Sistemas Distribuídos
Teste

9 de janeiro de 2017

10/01/2018

II

1.

```
public class Temperaturas implements Controlador{

    private int temperatura = 0;
    private int limite = 0;
    private boolean estado = false; // on -> true ; false -> off

    private ReentrantLock lock = new ReentrantLock();
    private Condition cond = lock.newCondition();

    public void temperatura (int centigrados){
        lock.lock();

            temperatura = centigrados;
            boolean estadoAntigo = estado;
            estado = (temperatura < limite);
            if(estadoAntigo != estado){
                cond.signalAll();
            }

        lock.unlock();
    }

    public void limiar(int centigrados){
        lock.lock();

            limite = centigrados;
            boolean estadoAntigo = estado;
```

```
        estado = (temperatura < limite);
        if(estadoAnterior != estado){
            cond.signalAll();
        }
    lock.unlock();
}

public boolean on_off(boolean estadoAtual){
    lock.lock();
    boolean res = !(estado == estadoAtual);
    while(estado == estadoAtual){
        cond.await();
    }
    lock.unlock();
    return res;
}
```

2.

```
public class TrataCliente implements Runnable{
    private boolean ant = false;
    private Temperatura3 temp;
    private Socket s;
public void run(){

try{
    OOS out = new OOS(s.getOS());
    OIS in = new OIS(s.getIS());

    while(true){
        OpsTemp op;
        op = (OpsTemp) in.readObject();
```

Ao aqui podia ser igual ao estado atual de Temp.

```
public enum OpsTemp {
    Temperatura,
    Limiar,
    On_Off;
}
```

Controlo sobre operações { Feito, Erro } → Mudou, Não_Mudou;

```
switch(op){
    case Temperatura:
        int aux = in.readInt();
        temp.temperatura(aux);
        out.writeObject(OpsTemp.Feito);
        out.flush();
        break;

    case Limiar:
        int aux = in.readInt();
        temp.limiar(aux);
        out.writeObject(OpsTemp.Feito);
        out.flush();
        break;

    case On_Off:                    ← Aqui podia perguntar o estado, mas vou assumir o último!
        boolean res = temp.on_off(ant);

        if(res){
            out.writeObject(OpsTemp.Mudou);
        }
        else{
            out.writeObject(OpsTem.Nao_Mudou);
        }
        ant = !ant;
        out.flush();
        break;

    default: out.writeObject(OpsTem.Erro);
             out.flush();
}
}catch(Exception e){
        e.printStackTrace();
    }
finally{ s.shutdownOutput();
         s.shutdownInput();
       } s.close();
}
}
```

```
public class Servidor {
    public static void main() {
        Temperaturas temp = new Temperaturas();
        ServerSocket ss = new ServerSocket(9999);
        while(true) {
            Socket es = ss.accept();
            (new Thread(new TrataCliente(es, temp))).start();
        }
    }
}
```

Aqui podia passar o estado atual de temp.

I

1. O objectivo das primitivas lock/unlock é garantir que apenas 1 processo se encontra na zona envolvida por estas primitivas. O objectivo das primitivas wait/notify é que os processos voluntariamente adormeçam enquanto um predicado não é verificado sendo acordados por outros processos quando esses alteram algo e existe a possibilidade de o predicado ser valido. (2)

A forma de utilização de lock e unlock é basicamente adquirir o lock (lock), fazer o que é necessário e depois libertar (unlock). A forma de utilização de wait/notify está normalmente associada a ter um lock para verificar um predicado e caso ñ se verifique liberta-se o lock, adormecendo (wait) sendo que quando for acordado terá de adquirir o lock. Quando "passa" o predicado liberta o lock. Depois de realizar o que tem de realizar adquire novamente o lock, pois vai mexer em algo partilhado e consoante aquilo que foi pode notificar quem está à espera naquela condição libertando o lock de seguida.

2. Na minha opinião, na maior parte das aplicações é importante ter uma arquitetura baseada em camadas, para poder "separar" um pouco as diferentes partes da mesma (ex: interface, negócio e dados) e que estas sejam independentes pelo que acho que esta arquitetura seria uma boa opção. Depois para formar as camadas (dos dados principalmente) bastante estável obrigaria uma arquitetura

baseada em objectos. Devido às características temporais dos dados nas redes sociais e, ao facto dos sistemas terem de ser fortemente assíncronos obrigaria também uma arquitetura baseada em eventos.

3. A relevância do sistema operativo na resolução de problemas de exclusão mútua no modelo de memória partilhada é pelo facto de ao invocar a operação lock (e caso não possa obter) o SO passa o processo de escalonado para bloqueado associado à lista daquele lock (sendo que posteriormente será escalonado) evitando assim esperas ativas.

No modelo de passagem de mensagens (ou seja em SD) o SO é importante nisto que quando um processo quer aceder à sua zona crítica envia mensagem (1 ou várias) e em vez de esperar pela resposta ou prossegue (outro processamento) ou então o SO passa-o de escalonado a bloqueado (associado às mensagens) sendo que quando recebe uma mensagem (resposta) volta a estar escalonada e verifica se já pode aceder (já recebeu todas as mensagens) e caso não passa volta ao estado bloqueado, evitando esperas ativas.

Em ambos os casos evita esperas ativas.

Podia falar do coordenador.