*sa_handler)(int);
      sigset_t sa_mask;
      int sa_flags;
    };
int  sigaddset(sigset_t *set, int signo);
int  sigdelset(sigset_t *set, int signo);
int  sigemptyset(sigset_t *set);
int  sigfillset(sigset_t *set);
int  sigismember(const sigset_t *set, int signo);
int  sigsuspend(const sigset_t *sigmask);

int  sigprocmask(int how, const sigset_t *restrict set,
                 sigset_t *restrict oset);
SIG_BLOCK  SIG_UNBLOCK  SIG_SETMASK

int  sigpending(sigset_t *set);

int  kill(pid_t pid, int sig);
```