Examen SE 205

novembre 2017

1h30 - Barème indicatif, **sans document**
Les téléphones portables doivent être éteints et rangés dans vos sacs.
Les calculatrices sont interdites.

Les différentes parties de l'examen sont indépendantes.

Il sera tenu compte de la présentation et de la clarté dans la rédaction.
Seules les réponses précises et justifiées seront considérées

1 Modèle d'exécution C/POSIX (3 points)

Nous disposons d'une structure de tampon non-protégé et ses fonctions ainsi qu'une structure de tampon protégé contre les accès concurrents:

```
typedef struct {
  int first, last, size, max_size;
  void ** buffer;
} buffer_t;

// Allocate and initialize buffer structure
buffer_t * buffer_init(int size);

// Remove and return an element from
// buffer. When empty, return NULL.
void * buffer_get(buffer_t * b);
```

```
// Append an element into buffer
// When full, return 0. Othewise, return 1.
int buffer_put(buffer_t * b, void * d);

int buffer_size(buffer_t * b);

typedef struct {
  buffer_t          * buffer;
  pthread_mutex_t   * mutex;
  pthread_cond_t    * guard;
} protected_buffer_t ;
```

Nous implantons des fonctions manipulant protected_buffer_t telle que protected_buffer_put qui dépose un élément dans le tampon et bloque si celui-ci est vide. Nous implantons une fonction protected_buffer_remove qui retire et retourne un élément du tampon et retourne NULL si le tampon est vide. Complétez le code en prenant en compte les accès concurrents et les autres opérations (bloquantes) :

```
// Extract and return an element.
// If this is not possible immediately, return NULL.

void * protected_buffer_remove(protected_buffer_t * b){
  void * d;

  d = buffer_get(b->buffer);

  return d;
}
// NE PAS REPONDRE SUR LA FEUILLE D'ENONCE
```

2 Modèle d'exécution Java natif (3 points)

Nous avons développé une classe Java de tampon protégé contre les accès concurrents qui offre des méthodes pour ajouter et retirer des éléments suivant plusieurs sémantiques bloquante, non-bloquante et temporisée. L'implantation de la méthode poll qui retire un élément du tampon avant une échéance *deadline* se trouve rappelée ci-dessous. Indiquez et détaillez les limitations de cette implantation en terme d'exécution temporelle et donnez des exemples illustrant ces limitations.

```
Object poll(long deadline) {
   long    timeout;
   synchronized(this) {
      while (size == 0)
          try {
              timeout=deadline - System.currentTimeMillis();
              wait(timeout);
              break ;
          } catch (InterruptedException e){} ;
      if (size==0) return null;
      if (size == maxSize) notifyAll();
      return super.get();
   }
}
```

3 Modèle d'exécution alternatif (3 points)

Nous souhaitons concevoir un système qui reçoit et traite des requêtes en tirant profit d'une architecture multi-cœurs. Nous utilisons un patron Executor comme celui vu en cours et en TP. On dispose d'un octo-cœurs (8) et souhaitons que le système accepte 1) de traiter en parallèle et sans attente au minimum 8 requêtes concurrentes, 2) de traiter en parallèle au plus 16 requêtes simultanées sans en mettre en attente et 3) de détruire les tâches créées après 10s d'inactivité. Détaillez la structure (notamment les patrons intermédiaires) et le fonctionnement du patron Executor. Précisez la configuration de ses principaux paramètres pour le cas décrit.

4 Read and Write Locks (4 points)

Nous fournissons l'implantation d'un verrou lecteur / écrivain. Plusieurs threads peuvent accéder au verrou en lecture si aucun thread ne possède le verrou en écriture. Si un thread dispose du verrou en écriture, aucun thread ne peut prendre le verrou que ce soit en lecture ou en écriture.

Expliquez l'utilité de l'attribut writeRequests et donnez un scenario problématique ainsi évité. Expliquez pourquoi ce verrou n'est pas réentrant et donnez un scenario faisant apparaître le problème.

```
int readers = 0;
int writers  = 0;
int writeRequests = 0;

synchronized void lockRead(){
  while (writers > 0 || writeRequests > 0)
    wait();
  readers++;
}

synchronized void unlockRead(){
  readers--;
  notifyAll();
}
```

```
synchronized void lockWrite(){
  writeRequests++;
  while (readers > 0 || writers > 0)
    wait();
  writeRequests--;
  writers++;
}

synchronized void unlockWrite(){
  writers--;
  notifyAll();
}
```

REPONDRE AUX QUESTIONS SUIVANTES SUR UNE FEUILLE SEPAREE

N'OUBLIEZ PAS D'Y INSCRIRE VOS NOM ET PRENOM.

VOUS POUVEZ REPONDRE EN FRANCAIS

5 Safety and Liveness (4 points)

A property is a set of histories. Consider histories in which processes can propose values in {0,1}, perform steps of computations, and output values in {*commit, abort*}. We assume that in a history, every process proposes a value *at most once*, outputs a value *at most once*, and only if it previously proposed a value.

Classify the following three properties into safety/liveness. If a property is an intersection of the two, specify the corresponding safety and liveness properties. Justify your answers. Make no mistake about the meaning of "eventually": it does not mean "possibly" but "there is a time after which".

1. Every process eventually outputs a value.
2. If every process proposes 1 and no process crashes (stops taking steps), then no process can output *abort.*
3. Eventually, all processes output the same value.

6 Linked List (3 points)

In the validate function of the lazy linked-list implementation below, is checking curr.marked really necessary? Justify your answer.

Lazy synchronization: logical removals and wait-free contains

```
private boolean validate(Node pred, Node
    curr) {

  return !pred.marked && !curr.marked &&
           pred.next==curr;
}
```

- remove first marks the node for deletion and then physically removes it
- contains returns true iff the node is reachable and not marked
- A node is in the set iff it is an unmarked reachable node

```
public boolean remove(int item)
   while (true){
      Node pred=head;
      Node curr=pred.next;
      while (curr.key<item){
         pred=curr;
         curr=curr.next;
      }
      pred.lock();
      try {
         curr.lock();
         try {
          if (validate(pred,curr)){
          if (curr.key!=item){
              return false;}
          curr.marked=true;
          pred.next=curr.next;
          return true;}
         } finally{
             curr.unlock(); }
      } finally{
        pred.unlock();}
   }
}
```


Lazy synchronization: wait-free contains

```
public boolean insert(int item){
   while (true){
      Node pred=head;
      Node curr=pred.next;
      while (curr.key<item){
         pred=curr;
         curr=curr.next;
      }
      pred.lock();
      try {
         curr.lock();
         try {
          if (validate(pred,curr)){
           if (curr.key==item) {
               return false;
             }
          Node node = new Node(item);
          node.next=curr;
          pred.next=node;
          return true;
         } finally{
             curr.unlock(); }
      } finally{
        pred.unlock();}
     } }
}
```

```
public boolean contains(int item){

   Node curr=head;
   while (curr.key<item){
      curr=curr.next;
   }
   return (curr.key==item)&& !curr.marked ;
}
```