Université de Provence
MASTER INFORMATIQUE
Programmation Parallèle
et Distribuée
2007-2008

Programmation Parallèle et Distribuée

Examen – 16 juin 2008

3h00

Aucun document autorisé. Les réponses doivent être argumentées.
Les parties sont indépendantes. Le barème est indicatif.

A Programmation Parallèle

A.1 Qu'est-ce qu'une machine PRAM ? Bien préciser les différents mode d'accès possibles à la mémoire partagée. *(1 point)*

A.2 Etant donné un tableau de n entiers, on cherche à déterminer son maximum. Proposer un algorithme pour une machine CREW avec p processeurs. On pourra commencer par répondre pour des valeurs particulières de p.

Rappeler la définition de la complexité et de l'efficacité. Quelles en sont les valeurs pour votre algorithme ?

(3 points)

B Synchronisation JAVA

B.3 Le programme `Mutex` donné en annexe est-il correct pour l'exclusion mutuelle ? (on considérera qu'on a adopté les mêmes conventions que dans le cours) *(2 points)*

B.4 Proposer une solution en Java au problème suivant, dit "du Père Noël". *(4,5 points)*

Le Père Noël dort au Pôle Nord tant qu'il n'est pas réveillé par ses neuf rênes ou par trois de ses dix elfes. A son réveil, il effectue l'une des actions suivantes de manière atomique :

- Si ce sont les rênes qui l'ont réveillé, le Père Noël les harnache, remplit le traîneau de jouets et part ensuite les livrer dans le monde entier. Au retour, il permet aux rennes de se mettre en vacances.
- Si ce sont des elfes qui l'ont réveillé, le Père Noël propose de nouvelles idées de jouets à ceux-ci. Les elfes repartent alors fabriquer des jouets avec leurs collègues.

Les rênes sont prioritaires sur les elfes et le temps du Père Noël est trop précieux pour qu'il ait à gérer lui-même les groupes.

C Réplication

C.5 Enumérer les différents types de défaillance que l'on peut rencontrer dans un système distribué. Pour chacun, donner un exemple concret. *(2 points)*

C.6 Qu'est-ce la consistance stricte ? A quoi peuvent servir les consistances plus faibles ? *(1 point)*

C.7 Parmi les trois scenarios suivants (avec les conventions du cours), lesquels respectent la consistance séquentielle (bien justifier la réponse) *(2 points)*

P_1	$W(x)a$			
P_2		$W(x)b$		
P_3			$R(x)b$	$R(x)a$

P_1	$W(x)a$			
P_2		$W(x)b$		
P_3			$R(x)b$	$R(x)a$
P_4		$R(x)a$		

P_1	$W(x)a$			
P_2		$R(x)a$	$W(x)b$	
P_3			$R(x)b$	$R(x)a$

D Election et RMI

D.8 Rappeler la spécification du problème de l'Election. *(0,5 point)*

D.9 Implémenter en RMI, l'algorithme de LeLann, Chang et Roberts. Rappeler préalablement dans quelle(s) condition(s) celui-ci fonctionne. *(4 points)*

On respectera les consignes suivantes pour représenter l'anneau de taille n :

- $n < 255$ (sans que cette valeur soit utilisable par l'algorithme)
- les machines ont pour adresses 192.168.20.1 192.168.20.254
- la machine de numéro i ($1 \leq i \leq 254$) a pour adresse 192.168.20.i
- le voisin de gauche (resp. de droite) de la machine d'adresse `ip` est la machine d'adresse `ip-1` (resp. `ip+1`).
- on communiquera via deux objets RMI : `"gauche"` et `"droit"` qui représenteront les canals de communication en mode FIFO. Ces objets possédent deux méthodes `envoyer(Object msg)` et `Object recevoir()`.
- Seules les machines correspondantes peuvent utiliser les méthodes précédentes. Par exemple, une machine peut `envoyer` sur le `gauche` de la machine située à sa droite. Celle-ci est la seule à pouvoir faire `recevoir()`.
  Il ne vous est pas demandé d'implémenter syntaxiquement ces interdictions.

On ne gèrera pas les aspects déploiements et on supposera que les identités sont correctement fournies, lors du déploiement, en ligne de commande :

192.168.20.i $ `java LCR id`

E Annexe

Quelques primitives de synchronisation en Java 1.5 :

- `Thread`
  - méthode `run()` : activité
  - méthode `start()` : démarrage de l'activité
  - `static int activeCount()` : renvoie le nombre de threads actuellement exécutés
  - `static int enumerate(Thread[] tarray)` : stocke l'ensemble des threads du même groupe dans le tableau et renvoie le nombre de threads.
  - `static Thread currentThread()` : renvoie le thread en train d'être exécuté. (utile avec `Runnable`)
- `Semaphore`
  - `Semaphore(int tickets)` : déclare un sémaphore `tickets`-aire.
  - `Semaphore(int permits, boolean fair)` : déclare un sémaphore `tickets`-aire, avec attente (presque) FIFO si `fair` est `true`.
  - `void acquire()` : bloque tant que moins de n threads possèdent un "ticket" du sémaphore.
  - `void acquire(int tickets)` : bloque jusqu'à ce que moins de n threads possèdent un "ticket" du sémaphore et consomme `tickets` tickets de sémaphore.
  - `void release()` libère un ticket,
  - `void release(int tickets)` libère `tickets` tickets.
  - `boolean tryAcquire()`
  - `boolean tryAcquire(int tickets)`
  - `boolean tryAcquire(long timeout,TimeUnit unit)`
  - `boolean tryAcquire(int tickets,long timeout,TimeUnit unit)`
  - `int getQueueLength()` renvoie le nombre de threads en attente,

  - `boolean hasQueuedThreads()` renvoie `true` s'il y a des threads en attente.
  - `Collection<Thread> getQueuedThreads()` renvoie la collection des `Thread`s en attente sur ce sémaphore.
- `CyclicBarrier` barrière cyclique
  - `CyclicBarrier(int parties)` : barrière pour `parties` threads,
  - `CyclicBarrier(int parties, Runnable barrierAction)` : barrière pour `parties` threads, `barrierAction` est exécutée avant de lever la barrière.
  - `int await()` : attendre que tous les threads aient atteint la barrière. Renvoie le nombre de threads qu'il fallait attendre.
  - `int await(long timeout, TimeUnit unite)` : idem avec un délai d'attente de `timeout`
- `CountDownLatch` barrière de type "compte à rebours"
  - `CountDownLatch(int n)` crée une barrière avec un compte à rebours commençant à n,
  - `void await()` attendre que le compte à rebours soit terminé,
  - `boolean await(long delai, TimeUnit unit)` attendre au plus $delai$ `unit` que le compte à rebours soit terminé,
  - `void countdown()` décrémente le compteur. Si 0 est atteint, tous les threads bloqués sont alors libérés,
  - `long getCount()` renvoie l'état du compteur.
- `ArrayBlockingQueue<E>` file de taille bornée
  - `ArrayBlockingQueue(int capacite)` construit une file de taile `capacite`,
  - `ArrayBlockingQueue(int capacite, boolean fair)` construit une file de taile `capacite` et équitable : les accès sont FIFO,
  - `void put(E elt)` ajoute `elt` à la file, en attendant si celle-ci est pleine,
  - `E take()` renvoie le premier élément de la file et le retire, en bloquant tant que la file est vide,
  - `E poll(long delai, TimeUnit unit)` renvoie le premier élément de la file et le retire, en attendant au plus $delai$ `unit`,
  - `offer(E elt)` ajoute `elt` à la file, en retournant immédiatement sans ajouter si la file est pleine.
- `LinkedBlockingQueue<E>` file optionnellement bornée
  - `LinkedBlockingQueue()` crée une file d'attente (de taille maximale `Integer.MAX_VALUE`),
  - `LinkedBlockingQueue(int capacite)` crée une file d'attente de taille maximale `capacite` éléments,
  - et méthodes identiques.
- `SynchronousQueue<E>` file de capacité nulle (rendez-vous.
  - `SynchronousQueue()`
  - `SynchronousQueue(boolean fair)`, avec accès FIFO si `fair` est vrai,
  - même méthodes (certaines sont trivialisées).
- `ConcurrentLinkedQueue<E>` file d'attente non bornée, threadsafe et sans attente.
  - `boolean add(E elt)` ajoute `elt` et renvoie `true`,
  - `E poll()` retourne et enlève le premier élément de la file,
  - `E peek()` retourne sans enlèver le premier élément de la file.