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

Programmation Parallèle et Distribuée

Examen – mercredi 10 juin 2009
3h00

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

A Synchronisation JAVA

A.1 Qu'est-ce qu'une structure de données *thread-safe* ? Donner, en Java, un exemple d'objet thread-safe et un exemple d'objet non thread-safe. *(1 point)*

A.2 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 Producteurs et Consommateurs

B.3 Proposer une définition *formelle* du "problème des producteurs et consommateurs" (bien spécifier les "agents" et les structures de données en jeu). *(1,5 points)*

B.4 En vous appuyant sur le code fourni en annexe, proposer deux solutions `FileABQ` et `FileALS` au problème producteurs / consommateurs implémentant l'interface `FileDAttente`. La première devra utiliser `ArrayBlockingQueue`, la seconde `ArrayList` et `Semaphore`. *(4,5 points)*

B.5 Connaissez-vous d'autre(s) structure(s) de données utile(s) pour implémenter une solution à ce problème ? Les comparer brièvement (y compris avec celles de l'exercice précédent). *(1 points)*

C Réplication

C.6 Enumérer les différents types de défaillance que l'on peut rencontrer dans un système distribué. Pour chacun, donner un exemple concret. *(2 points)*

C.7 Qu'est-ce que la consistance stricte ? A quoi peuvent servir les consistances plus faibles ? *(1 point)*

C.8 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 Consensus

On considère deux processus P_1 et P_2 synchrones et communiquant par messages. On cherche à résoudre le problème du consensus binaire pour différents types de perte de messages.

D.9 Rappeler la définition exacte du problème du Consensus. *(1,5 points)*

D.10 Pour chacun des cas suivant, indiquer s'il existe un algorithme de consensus. Dans le cas positif, donner en pseudo-code cet algorithme, et préciser sa complexité en temps. Dans le cas négatif donner une preuve d'impossibilité.

1. tous les messages parviennent à destination,
2. seuls les messages émis par le processus P_1 peuvent éventuellement être perdus,
3. sur toute l'éxécution, si des messages sont perdus, ce sont toujours ceux émis par le même processus
4. lors d'une ronde donnée, au plus l'un des deux messages peut éventuellement être perdu,
5. les messages provenant des deux processus peuvent être perdus (éventuellement en même temps).

(3,5 points)

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 `Threads` 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.