Université de Provence
MASTER INFORMATIQUE
Programmation Parallèle
et Distribuée
2008-2009

Programmation Parallèle et Distribuée

Examen – 10 septembre 2009
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.

(2,5 points)

B Synchronisation JAVA

On s'intéresse aux solutions en Java au problème suivant, dit des lecteurs et des rédacteurs. En voici une définition informelle :

Un système distribué est constitué de deux types d'entités : les lecteurs et les rédacteurs. Ceux-ci se partage une structure de données, qui peut être accédée en "lecture" et en "écriture". Comme leur nom l'indique, les lecteurs n'effectuent que des lectures et les rédacteurs uniquement des écritures. Les lectures peuvent être concurrentes, ie simultanées. Par contre, les écritures se font nécessairement par un seul rédacteur à la fois : en exclusion de tout autre processus.

B.3 Proposer une *spécification formelle* du problème des lecteurs et des rédacteurs. Pour chacun des axiomes proposés, préciser son type. *(2 points)*

On considère le code fourni en annexe comme une ébauche de solution. Les corrections/ajouts effectués sont à reporter sur votre copie.

B.4 Compléter la méthode `main` afin qu'elle permette de créer et démarrer les `Lecteurs` et `Redacteurs` en respectant la syntaxe indiquée en commentaire. *(1 points)*

On cherche maintenant à compléter/corriger le code fourni. On appelle *accès* toute demande de lecture ou d'écriture.

B.5 Dans combien d'états, en terme de gestion des accès, peut être la variable `partagee`. Quelles sont les valeurs correspondantes pour les variables `lecteurs` et `redacteur` dans le code ci-joint. *(0,75 point)*

B.6 Qu'est-ce qu'un objet `CountDownLatch` et un objet `Semaphore` ? *(0,75 point)*

B.7 Expliquer informellement le fonctionnement du code donné en annexe. Que se passerait-il si le corps des boucles `while` lignes 37 et 43 était vide ? *(1,5 points)*

B.8 Quel est le principal problème, en terme de programmation concurrente, pour le code fourni en annexe ?

Amender, corriger et compléter ce code pour résoudre le problème des lecteurs et des rédacteurs. *(3 points)*

C Réplication

C.9 Rappeler la définition de la consistance séquentielle. Parmi les trois scenarios suivants (avec les conventions de notation 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	$W(x)b$			
P_2			$R(x)a$	$R(x)b$
P_3		$W(x)a$		
P_4		$R(x)b$		

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

D Election et RMI

D.10 Rappeler la spécification du problème de l'Election. *(0,5 point)*

D.11 Rappeler quels sont les principaux paramètres à considérer lorsque l'on conçoit un algorithme distribué (on distinguera notamment les aspects statiques, dynamiques et de terminaison). *(1,5 points)*

D.12 Implémenter en RMI, l'algorithme de LeLann, Chang et Roberts. Rappeler préalablement dans quelle(s) condition(s) celui-ci fonctionne. *(3,5 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.id $ `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.