Année 2010-2011 1ère session

PROGRAMMATION SYSTÈME
PG 204
DENIS BARTHOU

Filière : Informatique Année : première Semestre : premier semestre

Date de l'examen : 10 janvier 2011 Durée de l'examen : 2h

Documents autorisés ☒ sans document ☐
Calculatrice autorisée ☒ non autorisée ☐

Autre :

SUJET

1 Descripteurs de fichier

Ecrivez un programme qui écrit vers une FIFO tout ce qu'il lit de l'entrée standard. La FIFO est nommée "input" et on supposera qu'elle existe déjà (ne pas la créer). On écrira le programme en utilisant les fonctions `open`, `read`, `write` et `close`. On ne fera aucune gestion des erreurs et on omettra les includes. Attention, votre code ne doit pas excéder 15 lignes !

2 Signaux

Ecrivez un programme `run` qui prend le nom d'un exécutable `exec` en paramètres de ligne de commande et lance cet exécutable 5 fois. L'exécutable `exec` ne prendra pas d'argument. Dès que l'exécutable `exec` termine son exécution normalement, le programme `run` le relance, et après la cinquième exécution de l'exécutable, il n'est plus relancé. Si l'exécutable `exec` est arrêté à cause d'un signal (comme une erreur de segmentation), le programme `run` devra afficher un message et terminer. On utilisera les fonctions `fork`, `waitpid` et `execvp`.
Pour simplifier l'écriture du programme, on ne fera aucune gestion des erreurs et on supposera qu'il y a bien toujours 1 paramètre en ligne de commande (le nom de l'exécutable). On omettra par ailleurs les includes.

3 Programme mystère

Considérez le programme suivant:

```
int main (int argc, char *argv[])
{
 int i = 0;
 int tube[2];
 int in = STDIN_FILENO;
 int out = STDOUT_FILENO;
 int nb = argc -1;

 do {
   pipe(tube);
   if (fork() != 0) {
     close(tube[1]); in = tube[0];
   } else {
     close(tube[0]); out = tube[1];
```

```
      break;
    }
    i++;
  } while (i < nb-1)

  if (in != STDIN_FILENO) {
     dup2(in, STDIN_FILENO); close(in);
  }

  if (out != STDOUT_FILENO) {
    dup2(out, STDOUT_FILENO); close(out);
}
 execlp(argv[i+1], argv[i+1], NULL);
}
```

- Expliquez en détail le déroulement de ce code (un schéma peut être utile).
- Quel sera le résultat de l'exécution lorsque ce programme est lancé avec les paramètres "ls cat wc"

4 Threads et synchronisation

On considère un système d'exploitation devant ordonnancer des processus sur une machine à 8 coeurs. Au maximum, 8 processus peuvent être exécutés en même temps, et s'il y a plus de 8 processus prêts à être exécutés, ceux au delà de 8 sont placés dans une file d'attente.
Le système d'exploitation considéré stocke les processus en attente dans un file d'attente `queue_t wait_queue;` avec `queue_t` le type C correspondant à la structure d'une file de processus. Les fonctions de gestion de la file (une FIFO) sont:

- `process_t getFirst(queue_t q)`: retourne le premier processus de la file d'attente et l'enlève de la file. Si la file est vide, la fonction fait un segfault.
- `queue_t add(queue_t q, process_t p)`: ajoute un processus à la file d'attente, et retourne la nouvelle file d'attente. Si la file est pleine, la fonction fait un segfault.
- `int isEmpty(queue_t q)`: retourne 0 si la file d'attente n'est pas vide, 1 si elle est vide.
- `int isFull(queue_t q)`: retourne 0 si la file d'attente n'est pas pleine, 1 si elle est pleine.

Par ailleurs, l'ordonnanceur du système d'exploitation lance l'exécution d'un processus avec l'appel à `void exec(process_t p)`. Cette fonction lance le processus et retourne (toujours) à la fin de l'exécution du processus (ce n'est pas une fonction execvp).
L'ordonnanceur gère la file d'attente des processus avec 8 threads (1 thread par coeur), la file d'attente étant partagée. Chaque thread de l'ordonnanceur fait, en boucle, les actions suivantes: il regarde si la liste est vide, si elle n'est pas vide, prend le premier processus, le lance.

a- Ecrire le code d'un thread de l'ordonnanceur. On n'écrira pas le code de lancement de ces threads.

b- Quels sont les problèmes pouvant se poser du fait qu'il y a 8 threads exécutant ce code ? Quelle est la section critique du code ? Montrer sur un exemple un problème pouvant se produire avec 2 threads (on déroulera les instructions des deux threads dans le temps).

c- Modifier le programme des threads avec des `pthread_mutex` pour régler ce problème. Attention, on doit toujours pouvoir exécuter 8 processus sur les 8 coeurs simultanément. Ecrire une fonction qui fait l'initialisation des mutex.

On considère la fonction suivante:

```
void run(process_t p) {
  while (isFull(wait_queue)) ;
  add(wait_queue, p);
}
```

Cette fonction est appelée par un seul thread du système d'exploitation et permet d'ajouter dans la file d'attente de nouveaux processus à exécuter.

d- La fonction fait de l'attente active. Pourquoi cela n'est pas une bonne solution ? Quel autre problème à cette fonction ?

e- Modifiez la fonction en rajoutant des mutex pour régler ces deux problèmes.

f- Montrer que votre programme n'a pas de deadlock, en montrant que l'une des 4 conditions vues en cours pour qu'il y ait deadlock n'est pas vérifiée.

5 Annexe

```
SYNOPSIS
     #include <unistd.h>

     int
     pipe(int fildes[2]);

DESCRIPTION
     The pipe() function creates a pipe (an object that allows unidirectional data flow) and allocates a pair of file
     descriptors.  The first descriptor connects to the read end of the pipe; the second connects to the write end.

     Data written to fildes[1] appears on (i.e., can be read from) fildes[0].  This allows the output of one program to be
     sent to another program: the source's standard output is set up to be the write end of the pipe; the sink's standard
     input is set up to be the read end of the pipe.  The pipe itself persists until all of its associated descriptors are
     closed.
```

```
SYNOPSIS
     #include <unistd.h>

     int
     dup(int fildes);

     int
     dup2(int fildes, int fildes2);

DESCRIPTION
     Dup() duplicates an existing object descriptor and returns its value to the calling process (fildes2 = dup(fildes)).
     The argument fildes is a small non-negative integer index in the per-process descriptor table.
     In dup2(), the value of the new descriptor fildes2 is specified.  If this descriptor is already in use, the descriptor
     is first deallocated as if a close(2) call had been done first.
```

```
SYNOPSIS
     #include <sys/wait.h>

     pid_t
     wait(int *stat_loc);

     pid_t
     waitpid(pid_t pid, int *stat_loc, int options);

DESCRIPTION
     The waitpid() function suspends execution of its calling process until stat_loc information is available for a terminated
     child process, or a signal is received.  On return from a successful waitpid() call, the stat_loc area contains terminatio

     The pid parameter specifies the set of child processes for which to wait.  If pid is -1, the call waits for any child
     process.  If pid is 0, the call waits for any child process in the process group of the caller.  If pid is greater
     than zero, the call waits for the process with process id pid.  If pid is less than -1, the call waits for any process
     whose process group id equals the absolute value of pid.
```


```
     The pid parameter specifies the set of child processes for which to wait.  If pid is -1, the call waits for any child
     process.  If pid is 0, the call waits for any child process in the process group of the caller.  If pid is greater
     than zero, the call waits for the process with process id pid.  If pid is less than -1, the call waits for any process
     whose process group id equals the absolute value of pid.
```