Programmation réseau et concurrente

Benoît Barbot

Département informatique, Université Paris-Est Créteil, M1

Mardi 2 février 2016, Cours 3 : Concurrence

Plan

Client

```
public class Client {
  private SocketChannel sc;
  private boolean connected = false;
  public boolean isConnected(){ return connected;}
  public void disconnect() { connected = false;}

  public Client( String addr, int port ) throws Exception {
    InetSocketAddress isa=new InetSocketAddress(addr, port);
    if(isa.isUnresolved())throw
      new Exception("Fail to resolve address:"+addr);
    sc = SocketChannel.open();
    sc.configureBlocking(true);
    sc.connect(isa);
    connected=true;
  }
...
}
```

RepeatNetwork

```
class RepeatNetwork implements Runnable {
  private SocketChannel sc;
  private ByteBuffer buff;
  private Client main;
  public RepeatNetwork(SocketChannel sc, Client main){
    this.sc =sc; this.main = main;
    buff = ByteBuffer.allocateDirect(1024);
  }
  public void run() {
    try {
      while (main.isConnected()) {
        buff.clear();
        if (sc.read(buff) == -1)
          throw new IOException("Connection␣Close");
        buff.flip();
        CharBuffer cb=Charset.forName("UTF-8").decode(buff);
        System.out.println(cb.toString());
      }
    } catch (IOException e) {
      main.disconnected();
    };
  }
}
```

RepeatKeyboard

```
class RepeatKeyboard implements Runnable {
  private SocketChannel sc;
  private ReadableByteChannel keyboard;
  private ByteBuffer buff = ByteBuffer.allocateDirect(1024);
  private Client main;
  public RepeatKeyboard(SocketChannel sc, Client main){
    this.sc =sc; this.main = main;
    keyboard = Channels.newChannel(System.in);
  }
  public void run() {
    try {  while (main.isConnected()) {
        buff.clear();
        int n = keyboard.read(buff);
        if (n == -1)
        throw new IOException("User quit");
        buff.flip();
        CharBuffer c = Charset.defaultCharset().decode(buff);
        ByteBuffer b = Charset.forName("UTF-8").encode(c);
        while (b.hasRemaining()) { sc.write(b); }
      } }catch(IOException e){
      main.disconnected();
    };
  }
}
```

Client2

```
public class Client {
 ...
 public void startThreads() throws IOException{
   Thread rnt = new Thread(new RepeatNetwork(sc, this));
   Thread rkt = new Thread(new RepeatKeyboard(sc, this));
   rnt.start();
   rkt.start();
   try {  rnt.join();  rkt.join();
   } catch (InterruptedException e) {}
 }
}
```

Plan

Sérialisation automatique en Java

Caractéristique

- Automatique
- Gestion des objets nulle
- Gestion des structures récursives circulairement
- Binaire
- Simple d'utilisation

Sérialisation automatique en Java

Caractéristique

- Automatique
- Gestion des objets nulle
- Gestion des structures récursives circulairement
- Binaire
- Simple d'utilisation

Défault

- Lent (à cause de l'introspection)
- Lourd pour un format binaire
- Très sensible aux modifications des classes

Examples

Serialisation

```
public class TestSerializable implements Serializable {
    int i;
    String s;
    ...
}
public static void main(String[] argv) throws IOException {
  out.print("Serialize:'");
  TestSerializable ts = new TestSerializable(32,"foo");
}
```

Examples

Serialisation

```
public class TestSerializable implements Serializable {
    int i;
    String s;
    ...
}
public static void main(String[] argv) throws IOException {
  out.print("Serialize:'");
   TestSerializable ts = new TestSerializable(32,"foo");
}
```

Contenu de la sérialisation.

```
.....  .sr  .TestSerializable...,
...Z<.....  .I  .iL  .st  .Ljava/
lang/String;xp     t  .foo
```

Sensibilité aux changements exemple

Classe précédente

1. Serialize => Fichier
2. Ajout de `toString()`
3. Fichier => Deserialize

Sensibilité aux changements exemple

Classe précédente

1. Serialize => Fichier
2. Ajout de toString()
3. Fichier => Deserialize

Résultat

```
Exception in thread "main" java.io.InvalidClassException:
  TestSerialization; local class incompatible:
  stream classdesc serialVersionUID = -2203064095764525383,
  local class serialVersionUID = -1307691225281433769
  at ...
  at Main.main(Main.java:41)
```

serialVersionUID

Ajouter à la classe

```
private static final long
  serialVersionUID = -1307691225281433769L;
```

serialVersionUID

Ajouter à la classe

```
private static final long
  serialVersionUID = -1307691225281433769L;
```

Utilisations

- Généré un nombre aléatoire à la création
- Le régénérer lorsque la structure change
- Champs retirés ignorés
- Champs ajoutés non initialisés

Structure plus complexe

Conteneur

Tous les membres doivent être Serialisable

$\Rightarrow$ java . io . NotSerializableException

Structure plus complexe

Conteneur

Tous les membres doivent être `Serialisable`

$\Rightarrow$ `java.io.NotSerializableException`

Hiérarchie

Sous classe peut-être `Serialisable` ,
même si classe de base non serialisable $\Rightarrow$ Constructeur vide pour la classe de base

Structure plus complexe

Conteneur

Tous les membres doivent être Serialisable

$\Rightarrow$ java.io.NotSerializableException

Hiérarchie

Sous classe peut-être Serialisable ,
même si classe de base non serialisable $\Rightarrow$ Constructeur vide pour la classe de base

Contrôle fin

```
private void writeObject(java.io.ObjectOutputStream out)
  throws IOException
private void readObject(java.io.ObjectInputStream in)
  throws IOException, ClassNotFoundException;
private void readObjectNoData()
  throws ObjectStreamException;
```

Rappel Exemple

```
public class TestThread implements Runnable {
  public static int t = 0;
  public void run() {
    for(int i=0; i< 10000; i++){
      t = t + 1;
    }
  }
  public static void main(String[] args) throws InterruptedEx
    Thread t1 = new Thread(new TestThread());
    Thread t2 = new Thread(new TestThread());
    t1.start();
    t2.start();

    t1.join();
    t2.join();

    System.out.println("Value of t:"+ t );
  }
}
```

Thread rappel

Accès concurrent aux variables

1. Interlacement des threads non déterministe
2. Nécessité de protéger les accès concurrents

Thread rappel

Accès concurrent aux variables

1. Interlacement des threads non déterministe
2. Nécessité de protéger les accès concurrents

Problème courant de “race condition”

1. Compteur
2. Accès concurrents à un tableau
3. Accès concurrents à une liste
4. ...

Méthodes de synchronisation

Lock / Mutex

Un seul Thread possède le lock à la fois

Sémaphore

N Threads utilisent le sémaphore à la fois

fence / barrière de synchronisation

N threads doivent avoir atteint le même point pour progresser

Exclusion mutuelle

Garantir un seul thread dans une zone

- Délimité une zone
- Implémenter un processus d'exclusion

Exclusion mutuelle

Garantir un seul thread dans une zone

- Délimité une zone
- Implémenter un processus d'exclusion

```
for(int i=0; i< 10000; i++){
  // Entree zone d'exclusion
  t = t + 1;
  // Sortie zone d'exclusion
}
```

Propriété pour les Mutex

Définition du système

N threads, v variables partagées

```
while(true){
  lock(v);
  // Section critique
  unlock(v);
  // Section non critique
}
```

Chaque thread reste un temps fini en section critique

Propriété pour les Mutex

Définition du système

N threads, v variables partagées

```
while(true){
  lock(v);
  // Section critique
  unlock(v);
  // Section non critique
}
```

Chaque thread reste un temps fini en section critique

Exclusion mutuelle

Un seul thread dans la section critique

Propriété pour les Mutex

Définition du système

N threads, v variables partagées

```
while(true){
  lock(v);
  // Section critique
  unlock(v);
  // Section non critique
}
```

Chaque thread reste un temps fini en section critique

Exclusion mutuelle

Un seul thread dans la section critique

Absence d'inter blocage

Si deux threads exécutent lock l'un des deux l'obtient.

Propriété pour les Mutex 2

Pas de famine

Un thread ne peut pas attendre le lock un indéfiniment

Propriété pour les Mutex 2

Pas de famine

Un thread ne peut pas attendre le lock un indéfiniment

Attente bornée

Si un thread T, demande le lock, le nombre de thread pouvant obtenir le lock avant T est bornée.

Propriété pour les Mutex 2

Pas de famine

Un thread ne peut pas attendre le lock un indéfiniment

Attente bornée

Si un thread T, demande le lock, le nombre de thread pouvant obtenir le lock avant T est bornée.

Pas de privilège

Les thread sont traités équitablement

Méthodes - Systèmes

pthread_mutex_t

- Peut générer un appel système
- Lent
- Attente inactive
- Repose sur d'autres systèmes d'exclusion
- Fonctionne entre processus

Méthodes - Systèmes

pthread_mutex_t

- Peut générer un appel système
- Lent
- Attente inactive
- Repose sur d'autres systèmes d'exclusion
- Fonctionne entre processus

Propriété

dépend de l'implémentions sous-jacente,
Exclusion mutuelle, pas d'inter blocage

Méthodes - Matérielle

Instruction CPU

Instruction processeur atomique test_and_set(v){v2=v; v=1; **return** v2}

- Rapide
- Attente active
- Uniquement Mutex

Méthodes - Matérielle

Instruction CPU

Instruction processeur atomique test_and_set(v){v2=v; v=1; **return** v2}

- Rapide
- Attente active
- Uniquement Mutex

Utilisation

```
void lock(v){
  while( test_and_set(v){};
}
void unlock(v){
  v=0;
}
```

Méthodes - Matérielle

Instruction CPU

Instruction processeur atomique test_and_set(v){v2=v; v=1; **return** v2}

- Rapide
- Attente active
- Uniquement Mutex

Utilisation

```
void lock(v){
  while( test_and_set(v){};
}
void unlock(v){
  v=0;
}
```

propriété

Exclusion mutuelle, pas d'inter blocage

Méthodes - Logicielles

Algorithme garantissant l'exclusion mutuelle

1. Construit à partir d'instruction classique
2. Souvent une attente active
3. portable entre les langages

propriété

Dépends de l'algorithme utilisé

Algorithme de Dekker

```
1  boolean[] wants_to_enter = { false, false }
2  int turn = 0;
3
4  void run(i){ // i=0 v i=1
5    while(true){
6      wants_to_enter[i] = true;
7      while(wants_to_enter[1 -i]) {     //WE
8        if(turn != i) {                  //WE
9          wants_to_enter[i] = false;    //WE
10         while(turn != i){};
11         wants_to_enter[i] = true;
12       }
13     }
14     // section critique                //WE
15     turn = 1-i;                       //WE
16     wants_to_enter[i] = false;        //WE
17     // section non critique
18   }
19 }
```

Preuve des propriétés

Méthodes automatiques

Explorer l'espace d'états : (PC, variables)

Preuve des propriétés

Méthodes automatiques

Explorer l'espace d'états : (PC, variables)

Raisonnement sur des invariants

...

Problèmes d'optimisation

```
public class Dekker implements Runnable {
    static volatile boolean wte0 = false;
    static volatile boolean wte1 = false;
    static volatile int turn = 0;
    static volatile int t =0;
    int i;
    public Dekker(int i ){this.i=i;}
...
    public static void main(String[] strings) throws
      InterruptedException {
        Thread t0 = new Thread(new Dekker(0));
        Thread t1 = new Thread(new Dekker(1));
        t0.start();
        t1.start();
        t0.join();
        t1.join();
        System.out.println(t);
    }
}
```

Problèmes d'optimisation

```
public void run(){ // i=0 v i=1
        int j =0;
        while( j < 10000000){
            if(i==0){wte0=true;}else{wte1=true;};
            while((i==0?wte1:wte0)) {
                if(turn != i) {
                    if(i==0){wte0=false;}else{wte1=false;};
                    while(turn != i){};
                    if(i==0){wte0=true;}else{wte1=true;};
                }
            }
            // section critique
            t = t + 1;
            turn = 1-i;
            if(i==0){wte0=false;}else{wte1=false;};
            // section non critique
            j++;
        }
    }
```