Master1 – POSIX **2005-2006** **Examen**	- Durée : 3 heures - Toute documentation autorisée - Barème indicatif *Luciana Arantes, Bertil Folliot et Olivier Marin*

1. THREADS (11 PTS)

Nous voulons offrir un ensemble de fonctions qui permet à des threads de communiquer par tubes. Le tube de cet exercice est un mécanisme de communication qui permet à une thread de lire ou d'écrire des caractères.

A cette fin, nous avons défini la structure suivante :

```
struct tube {
  char vect [TAILLE_PIPE];  /* vecteur pour sauvegarder le contenu du tube */
  int lect_off, ecr_off;  /* indice de la prochaine case de vect à lire ou libre respectivement */
  int nb_char; /* nombre de caractères dans le tube */
 /* à compléter */
};
```

où :

vect [TAILLE_PIPE] : vecteur de taille TAILLE_PIPE qui sauvegarde les caractères qui n'ont pas été encore lus dans le tube. Ce vecteur est géré de façon **circulaire** ;

lect_off et *ecr_off* : indice de la prochaine case de *vect* à lire ou à écrire respectivement ;

nb_char : nombre de caractères dans *vect*.

Les fonctions que nous offrons sont les suivantes :

struct tube* creer_tube_thread (void);

Fonction qui renvoie un pointeur vers une structure tube allouée dynamiquement.

void detruire_tube_thread (struct tube* tube);

Fonction qui libère une structure tube allouée.

int lire_tube_thread (struct tube* tube, char *buf, int n);

Fonction **bloquante** qui lit au plus ***n*** caractères du tube.

L'algorithme correspondant à cette fonction est le suivant :

Si le tube contient *x* caractères, la fonction extrait du tube *nb_lus = min (x, n)* caractères qui sont alors copiés dans *buf*;

Si le tube est vide, la thread est mis en sommeil jusqu'à ce que le tube ne soit plus vide.

La fonction renvoie le nombre de caractères lus dans le tube (*nb_lus* caractères).

int ecrire_tube_thread (struct tube* tube, char *buf, int n);

Fonction qui écrit de façon atomique ***n*** caractères dans le tube, s'il y a de la place.

L'algorithme correspondant à cette fonction est le suivant :

S'il y a au moins *n* emplacements libres dans le tube, une écriture atomique est réalisée et le nombre de caractères écrits est renvoyé. Dans ce cas **toutes les threads** lecteurs en attente sur le tube seront réveillées.

Sinon la fonction renvoie -1.

Observations :

- La fonction *creer_tube_thread* n'est appelée que par la *thread main*, avant qu'elle ne créée les threads qui utiliseront le tube.
- La thread *main* n'utilise jamais le tube.
- Nous considérons que les programmes utilisent correctement les fonctions.
- Vous trouverez en annexe un exemple d'utilisation de ces fonctions où une thread envoie un message à une autre par le tube.

1.1 (2 pts)

Indiquez quelles sont les variables nécessaires pour compléter la struct tube. Donnez le code des fonctions *creer_tube_thread* et *detruire_tube_thread*.

1.2 (2 pts)

Donnez le code de la fonction *lire_tube_thread*.

1.3 (2 pts)

Donnez le code de la fonction *ecrire_tube_thread*.

1.4 (1 pt)

Supposons qu'il y a une variable globale : *int nb_thread;* qui contrôle le nombre de threads en exécution (sans compter la thread main elle-même). Elle est mise à jour par la thread main.

Modifiez la fonction *ecrire_tube_thread* afin qu'elle délivre un signal SIGPIPE à une thread écrivain si au moment de l'écriture des caractères sur le tube il n'existe pas d'autres threads qui pourraient les lire.

1.5 (2 pts)

Considérez un programme où plusieurs threads lisent (threads lecteur) dans le tube et d'autres y écrivent (threads écrivain). Supposons qu'à un certain moment, la thread *main* annule une des threads lecteur en appelant la fonction *pthread_cancel*.

Est-que cette annulation pourrait bloquer les autres threads lecteur ? Justifiez votre réponse.

1.6 (2 pts)

On souhaite maintenant que la fonction *ecrire_tube_thread* soit bloquante lorsque le tube est plein. Indiquez quelles modifications on doit apporter aux fonctions de lecture/écriture ainsi qu'à la structure de tube. Ecrivez ces nouvelles fonctions.

2. IPC SYSTEM V (6 PTS)

Un programmeur (très) inexpérimenté a voulu rédiger une petite application dans laquelle un processus émetteur envoie un message mot par mot à **deux** processus récepteurs de sa famille. Le résultat recherché de l'application est de faire afficher (NB : une seule fois) le message original dans son ordre initial. Par exemple, si l'émetteur envoie successivement les mots "je", suis", "en", "examen", l'affichage final de l'application sera : "je suis en examen". Le code produit par notre programmeur est le suivant :

```
int i, pid, n = 6;

void send(char *buf) {
    char *msg[] = {"Je", "suis", "un", "petit", "programme", "\n"};
    for(i = 0; i < n; i++) {
      strcpy(buf, msg[i]);
    }
    exit(0);
}
```

```
void receive(char *buf) {
    if (pid = fork() != -1) {
      for(i = 0; i < (n/2); i++) {
        printf("%s ", buf);
      }
}   }

int main()
{
    char *shared;

    if ((pid = fork()) == -1) {perror("fork"); exit(1);}
    if (pid == 0)
      send(shared);
    else
      receive(shared);

    if (pid == 0) exit(0);
    for (i = 0; i < 2; i++)
      wait(0);
    printf("fin du programme\n");
    return 0;
}
```

2.1 (1 pt)

Expliquez pourquoi ce programme ne peut pas fonctionner.

2.2 (3 pts)

Sans altérer les créations de processus, corrigez le code à l'aide d'IPCs System V (en évitant toutefois l'utilisation de files de messages) afin que l'application fonctionne.

2.3 (2 pts)

Quelles modifications faut-il apporter pour que chacun des processus récepteurs affiche le message dans son intégralité ?

Voici un exemple de résultat attendu au niveau de l'affichage :

```
Récepteur1> Je
Récepteur2> Je
Récepteur2> suis
Récepteur1> suis
Récepteur1> un
Récepteur2> un
...
```

3. IPC POSIX (3 PTS)

3.1 (3 pts)

Un processus P1 crée un processus P2 (fils de P1) qui à son tour crée un processus P3 (fils de P2 et petit-fils de P1).

Lorsque le processus P3 est créé, il fait passer son PID à son grand-père P1 au moyen d'un segment de mémoire partagée. P1 affiche alors : "petit-fils (PID = <pid_p3>) créé". Le processus P3 se termine juste après. Quand P2 prend connaissance de la terminaison de P3, il dépose son propre PID dans le même segment de mémoire partagée à destination de P1, puis se termine. P1 attend la terminaison de P3 pour afficher "petit-fils terminé". Ensuite, P1 attend la terminaison de son fils P2 pour afficher : "fils (PID = <pid_p2>) terminé".

Codez un tel programme en utilisant exclusivement les sémaphores et la mémoire partagée POSIX.

ANNEXE

```
/* variables globales */
struct tube* tube;
....
void *lecteur (void *arg) {
   char buffer [10];
   lire_tube_thread (tube, buffer, 5);
   printf ("contenu buffer :%s\n", buffer);
   return NULL;
}
void *ecrivain (void *arg) {
   char buffer [10];
   strcpy (buffer,"toto");
   ecrire_tube_thread (tube,buffer,5);
   return NULL;
 }

int main (int argc, char ** argv) {
  int status;
  pthread_t tid_lecteur, tid_ecrivain;

   if ((tube = creer_tube_thread ()) == NULL)
      return EXIT_FAILURE;

   if  ((pthread_create (&(tid_lecteur), NULL, lecteur, (void *)NULL ) != 0) | |
       (pthread_create (&(tid_ecrivain), NULL, ecrivain, (void *)NULL ) != 0)) {
          detruire_tube_thread (tube);
          return EXIT_FAILURE;
    }
     if ( (pthread_join (tid_lecteur, (void**) &status) !=0) | |
         (pthread_join (tid_ecrivain, (void**) &status))) {
            detruire_tube_thread (tube);
             return EXIT_FAILURE;
   }
  detruire_tube_thread (tube);
  return EXIT_SUCCESS;
}
```