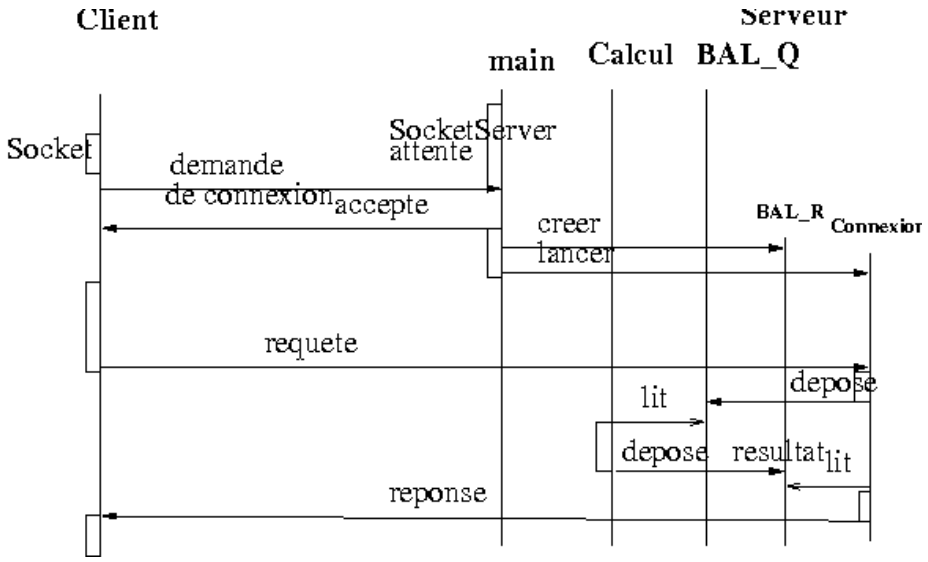
I.U.T de Montpellier **2ème année**
Département Informatique **Durée 2h00**
Aucun documents

Programmation répartie

Client / serveur

Dans une application client/serveur, le serveur reçoit des requêtes de plusieurs clients. Pour cela, il utilise une boite à lettres séparée pour chacun de ces clients ou les requêtes du client sont déposées. Les requêtes peuvent être arrivées en rafale : plusieurs requêtes d'un même client avant que les requêtes précédentes soient terminées.

* **Le serveur** fonctionne de la façon suivante. La tâche principale (*main*) s'occupe uniquement de l'accueil des demandes de communication. Dès le lancement du serveur, une deuxième tâche *Calcul* tourne également pour que les calculs demandés soient faites indépendamment du *main*. Une boite à lettres *BAL_Q* est aussi créée où les requêtes de calcul pour solliciter la tâche *Calcul* doivent être déposées. Quand une demande de connexion arrive à la port énoncée du serveur, une tâche de type *Connexion* pour gérer la communication avec le client par socket et une boite à lettres *BAL_R[i]* dédiée aux réponses à ce client *i* doivent être créées. Les clients sont identifiés par une indice *i*. La tâche *Connexion[i]* existe jusqu'à la déconnexion du client i (dernier message ''stop'' qui arrive du client). La communication entre le client et la tâche *Connexion[i]* du serveur se passe par un socket TCP. Quand la tâche *Connexion[i]* détecte un message provenant du client (via le socket), elle dépose la requête reçue dans la boite à lettres *BAL_Q* en mettant son identifiant *i* dans l'en-tête du message. C'est la tâche *Calcul* qui doit prendre en charge les requêtes dans cette boite *BAL_Q*. Elle exécute les requêtes lues et quand l'exécution d'une requête est terminée *Calcul* dépose la réponse dans *BAL_R[i]*, lit la requête suivante de sa boite *BAL_Q*, s'il y en a et l'exécute, etc.



* **La classe Java Bal** (dont les instances sont BAL_Q, BAL_R[1], ... BAL_R[i]...) doit implémenter une boite circulaire avec 25 places pour des requêtes de type String. Deux méthodes permettent de gérer les messages :

void deposer(String s) - qui dépose le message s à la place suivante dans la boite circulaire. Si pas de place dans la boite, la méthode doit être *bloquante* en attendant la lecture d'un message déjà déposé.

String lire() - qui permet de lire et retourner le message le plus ancien non encore lu de la boite circulaire. Si pas de message non encore lu dans la boite, la méthode doit être *bloquante* en attendant l'écriture d'un nouveau message.

Q1 : Une boite à lettres (Bal) peut être accédée par plusieurs utilisateurs concurrentiels. Quelles sont des anomalies possibles et quelles solutions pouvez-vous envisager pour les éviter ? (2 points)

Q2 : Ecrivez la classe *Bal*. La solution doit garantir l'utilisation correcte en mode concurrentiel. (4 points)

*** La tâche *Connexion*** s'occupe d'un client. Elle attend les requêtes du client par un socket connu par son constructeur. Elle dépose la requête dans la boite *BAL_Q* pour la tâche *Calcul*. Elle reçoit la réponse dans sa boite à lettre de réponse de *Calcul*. Pour simplifier, supposons que les boites à lettres sont des variables statiques du *Serveur*. Par exemple :

```
static Bal BAL_Q = new Bal();                 // Bal pour les requetes de calculs
static Bal BAL_R[] = new Bal[50];             // Bals pour les réponses
```

Pour récupérer les entrées/sorties d'un socket *sockcli* vers le client dans cette classe :
- on propose deux variables :

```
private BufferedReader ins ;
private PrintWriter outs ;
```

- et vous pouvez utiliser les méthodes suivantes :

```
private  BufferedReader creer_inp(socket sockcli){
  BufferedReader inbr = new BufferedReader (
                                        new InputStreamReader(
                                            sockcli.getInputStream()) );
        return inbr ;
    }
private  PrintWriter creer_outp(socket sockcli){
    PrintWriter outbr = new PrintWriter( new BufferedWriter(
                new OutputStreamWriter(soc.getOutputStream())), true);
        return outbr ;
    }
```

Q3 : Ecrivez la classe *Connexion* nécessaire telle que les tâches *Connexion* soient correctement lancées par la solution proposée dans le main (voir plus loin). (4 points)

*** La tâche principale *main*** initialise le serveur.

Q4 : Donnez le pseudo-code (en un langage de description) du *main* du serveur (2 points)

Q5 : Continuez à écrire la tâche main suivante en java. On suppose que les prototypes des classes *Connexion* et *Calcul* sont disponibles. Les tâches doivent être gérées par un *ExecutorService* suffisamment grand. (4 points)

```
public class Serveur {
```

```
static Bal BAL_Q = new Bal();                  // Bal pour les requetes de calculs
static Bal BAL_R[] = new Bal[50];              // Bals pour les réponses

static void initBR() {
  for (int i =0 ; i < 50 ; i++)
    Serveur.BAL_R[i] = new Bal();
 }

  public static void main(String[] args) throws Exception {
        int nbcl =0;                        // numero de clients (max 50)
        Serveur.initBR();
...
```

// écrire la suite de l'algorithme du main

ANNEXE

public interface Executor

It executes submitted Runnable tasks. Submission is with **execute(Runnable R)** method.

```
Example :  Executor executor1 = anExecutor;
                executor1.execute(new RunnableTask1());
```

public interface ExecutorService extends Executor

New methods :

boolean awaitTermination(long timeout, TimeUnit unit)
Blocks until all tasks have completed execution after a shutdown request, or the timeout occurs, or the current thread is interrupted, whichever happens first.

boolean isShutdown()
Returns true if this executor has been shut down.

boolean isTerminated()
Returns true if all tasks have completed following shut down.

void shutdown()
Initiates an orderly shutdown in which previously submitted tasks are executed, but no new tasks will be accepted.

List<Runnable> shutdownNow()
Attempts to stop all actively executing tasks, halts the processing of waiting tasks, and returns a list of the tasks that were awaiting execution.

public class Executors

static ExecutorService newCachedThreadPool()
Creates a thread pool that creates new threads as needed, but will reuse previously constructed threads when they are available.

static ExecutorService newFixedThreadPool(int nThreads)
Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue.

static ExecutorService newFixedThreadPool(int nThreads, ThreadFactory threadFactory)
Creates a thread pool that reuses a fixed number of threads operating off a shared unbounded queue, using the provided ThreadFactory to create new threads when needed.

static ScheduledExecutorService newScheduledThreadPool(int corePoolSize)
Creates a thread pool that can schedule commands to run after a given delay, or to execute periodically.

```
static ExecutorService  newSingleThreadExecutor()
        Creates an Executor that uses a single worker thread operating off an
unbounded queue.
static ScheduledExecutorService    newSingleThreadScheduledExecutor()
        Creates a single-threaded executor that can schedule commands to run
after a given delay, or to execute periodically.
static Callable<Object>      callable(Runnable task)
        Returns a Callable object that, when called, runs the given task and
returns null.
static <T> Callable<T>  callable(Runnable task, T result)
        Returns a Callable object that, when called, runs the given task and
returns the given result.
```

Class Semaphore {

```
         Semaphore(int permits)
            //Creates a Semaphore with the given number of permits and nonfair
fairness setting.
         Semaphore(int permits, boolean fair)
           //Creates a Semaphore with the given number of permits and the given
fairness setting.
...
 void       acquire()
             //Acquires a permit from this semaphore, blocking until one is
available, or the thread is interrupted.
 void       acquire(int permits)
           //Acquires the given number of permits from this semaphore, blocking
until all are available, or the thread is interrupted.
 void       release()
         //Releases a permit, returning it to the semaphore.
 void       release(int permits)
             //Releases the given number of permits, returning them to the
semaphore.
 boolean tryAcquire()
           //Acquires a permit from this semaphore, only if one is available at
the time of invocation.
 boolean tryAcquire(int permits)
            //Acquires the given number of permits from this semaphore, only if
all are available at the time of invocation.
...
}
```

Synchronization coopérative

Les méthodes suivantes permettent de synchroniser des threads. Elles sont définies dans la classe `Object` :

- **`wait()`** lève le verrou sur l'objet}et bloque le thread appelant jusqu'à ce qu'une méthode `notify()` ou `notifyAll()` ne le réveille. Une interruption ou un timeout peut aussi conduire au déblocage du thread. Une fois débloqué, il est mis en attente pour l'accès exclusif à la section critique. Il passe donc en mode prêt.
- **`notify()`** réveille un thread alors bloqué par un `wait()`. Si aucun thread n'est bloqué, rien ne se passe. Le thread débloqué doit alors prendre le verrou dès qu'il le pourra.
- **`notifyAll()`** réveille tous les threads bloqués. C'est le thread qui prend en premier le verrou qui accède à la section critique.

---

public class ServerSocket {

ServerSocket(int port) // Creates a server socket, bound to the specified local port.
ServerSocket(int port, int backlog)
// Creates a server socket with the specified local port number and backlog.
ServerSocket(int port, int backlog, InetAddress bindAddr)
//Create a server with the specified port, listen backlog, and local IP address to bind to.
Socket accept() // Listens for a connection to be made to this socket and accepts it.
void close() //Closes this socket.
...
}

---

public class Socket {

Socket(InetAddress address, int port) //Creates a socket and connects it to the specified port and IP address.
Socket(InetAddress address, int port, InetAddress localAddr, int localPort)
// Creates a socket and connects it to the specified remote address on the specified remote port.
Socket(String host, int port)
// Creates a stream socket and connects it to the specified port number on the named host.
Socket(String host, int port, InetAddress localAddr, int localPort)
// Creates a socket and connects it to the specified remote host on the specified remote port.
InputStream getInputStream() // Returns an input stream for this socket.
OutputStream getOutputStream() // Returns an output stream for this socket.
void close() // Closes this socket.
...
}

---

public class PrintWriter

void close()
Closes the stream and releases any system resources associated with it.
...
void print(Object obj)
Prints an object.
void print(String s)
Prints a string.
void println(Object x)
Prints an Object and then terminates the line.
void println(String x)
Prints a String and then terminates the line.

---

public class BufferedReader

void close()
Closes the stream and releases any system resources associated with it.
...
int read(char[] cbuf, int off, int len)
Reads characters into a portion of an array.
String readLine()
Reads a line of text