Université de Montpellier
-
Faculté des Sciences

Session : 1	Durée de l'épreuve : 2 heures
Date : 04 janvier 2017	Documents autorisés : aucun
Master Informatique	Matériel utilisé : aucun
Master $1^{ère}$ année : Réseaux et communication (HMIN101M)	

Sujet

Les exercices sont indépendants et peuvent être traités dans l'ordre qui vous semble le plus efficace. lisez attentivement l'énoncé en entier avant de répondre. Le barème est donné à titre indicatif et peut changer.

Dans les schémas algorithmiques demandés, vous devez utiliser correctement les opérations (en C) fournies en cours pour la manipulation des outils étudiés. Les prototypes de ces opérations sont fournis en annexe.

Enfin, vous devez mettre l'accent sur la lisibilité de vos réponses.

Exercice 1 : questions diverses (5 points)

1. Réponse unique : en mode RPC, pour se connecter à un serveur, un client doit spécifier (en plus d'un protocole de communication) :
   (a) l'hôte du serveur et un numéro de port,
   (b) l'hôte du serveur, un numéro de port, un identifiant de programme et une version,
   (c) l'hôte du serveur, un identifiant de programme et une version,
   (d) l'hôte du serveur, un identifiant de procédure et son numéro.
2. Réponse unique : toujours en mode RPC, un serveur peut réaliser plusieurs implémentations de la même procédure :
   (a) vrai,
   (b) faux.

   Si la réponse est vrai, un client peut utiliser une seule implémentation : **(aa)** vrai, **(ab)** faux. Justifier.
3. Avec le protocole HTTP, il est possible d'implémenter des solutions pour un échange bidirectionnel (initié des deux côtés) entre le client et le serveur. Ces solutions sont connues sous les noms de long polling et HTTP streaming. Choisir **la** bonne affirmation dans les phrases ci-dessous :
   (a) dans le HTTP streaming, on indique dans la requête le fait de vouloir une réponse partielle,
   (b) dans le long polling, le client envoie des requêtes HTTP avec une certaine fréquence choisie par le développeur, sans attendre la réponse du serveur,
   (c) dans le HTTP streaming, le serveur envoie des réponses dans lesquels il indique que ce sont des réponses partielles,
   (d) dans le long polling, le serveur envoie des réponses complètes et impose au client de lui envoyer des requêtes juste après.
4. Avec le protocole Websockets, il est possible :
   (a) d'établir une connexion TCP (avec échange de messages bidirectionnel) en exploitant les fonctions du protocole HTTP,
   (b) de réaliser des échanges de messages bidirectionnels de façon standardisée en s'appuyant sur le HTTP streaming uniquement,

(c) de réaliser des échanges de messages de façon standardisée en se basant sur le long polling uniquement,

(d) de réaliser des échanges de messages de façon standardisée en s'appuyant sur ces deux solutions combinées.

Choisir **une seule** réponse.

5. Soit deux fonctions $f1$ et $f2$ implémentées et utilisées comme suit :

```
1  int x;
2  pthread_mutex_t verrou;
3  pthread_cond_t changeValX;
4
5  //fonction sans synchronisation
6  void calcul(){...}
7
8  void *f1 (void * p){
9    pthread_mutex_lock(&verrou);
10   cout<<"f1 attend que x soit > 10"<<endl;
11   pthread_cond_wait(&changeValX, &verrou);
12   pthread_mutex_unlock(&verrou);
13
14   cout <<"f1 commence son calcul"<<endl;
15   calcul();
16   cout <<"f1 se termine"<<endl;
17   pthread_exit(NULL);
18 }
```

```
1  void *f2 (void * p){
2    pthread_mutex_lock(&verrou);
3    cout <<"f2 modifie x"<<endl;
4    x = 20;
5    pthread_mutex_unlock(&verrou);
6    cout <<"f2 reveille f1"<<endl;
7    pthread_cond_signal(&changeValX);
8    cout <<"f2 se termine"<<endl;
9    pthread_exit(NULL);
10 }
11
12 int main(){
13   pthread_t idf1, idf2;
14   x = 5;
15   pthread_mutex_init(&verrou, NULL);
16   pthread_cond_init(&changeValX, NULL);
17   pthread_create(&idf1, NULL, f1, NULL);
18   pthread_create(&idf2, NULL, f2, NULL);
19   pthread_join(idf1, NULL);
20   pthread_join(idf2, NULL);
21 }
```

A l'exécution du programme principal, un interblocage peut se produire :

(a) donner une trace d'exécution (ou un ordre d'exécution des instructions) provoquant cet interblocage.

(b) expliquer comment corriger cette erreur.

Exercice 2 : envoi/réception de messages (4 points)

Soit la définition des fonctions suivantes :

- **int envoiTCP(int socket, char * message, int size)** : envoie, sur *socket*, la suite des *size* octets stockés à l'adresse *message*. Elle suppose que *socket* est un descripteur d'une socket valide, i.e. qui a été configurée et connectée à une socket distante avant son invocation/appel. L'envoi se fait en utilisant le protocole TCP/IP. Enfin, la fonction retourne 1 en cas de succès de l'envoi des *size* octets, 0 si la socket a été fermée par la couche transport (si, par exemple, le récepteur s'est déconnecté), -1 sinon.
- **int receptionTCP(int socket, char * message, int size)** : reçoit *size* octets sur *socket* et les stocke à l'adresse *message*. Elle suppose que *socket* est un descripteur d'une socket valide, i.e. qui a été configurée et connectée à une socket distante avant son invocation/appel. La réception se fait en utilisant le protocole TCP/IP. Enfin, la fonction retourne 1 en cas de succès de la réception des *size* octets, 0 si la socket a été fermée par la couche transport (si, par exemple, l'émetteur s'est déconnecté), -1 sinon.

Ecrire le schéma algorithmique des deux fonctions, en utilisant les fonctions *send*(...) et *recv*(...).

Exercice 3 : IPC - sémaphores (5 points)

Dans cet exercice, nous souhaitons mettre en place un jeu de cartes, avec n processus représentant les joueurs. Sur l'ensemble du jeu, chaque joueur doit jouer x tours, chaque tour arrive en respectant un ordre global défini comme suit :

En numérotant les processus joueurs $Joueur1$ à $JoueurN$, ces derniers interviennent dans l'ordre $Joueur1$, $Joueur2$, ..., $JoueurN-1$, $JoueurN$, $Joueur1$, $Joueur2$, ..., $JoueurN-1$, $JoueurN$, ... jusqu'à ce que tous les joueurs aient effectué les x tours.

On souhaite résoudre ce problème d'ordre circulaire en utilisant un tableau de sémaphores à un seul élément (un seul sémaphore) et en partant du squelette algorithmique suivant pour un processus joueur $Joueurj$ $(1 <= j <= N)$:

```
1  Joueur j
2
3    int idSem = identifiant du tableau de semaphores;
4
5    for (int i = 1; i <= x; i++){
6      // j'attends mon tour
7      ...
8      // c'est a moi de jouer
9      je_joue(); // cette fonction est fournie (ne pas l'implementer).
10     // je passe le tour au joueur suivant
11     ...
12   }
13 Fin Joueur j
```

1. Reprendre ce schéma algorithmique en :
   - remplaçant la ligne 3 par les instructions nécessaires à la récupération de l'identifiant du tableau de sémaphores.
   - remplaçant les lignes en pointillés par les instructions permettant de compléter le schéma algorithmique (note : une ligne en pointillés ne représente pas nécessairement une seule instruction). Pour la manipulation du sémaphore, vous devez utiliser les opérations P et V décrites comme suit :

```
1  void P(int idSem, unsigned int i,
2         unsigned int k) {
3    struct sembuf semoi;
4    semoi.sem_num = i;
5    semoi.sem_op = -k;
6    semoi.sem_flg = 0;
7    semop(idSem, &semoi, 1);
8  }
```

```
1  void V(int idSem, unsigned int i,
2         unsigned int k){
3    struct sembuf semoi;
4    semoi.sem_num = i;
5    semoi.sem_op = k;
6    semoi.sem_flg = 0;
7    semop(idSem, &semoi, 1);
8  }
```

   Remarque : vous devez reprendre le schéma algorithmique en entier sur votre copie. Toute solution proposant des bouts de code séparés ne sera pas prise en compte et sera notée 0.

   Avez vous pensé à tous les joueurs (de 1 à N) ? Si ce n'est pas le cas, complétez votre solution.

2. Quelle est la valeur initiale du sémaphore ?
3. Montrer que votre solution répond bien au problème. En particulier, montrer que chaque processus joueur attend son tour et agit uniquement à son tour.
4. Que peut-il se passer si semoi.sem_flg était positionné à SEM_UNDO au lieu de 0 dans P et V ? Justifier avec un exemple précis.

Note 1 : Tout processus $Joueurj$ sait qu'il est le j^{eme} joueur.

Note 2 : Nous supposons qu'un processus P_{init} s'est chargé de la création du tableau de sémaphores et de son initialisation avant le début du jeu. Vous n'avez donc pas à vous préoccuper de cette partie.

Note 3 : La solution demandée doit permettre le lancement des processus joueurs dans un ordre quelconque. Toutefois, chaque processus doit intervenir seulement à son tour. Exemple : si $Joueur3$ est lancé en premier, il devra attendre jusqu'au moment où il sera à son tour de jouer.

Exercice 4 : File de messages (4 points)

Cet exercice reprend le jeu de cartes de l'exercice 2 pour résoudre le problème d'ordre circulaire en utilisant, cette fois, une file de messages.

Proposer un schéma algorithmique pour les joueurs. Remarque : une seule file de message est à utiliser et aucun autre moyen de communication/synchronisation entre processus n'est permis.

Note 1 : Tout processus $Joueurj$ sait qu'il est le j^{eme} joueur.

Note 2 : Nous supposons qu'un processus P_{init} s'est chargé de la création de la file de message avant le début du jeu. Vous n'avez donc pas à vous préoccuper de cette partie.

Exercice 5 : RdV (4 points)

Le code suivant tente d'implémenter un rendez-vous entre N threads. Chaque thread exécute la fonction *participant*(...). Cette fonction effectue une première étape d'un travail puis attend que tous les autres threads soient arrivés au rendez-vous avant de poursuivre son travail.

```
1  struct predicatRdv {
2    int nb_Thread_au_rdv; // initialisee a 0
3    int nb_Threads; // initialisee au nombre de threads a lancer
4    pthread_mutex_t verrou;
5    pthread_cond_t tous_arrives;
6  };
7
8  // structure en parametre d'un thread
9  struct params {
10   int idThread; // identifiant de thread, de 1 a N
11   struct predicatRdv * varPartagee;
12 };
13
14 void * participant (void * p){
15   struct params * args = (struct params *) p;
16   struct predicatRdv * predicat = args -> varPartagee;
17
18   // simulation d'un long calcul
19   cout <<"Thread "<< args -> idThread << ": je commence mon travail"<<endl;
20   sleep (5);
21
22   pthread_mutex_lock(&(predicat->verrou));
23   (predicat->nb_Thread_au_rdv)++;
24   cout <<"Thread "<< args -> idThread << " : je suis au rdv"<<endl;
25
26   if (predicat->nb_Thread_au_rdv < predicat->nb_Threads) {
27     cout <<"Thread "<< args -> idThread << " : j'attendts que tout le monde soit la" <<endl;
28     pthread_mutex_unlock(&(predicat->verrou));
29     pthread_cond_wait(&(predicat->tous_arrives), &(predicat->verrou));
30   }
31   else{
32     cout <<"Thread "<< args -> idThread << " : je suis le dernier"<<endl;
```

```
33      pthread_mutex_unlock(&(predicat->verrou));
34      pthread_cond_signal(&(predicat->tous_arrives));
35    }
36
37    cout <<"Thread "<< args -> idThread << " : je reprends le travail"<<endl;
38    sleep (4);
39    cout <<"Thread "<< args -> idThread << " : je termine"<<endl;
40    pthread_exit(NULL);
41  }
```

Nous supposons que le programme principale initialise correctement toutes les variables et lance correctement N threads (N >=1). Chaque thread est identifié par un indice unique entre 1 et N.

Ce code contient des erreurs que vous devez identifier et

1. Pour chaque erreur (à préciser), expliquer pourquoi il s'agit d'une erreur et quelle est sa conséquence à l'exécution (en donnant un exemple précis de trace d'exécution).
2. Corriger ces erreurs. Vous n'avez pas besoin de recopier tout le code, mais de préciser quelles sont les lignes concernées et les remplacer par vos corrections.

Annexe

Cette annexe contient un rappel des opérations vues en cours/TD/TP et qui pourraient vous être utiles.

- **key_t ftok(const char *pathname, int proj_id).**
- **int msgget(key_t key, int msgflg).**
- **int msgsnd(int msqid, const void *msgp, size_t msgsz, int msgflg).**
- **ssize_t msgrcv(int msqid, void *msgp, size_t msgsz, long msgtyp, int msgflg).**
- **int msgctl(int msqid, int cmd, struct msqid_ds *buf).**
- **int shmget(key_t key, size_t size, int shmflg) .**
- **void *shmat(int shmid, const void *shmaddr, int shmflg) .**
- **int shmdt(const void *shmaddr) .**
- **int shmctl(int shmid, int cmd, struct shmid_ds *buf) .**
- **int semget(key_t key, int nsems, int semflg) .**
- **int semop(int semid, struct sembuf *sops, unsigned nsops) .**
- **int semctl(int semid, int semnum, int cmd, ...)**
  avec
  union semun {int val; /* Value for SETVAL */
  struct semid_ds *buf; /* Buffer for IPC_STAT, IPC_SET */
  unsigned short *array; /* Array for GETALL, SETALL */
  struct seminfo *__buf; /* Buffer for IPC_INFO */};.
- **int pthread_create(pthread_t *thread, const pthread_attr_t *attr, void *(*start_routine) (void *), void *arg).**
- **int pthread_join(pthread_t thread, void **retval).**
- **int pthread_mutex_init(pthread_mutex_t *restrict mutex, const pthread_mutexattr_t *restrict attr).**
- **int pthread_mutex_destroy(pthread_mutex_t *mutex).**
- **int pthread_mutex_lock(pthread_mutex_t *mutex).**
- **int pthread_mutex_unlock(pthread_mutex_t *mutex).**
- **int pthread_cond_init(pthread_cond_t *restrict cond, const pthread_condattr_t *restrict attr).**
- **int pthread_cond_destroy(pthread_cond_t *cond).**
- **int pthread_cond_wait(pthread_cond_t *restrict cond, pthread_mutex_t *restrict mutex).**
- **int pthread_cond_broadcast(pthread_cond_t *cond).**
- **int pthread_cond_signal(pthread_cond_t *cond).**
- **int listen(int sockfd, int backlog).**
- **int accept (int descripteur, struct sockaddr *brCv, socklen t *lgbrCv).**
- **int connect(int sockfd, const struct sockaddr *addr, socklen_t addrlen) .**
- **ssize_t send(int sockfd, const void *buf, size_t len, int flags) .**
- **ssize_t sendto(int sockfd, const void *buf, size_t len, int flags, const struct sockaddr *dest_addr, socklen_t addrlen); .**
- **ssize_t recv(int sockfd, void *buf, size_t len, int flags) .**
- **ssize_t recvfrom(int sockfd, void *buf, size_t len, int flags, struct sockaddr *src_addr, socklen_t *addrlen) .**
- **int close(int fd).**