MI014 - Programmation Système Avancée avec POSIX 2010-2011 Deuxième Examen	Durée : 3 heures Toute documentation autorisée Barème indicatif

N.B: Lorsque vous devez donner un code de programme, vous n'êtes pas tenu de prendre en compte les erreurs lors des appels système. A moins qu'il n'en soit précisé autrement dans l'énoncé d'un exercice, on considère que les appels réussissent toujours s'ils sont formulés correctement. De même, il ne vous est pas demandé de préciser les fichiers d'en-tête associés aux appels que vous utilisez.

1. QUESTIONS DE COURS : ORDONNANCEMENT POSIX.4 3pts

1.1 (1 pt)

Dans POSIX.4, indiquez ce qui se passe lorsqu'une thread change la priorité d'une autre thread

a) avec une priorité supérieure à la sienne,

b) avec une priorité inférieure à la sienne

On considère maintenant le code suivant :

```
sched_setscheduler(0, SCHED_FIFO, 0);
int pid = fork();
if (pid) {
    sched_yield();
    printf("pere");
} else
    printf("fils");
```

1.2 (1 pt)

Quel est l'affichage produit par ce code ? Quel est l'affichage produit par ce code si l'on retire le `sched_yield()` ? Justifiez vos réponses.

1.3 (1 pt)

Quel est l'affichage produit par ce code si l'on remplace `sched_yield()` par `sched_setscheduler(pid, SCHED_RR, 0)` ? Justifiez votre réponse.

2. SOCKETS 5 pts

0 -1

Nous considérons N processus s'exécutant chacun sur une machine différente. Pour se synchroniser ils appellent la fonction `int barrier(int nproc, int sock)` où `nproc` est le nombre de processus qui participent à la barrière et `sock` est la socket utilisée par chaque processus pour communiquer avec les autres au moyen du protocole IP Multicast. La fonction `barrier` renvoie 0 en cas de succès ou -1 s'il y a eu un problème lors de l'utilisation des fonctions liées aux sockets.

Nous nous intéressons aussi à la fonction `int init_barrier()`, qui renvoie le descripteur de la socket de communication (-1 si elle échoue), permet l'initialisation de la communication nécessaire entre les processus qui participent à la barrière.

Par exemple, si `N=2` et chaque processus exécute le programme :

```
#define N  2
int fd;
int main (int argc, char * argv[]) {
        fd = init_barrier ();
        printf ("avant barrier 1 \n") ;
        barrier (N,fd);
        printf ("avant barrier 2 \n") ;
        barrier (N,fd);
}
```

Dans cet exemple, un processus n'affichera jamais "avant barrier 2" avant que l'autre ait affiché "avant barrier 1".

Nous considérons qu'il n'y a pas de perte de message, que chaque processus connait N, que l'adresse IP de diffusion multicast est 255.0.0.10 et que les processus attendent un message sur le port 7020

2.1 (2 pts)

Implémentez la fonction `int init_barrier()`, en prenant bien en compte les erreurs possibles lors des appels système.

2.2 (3 pts)

Implémentez la fonction `int barrier(int nproc, int sock)`, en prenant bien en compte les erreurs possibles lors des appels système.

3. THREADS 7 pts

Supposons qu'un processus initialise le vecteur `int vect[K]`, de taille `K`, avec des valeurs aléatoires entre 0 et K en appelant la fonction `rand` :

```
(int) ((float)K*rand()/ (RAND_MAX +1.0));.
```

Après cette initialisation, la thread main crée **N** threads **détachées** (**N<K**), qu'on appellera threads consommatrices, et qui vont venir lire les cases de `vect[]` à tour de rôle.

Tant qu'il y a des cases à consommer, une thread consommatrice accède au vecteur `vect[]`, lit le contenu d'une case non encore consommée et affiche celui-ci. Notez qu'une valeur de `vect[]` ne doit être consommée qu'une et une seule fois fois. De plus, une thread ne peut se terminer qu'une fois que les K valeurs de `vect[]` ont été consommées. La thread main affiche alors le *tid* de la thread qui a consommé la dernière valeur (`vect [K-1]`) et se termine.

N.B: la valeur de N est strictement inférieure à celle de K.

3.1 (1,5 pts)

Détaillez les variables dont vous allez avoir besoin pour implémenter un tel programme, en expliquant pour chacune son utilité et en précisant bien sa valeur d'initialisation.

3.2 (1 pt)

Donnez le code de la thread main, qui initialise les variables et crée les threads consommatrices.

3.3 (4,5 pts)

Donnez le code exécuté par chacune des threads consommatrices.

4. SIGNAUX 5 pts

On souhaite créer une boucle synchronisée de processus. Le père crée une chaine de N processus. Chaque fils attend le signal `SIGUSR1`.
A la réception d'un `SIGINT`, le père envoie un `SIGUSR1` au fils 1, qui envoie a son tour un `SIGUSR1` au fils 2, etc. le dernier fils renvoie un `SIGUSR1` au père, qui affiche alors "`END SYNC`".

4.1 (2 pts)

Programmez cette boucle synchronisée.

4.2 (1 pt)

Modifiez le programme pour que le premier fils affiche "`BEGIN LOOP`", et le dernier fils "`END LOOP`"

4.3 (2 pts)

Modifiez le programme pour qu'à la réception de `SIGQUIT`, le père transmette le signal au fils1, etc. comme dans la question (4.1). Cependant, à la réception d'un `SIGQUIT`, après avoir transmis le signal au suivant, le processus se termine. Le dernier fils envoie un `SIGKILL` au père.