Mestrado Integrado em Engenharia de Telecomunicações e Informática

Sistemas Distribuídos

Testes Resolvidos

David José Ressurreição Alves - A79625

Índice

1. Teste 1

Explique como funcionam genericamente as operações de wait(cond, lock) e signal(cond) ao nível do sistema operativo

R: As operações de wait(cond,lock) e signal(cond), são primitivas associadas a problemas onde exista cooperação e dependência entre processos, podendo um determinado processo ficar impedido de prosseguir com a sua execução enquanto um outro não realizar determinada ação. Neste contexto, a operação wait() permite a um processo se bloquear voluntariamente numa variável de condição. A nível do sistema operativo, o que acontece é que o processo é bloqueado(adormecido) e inserido na fila de espera dos processos que estão bloqueados na variável de condição, não gastando tempo de CPU. A operação signal traduz-se pela libertação de um desses processos da fila de espera, passando-o para o estado de "pronto", podendo de seguida ser escalonado.

Diga o que entende por "middleware orientado a mensagens" identificando sucintamente os seus principais componentes funcionais.

R: O middleware orientado a mensagens é um método de comunicação assíncrona e persistente baseado em trocas de mensagens entre a componente do software num sistema distribuído. Cada um dos componentes inclui duas filas de mensagens que armazenam as mensagens a enviar e as mensagens recebidas. As mensagens são enviadas para um gestor de mensagens, o "Message broker", que gere um conjunto de filas de mensagens e realiza as trocas de mensagens entre filas. A transmissão de mensagens ocorre através de canais estabelecidos por "Message Channel Agents"(MCA's) que empacotam/desempacotam as mensagens a receber/enviar pelo canal de rede.

Considere o algoritmo descentralizado de exclusão mútua distribuıda estudado nas aulas. Em que medida a sincronização dos relógios dos diversos processos é importante ?

R: O algoritmo descentralizado de exclusão mútua distribuída baseia-se na hora de mensagens entre os processos como forma de sincronização. Um processo quando quer executar uma determinada região crítica tem de enviar um pedido a todos os outros, pedido este que é acompanhado com uma marca temporal. É com base nesta marca temporal que um processo recetor de uma destas mensagens, decide adiar ou não a resposta de permissão, isto é, caso também este pretenda aceder à região crítica, compara o tempo entre o seu pedido e o recebido. Caso os relógios dos processos não estejam sincronizados serão induzidos erros na decisão do envio de resposta, levando a que processos que tenham pedido à mais tempo para executar a região crítica sejam ultrapassados por outros processos.

Considere um serviço simplificado de reserva e limpeza de salas para realização de testes. Assuma um número fixo de 10 salas, com capacidade para 20 pessoas cada. A reserva de salas deverá bloquear enquanto todas as salas pretendidas não se encontrarem livres. O teste inicia-se 10 minutos após a reserva de salas ter sido concluıda, com os alunos presentes nesse momento. A operação presença deverá bloquear até se inciar o teste, retornando "false"em caso de atraso. A operação começar limpeza devolve o identificador de uma sala cujo teste terminou, bloqueando até tal ser possível. As salas ficam livres após a sua limpeza.

```
interface Controlador {
  reserva(int testeId, int[] salaIds); //docente reserva conjunto de salas
  boolean presena(int testeId);//aluno regista presena no teste
  entrega(int testeId);// aluno entrega o seu teste
  int comear_limpeza();// obter sala para limpeza
  terminar_limpeza(int salaId);//fim de limpeza de sala
}
```

1. Apresente uma classe que implemente o interface Controlador tendo em conta que os seus métodos poderão ser invocados num ambiente multi-threaded.

```
class Sala{
   private int id;
   private int capacidade;
   //private boolean reservada;
   private boolean paraLimpeza;
   private ReentrantLock lock;
   public Sala(int id){
      this.id = id;
      this.capacidade = 20;
      //this.reservada = false;
      this.paraLimpeza = false;
      this.lock = new ReentrantLock();
   }
   public int getId(){
      return this.id;
   }
   public void lock(){
      this.lock.lock();
   }
   public void unlock(){
      this.lock.unlock();
   }
//
//
class Teste{
   private int id;
   private boolean comecou;
   private int nPresencas;
   private int nEntregas;
   private ArrayList<Sala> salas;
   public Teste (int id, ArrayList<Sala> salas){
      this.id = id;
      this.comecou = false;
      this.nPresencas = 0;
      this.nEntregas = 0;
      this.salas = salas;
   }
   public synchronized boolean entregar(){
      boolean res = false;
      this.nEntregas++;
      if (this.nEntregas == this.nPresencas) res = true;
      return res;
   }
   public synchronized boolean presenca(){
      boolean res = !comecou;
      try{
         if(res) {
            this.nPresencas++;
            this.wait();
```

```
            }
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        return res;
    }
    public synchronized void comecar (){
        this.comecou = true;
        this.notifyAll();
    }
    public ArrayList<Sala> getSalas (){
        return this.salas;
    }
}
//
//
class Controlador {
    private Map<Integer,Sala> salas;
    private LinkedList<Sala> paraLimpeza;
    private Map<Integer,Teste> testes;
    public Controlador(){
        this.salas = new HashMap <> ();
        for(int i = 0; i < 10; i++){
            Sala sala = new Sala(i);
            this.salas.put(i,sala);
        }
        this.paraLimpeza = new LinkedList <> ();
        this.testes = new HashMap <> ();
    }
    public int comecar_limpeza(){
        Sala sala = null;
        try{
            synchronized(this.paraLimpeza){
                while(this.paraLimpeza.size() == 0) this.paraLimpeza.wait();
                sala = this.paraLimpeza.pop();
            }
        }catch(InterruptedException e){
            e.printStackTrace();
        }
        //mantem o lock preso at terminar a limpeza
        sala.lock();
        return sala.getId();
    }
    public void terminar_limpeza (int salaId){
        Sala sala = null;
        synchronized(this.paraLimpeza){
            sala = this.salas.get(salaId);
            if(sala != null) sala.unlock();
        }
        //com o lock detido(o lock  reentrante)
        sala.unlock();
    }
    public Teste reserva(int testeId, int [] salaIds){
        TreeSet<Integer> sIds = new TreeSet <Integer> ();
        ArrayList<Sala> salas = new ArrayList<>();
        for(int i = 0; i < salaIds.length; i++) sIds.add(salaIds[i]);
        synchronized(this.salas){
            for(Integer id : sIds){
                Sala s = this.salas.get(id);
                s.lock();
                salas.add(s);
            }
        }
        Teste teste = null;
        synchronized(this.testes){
            if(!this.testes.containsKey(testeId)){
                teste = new Teste(testeId,salas);
                this.testes.put(testeId,teste);
            }
        }
        return teste;
    }
    public boolean presenca (int testeId){
        Teste teste = null;
        synchronized(this.testes){
            teste = this.testes.get(testeId);
        }
        return teste.presenca();
    }
    public void adicionaSalaLimpeza(Teste teste){
```

```
      ArrayList<Sala> salas = teste.getSalas();
      synchronized(this.paraLimpeza){
         for(Sala s: salas){
            this.paraLimpeza.add(s);
         }
         this.paraLimpeza.notifyAll();
      }
   }
   public void entrega (int testeId){
      Teste teste = null;
      synchronized(this.testes){
         teste = this.testes.get(testeId);
      }
      boolean acabou = teste.entregar();
      if (acabou){
         synchronized(teste){
            teste.notifyAll();
         }
      }
   }
}
```

2.Implemente um servidor em rede usando sockets TCP que disponibilize os métodos da classe desenvolvida na pergunta anterior. O servidor deverá suportar a conexão de múltiplos clientes em simultâneo.

```
class TreatClient implements Runnable{
   private Socket cs;
   private Controlador controlador;
   private BufferedReader in;
   private PrintWriter out;
   public TreatClient (Controlador controlador, Socket cs){
      this.cs = cs;
      this.controlador = controlador;
   }
   public void run (){
      try{
         in = new BufferedReader(new InputStreamReader(cs.getInputStream()));
         out = new PrintWriter(cs.getOutputStream(), true);
         String current;int testeId;
         try{
            while((current = in.readLine()) != null){
               switch(current){
                  case "reserva":
                     testeId = Integer.parseInt(in.readLine());
                     String [] tokens = (in.readLine()).split(" ");
                     int [] salas = new int [tokens.length];
                     for(int i = 0; i < tokens.length; i++) salas[i] =
                         Integer.parseInt(tokens[i]);
                     Teste t = this.controlador.reserva(testeId,salas);
                     System.out.println("Salas reservadas");
                     out.println("Salas reservadas");
                     Thread.sleep(60000);//1 mins
                     t.comecar();
                     out.println("Teste Comecou");
                     synchronized(t){
                        t.wait();
                     }
                     controlador.adicionaSalaLimpeza(t);
                     break;
                  case "presenca":
                     testeId = Integer.parseInt(in.readLine());
                     boolean presenca = this.controlador.presenca(testeId);
                     out.println("Presente: " + presenca);
                     System.out.println("Presente: " + presenca);
                     break;
                  case "entrega":
                     testeId = Integer.parseInt(in.readLine());
                     this.controlador.entrega(testeId);
                     out.println("Entregue");
                     System.out.println("Entregue");
                     break;
                  case "comecar_limpeza":
                     testeId = this.controlador.comecar_limpeza();
```

```
                    out.println("Comecar limpeza: " + testeId);
                    System.out.println("Comecar limpeza: " + testeId);
                    break;
                case "terminar_limpeza":
                    testeId = Integer.parseInt(in.readLine());
                    this.controlador.terminar_limpeza(testeId);
                    out.println("Fim de limpeza de sala");
                    System.out.println("Fim de limpeza de sala");
                    break;
                default:
                    out.println(" Operao  Invlida ");
                    break;
            }
         }
         System.out.println("Connection Closed!");
         this.cs.shutdownInput();
         this.cs.shutdownOutput();
         this.cs.close();
      }
      catch(NumberFormatException e){out.println("Argumento Invlido");}
      catch(InterruptedException e){e.printStackTrace();}
   }catch(IOException e){
      e.printStackTrace();
   }
  }
}
//
//
public class Servidor{
  public static void main (String [] args){
   try{
      ServerSocket socket = new ServerSocket(12345);
      Controlador controlador = new Controlador();
      while(true){
         Socket cs = socket.accept();
         System.out.println("Connection Accepted!!");
         Thread t = new Thread( new TreatClient (controlador,cs));
         t.start();
      }
   }catch(IOException e){
      e.printStackTrace();
   }
  }
}
```

2. Teste 2

Distinga, em termos de objetivo e forma de utilização, as primitivas de lock/unlock e wait/notify.

R: O objetivo das primitivas lock/unlock é garantir que apenas um processo se encontra na zona envolvida por estas primitivas. O objetivo das primitivas wait/notify é que os processos voluntariamente adormeçam enquanto o predicado de uma variável de condição não é verificada, sendo acordados por outros processos quando esses alteram algo e existe a possibilidade de o predicado ser válido.

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 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 o que tem de realizar adquire novamente o lock, pois vai mexer em algo partilhado e consoante aquilo que faz pode notificar quem está à espera naquela condição libertando o lock de seguida.

Das arquitecturas de sistemas distribuıdos que estudou qual ou quais se adequariam melhor a um sistema de suporte a redes sociais?

R: As redes sociais são baseadas em eventos/publicações as quais podem gerar reacções em determinados componentes. É exatamente neste contexto que foram criadas as arquiteturas baseadas em eventos que a meu ver são as que melhor se adequam a um sistema de suporte deste tipo de ambiente.

Este tipo de arquitetura é constituído por um grande repositório de eventos(dados), dados estes que são homogéneos, orientados a uma aplicação em tempo real. Trata-se de um barramento de eventos e vários componentes que de forma assíncrona(não esperando por um determinado evento) agem perante estes. Tal comportamento é identificável em qualquer rede social atual.

Neste tipo de arquitetura, os processos publicam eventos e o middleware assegura que apenas os processos que se subscreveram (“se inscreveram”) para esses eventos os receberão;

Qual a relevância do sistema operativo na resolução do problema de exclusão mútua no modelo de memória partilhada e no modelo de passagem de mensagens?

R: A função do sistema operativo na resolução do problema de exclusão mútua tem por base uma eficiente gestão dos recursos. Este é responsável por bloquear processos, prevenindo-os de consumir tempo de CPU, enquanto não tiverem permissão para avançar para a região crítica.

A ação do sistema operativo neste tipo de problemas no modelo de memória partilhada e no de passagem de mensagens apenas difere no momento em que os processos são bloqueados e libertados.

Em memória partilhada tal acontece quando se tenta obter o lock, no caso da passagem de mensagens, os processos são bloqueados desde que enviam os pedido até à receção da resposta.

Em ambos os casos evitam-se as esperas ativas.

Considere um serviço simplificado de controlo de aquecimento numa casa (exemplo de internet of things) onde os equipamentos se ligam por TCP/IP a um servidor. Existem os seguintes dispositivos: Um termómetro que se liga para indicar a temperatura atual da casa; Um controlo de temperatura alvo que se liga para indica o limiar de activação da caldeira; Um relé que controla a caldeira e que indica o seu estado actual (off = false ou on = true) e aguarda indicação de mudança de estado via o método on off. Os métodos temperatura e limiar devem retornar imediatamente, mas o método on off deve bloquear caso o estado fornecido seja idêntico ao estado actual do sistema. Por exemplo, se a temperatura actual for de 16° e o limiar actual for de 19° então o sistema sabe que a caldeira deve estar ligada para aquecer a casa; se o relé da caldeira invocar on off(true), indicando já estar ligado, então o método só deve retornar (indicando false) quando a temperatura ultrapassar os 19°.

```
interface Controlador {
temperatura(int centigrados);
// medidor indica temperatura actual
limiar(int centigrados);
// utilizador indica limiar de activao
boolean on_off(boolean estadoatual); // caldeira pergunta se estado mudou
}
```

1. Apresente uma classe que implemente o interface Controlador tendo em conta que os seus métodos poderão ser invocados num ambiente multi-threaded.

```
interface Controlador{
   public void temperatura(int centigrados);
   public void limiar(int centigrados);
   public boolean on_off(boolean estadoatual);
}
class Controlador_Temp implements Controlador{
   private int temperatura;
   private int limiar;
   private ReentrantLock lock;
   private Condition inalterado;
   public Controlador_Temp(int temperatura, int limiar){
      this.temperatura = temperatura;
```

```
      this.limiar = limiar;
      this.lock = new ReentrantLock();
      this.inalterado = lock.newCondition();
   }
   public void temperatura(int centigrados){
      this.lock.lock();
      int tempAnterior = this.temperatura;
      this.temperatura = centigrados;
      if (tempAnterior != centigrados) this.inalterado.signalAll();
      this.lock.unlock();
   }
   public void limiar(int centigrados){
      this.lock.lock();
      int limiarAnterior = this.temperatura;
      this.limiar = centigrados;
      if (limiarAnterior != centigrados) this.inalterado.signalAll();
      this.lock.unlock();
   }
   public boolean on_off(boolean estadoatual){
      boolean res = false;
      try{
         this.lock.lock();
         if(estadoatual == true){
            while(temperatura < limiar) inalterado.await();
            res = false;
         }else{
            while(temperatura >= limiar) inalterado.await();
            res = true;
         }
      }catch(InterruptedException e){
         e.printStackTrace();
      }
      finally{
         this.lock.unlock();
      }
      return res;
   }
}
```

2. Implemente um servidor em rede usando sockets TCP que disponibilize os métodos da classe desenvolvida na pergunta anterior. O servidor deverá suportar a conexão de múltiplos clientes em simultâneo.

```
class TreatClient implements Runnable{
   private Socket cs;
   private Controlador_Temp controlador;
   private BufferedReader in;
   private PrintWriter out;
   public TreatClient (Controlador_Temp controlador, Socket cs){
      this.cs = cs;
      this.controlador = controlador;
   }
   public void run (){
      try{
         in = new BufferedReader(new InputStreamReader(cs.getInputStream()));
         out = new PrintWriter(cs.getOutputStream(), true);
         String current;int temp;
         try{
            while((current = in.readLine()) != null){
               switch(current){
                  case "temperatura":
                     temp = Integer.parseInt(in.readLine());
                     this.controlador.temperatura(temp);
                     System.out.println("Temperatura Atual: " + temp);
                     out.println("Temperatura Atual: " + temp);
                     break;
                  case "limiar":
                     temp = Integer.parseInt(in.readLine());
                     this.controlador.limiar(temp);
                     out.println("Limiar Atual: " + temp);
                     System.out.println("Limiar Atual: " + temp);
                     break;
                  case "on_off":
                     boolean bool = Boolean.parseBoolean(in.readLine());
                     this.controlador.on_off(bool);
                     out.println("Estado da caldeira: " + (!bool));
```

```
                    System.out.println("Estado da caldeira: " + (!bool));
                    break;
                default:
                    out.println(" Operao  Invlida ");
                    break;
            }
         }
         System.out.println("Connection Closed!");
         this.cs.shutdownInput();
         this.cs.shutdownOutput();
         this.cs.close();
      }
      catch(NumberFormatException e){out.println("Argumento Invlido");}
   }catch(IOException e){
      e.printStackTrace();
   }
  }
}
public class Servidor{
  public static void main (String [] args){
   try{
      ServerSocket socket = new ServerSocket(12345);
      Controlador_Temp controlador = new Controlador_Temp(14,19);
      while(true){
         Socket cs = socket.accept();
         System.out.println("Connection Accepted!!");
         Thread t = new Thread( new TreatClient (controlador,cs));
         t.start();
      }
   }catch(IOException e){
      e.printStackTrace();
   }
  }
}
```

3. Teste 3

Explique como funcionam genericamente as operações de lock e unlock ao nıvel do sistema operativo.

R: As operações lock e unlock disponibilizadas pelo sistema operativo eliminam as esperas ativas no acesso a regiões críticas. Um processo que se encontre na secção crítica, impede os outros que pretendem entrar, de obter o lock, passando ao estado de bloqueado, não sendo escalonados e não consumindo tempo de processador. No fim de execução de região crítica, o processo invoca o unlock, libertando o lock, que muda o estado dos outros processos de bloqueado para pronto, podendo estes de seguida, serem escalonados.

Indique e descreva sucintamente as arquitecturas descentralizadas de sistemas distribuıdos que estudou.

R: As arquiteturas descentralizadas dividem-se em dois tipos, estruturadas e não estruturada. As estruturadas possuem uma estrutura organizacional bem definida. Nesta, a totalidade do domínio dos dados, é dividida pelo contradomínio dos constituintes da arquitetura através de uma função de hash. As não estruturadas não possuem qualquer tipo de estrutura, tratam-se de um conjunto das nós dispersos, cada um possuindo uma determinada vista parcial da rede. Estas baseiam-se no princípio básico de aleatoriedade, fator responsável pela troca de modos das vistas de qualquer dos nós constituintes.

Discuta as diferenças ao assegurar exclusão mútua entre processos a executar numa única máquina e num sistema distribuıdo.

R: A exclusão mútua processa-se de forma diferente caso haja realizado numa máquina ou num sistema distribuído. Numa máquina é alcançado através das primitivas de sincronização lock e unlock disponibilizados pelo sistema operativo, que permitem bloquear processos enquanto não conseguirem obter o lock de acesso à região crítica. Em sistemas distribuídos, quer centralizados ou descentralizados este é obtido por trocas de mensagens. O cliente envia uma mensagem a pedir permissão para escutar a região crítica e só poderá executar se o coordenador, no caso das centralizadas, ou todos os outros constituintes, no caso das descentralizadas, enviem uma mensagem de resposta.

Considere um sistema cliente/servidor de uma variante do jogo de "Batalha Naval". O servidor faz a gestão de uma grande matriz de n x m de submarinos; cada posição da matriz vale 1 se estiver ocupada ou 0 se estiver livre. Suponha que no inicio do jogo o conjunto de jogadores é conhecido e a matriz é inicializada de forma aleatória por uma função iniciar, já existente. O servidor aceita pedidos do tipo disparo c i j cujos argumentos identificam o cliente e as coordenadas do disparo (por exemplo, "joao 13 245"). A pontuação de cada disparo é calculada somando todos os submarinos do segmento da linha i entre j 4 e j + 4, que são afundados, não voltando a contar para outro disparo. Cada jogador pode efectuar 3 disparos. No terceiro disparo, o cliente é bloqueado até ao final do jogo, momento em que recebe como resposta o identificador do jogador vencedor. O jogo termina quando todos os jogadores usarem os seus disparos. Escreva em Java o código do servidor de forma a que atenda eficientemente pedidos concorrentes.

```
public class Matrix {
   int[][] submarinos;
   Map<String, Integer> jogadores;
   int nrAcabados;
   ReentrantLock lock;
   public Matrix(int m, int n, Map<String, Integer> jogadores) {
      this.submarinos = new int[m][n];
      this.jogadores = jogadores;
      this.nrAcabados = 0;
      this.lock = new ReentrantLock();
   }
   public void iniciar() {
      return;
   }
   boolean jogoAcabou() {
      System.out.println(nrAcabados);
      System.out.println(jogadores.size());
      return nrAcabados == jogadores.size();
   }
   public String getVencedor() {
      int max = 0;
      String jogador = null;
      for(Map.Entry<String, Integer> entry : this.jogadores.entrySet()) {
         if (entry.getValue() > max) {
            max = entry.getValue();
            jogador = entry.getKey();
         }
      }
      return jogador;
   }
   void pedido(String jogador, int coordenadaX, int coordenadaY, int nrPedido) {
      int j = coordenadaY - 4;
      int n = coordenadaY + 4;
      this.lock.lock();
      try {
         int pontuacao = this.jogadores.get(jogador);
         for(; j <= n; j++) {
            if(submarinos[coordenadaX][j] == 1) {
               pontuacao++;
               submarinos[coordenadaX][j] = 0;
            }
         }
         this.jogadores.put(jogador, pontuacao);
               synchronized (this) {
         System.out.println(nrPedido == 3);
         if(nrPedido == 3) {
            this.nrAcabados++;
            this.notifyAll();
         }
      }
      }
      finally {
         this.lock.unlock();
```

```
      }
      System.out.println("pedido processado");
   }
}
```

```
public class TreatClient implements Runnable {
   private Socket s;
   private int pedidos;
   private Matrix jogo;
   TreatClient(Socket s, Matrix m) {
      this.s = s;
      this.pedidos = 0;
      this.jogo = m;
   }
   public void trataPedido(String msg) {
      String[] res = msg.split(" ");
      jogo.pedido(res[0], Integer.parseInt(res[1]), Integer.parseInt(res[2]), ++pedidos);
   }
   @Override
   public void run() {
      String rd = null;
      try (Socket s = this.s;
          PrintWriter pw = new PrintWriter(s.getOutputStream(), true);
          BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));)
      {
         while(pedidos < 3 && (rd = in.readLine()) != null)
            trataPedido(rd);
         pw.println("Acabaram os seus pedidos aguarde...");
         synchronized (jogo) {
            while(!jogo.jogoAcabou()) jogo.wait();
         }
         pw.println("O vencedor foi: " + jogo.getVencedor());
      }
      catch(IOException | InterruptedException e) {
         e.printStackTrace();
      }
   }
}
public class Server {
   public static void main(String[] args) throws IOException {
      ServerSocket ss = new ServerSocket(12345);
      Map<String, Integer> jogadores = new HashMap<String, Integer>();
      jogadores.put("joao", 0);
      jogadores.put("maria", 0);
      Socket s = null;
      Matrix m = new Matrix(1000, 1000, jogadores);
      m.iniciar();
      while(true) {
         s = ss.accept();
         Thread t = new Thread(new TreatClient(s, m));
         t.start();
      }
   }
}
```

4. Teste 4

Explique a noção de região crítica e a necessidade dos mecanismos de sincronização de exclusão mútua e variável de condição.

R: Uma região crítica corresponde a um pedaço de código que manipula variáveis partilhadas, de acesso concorrente por múltiplas threads. A não atomicidade de operações como a escrita em variáveis leva à necessidade de mecanismos de sincronização de exclusão mútua, isto é, utilização de metodologias que garantam que apenas um processo se encontra a executar aquele código e a manipular aquelas variáveis. Instruções como o simples incremento de uma variável envolve ler o valor da variável, realizar a soma e só então a escrita. Em qualquer ponto de execução outra thread pode ler o valor da variável e realizar operações sobre ela com o valor incoerente. Também de forma a permitir a sincronização temporal de processo são utilizadas variáveis de condição que permitem que um processo se bloqueie temporariamente até que um determinado predicado se verifique.

Relacione o problema de sincronização do jantar de filósfos com o de transferência entre contas de um banco.

Em ambos é preciso sincronização, para aceder às contas/garfos.

Em ambos é necessário ter uma ordem acesso(lock) à conta/garfo que queremos aceder.

Basicamente a conta é o garfo.

Discuta em que sentido o mecanismo de invocação de procedimentos remotos estudado consegue satisfazer os objectivos de transparência tidos como desejáveis em sistemas distribuıdos.

R: Os mecanismos de inovação de procedimentos remotos(APL's) satisfazem os objetivos de transparência na medida em que todo o processo de envio e receção das mensagens, retransmissão da requisição após um timeout dos resultados são escondidos do programador. Na verdade do lado do cliente tudo ocorre como se a invocação fosse local, de retorno, etc. A única diferença é que em vez de a função estar implementada localmente, existe um stub, função "oca/vazia" que realiza todo o trabalho de obtenção dos resultados.

Considere um serviço de transporte de encomendas que apenas se efectua quando se atinge um peso mınimo de encomendas, 200kg. O primeiro transporte terá o número 1, o segundo 2, etc. Implemente um sistema cliente servidor em que cada cliente faz um pedido de transporte indicando quantos kg tem a sua encomenda. Cada pedido apenas retorna quando se inicia o transporte, indicando ao cliente o número do transporte em que a sua encomenda seguiu.

```
class Controlador {
   private int nrCurrEnc;
   private int kgCurrEnc;
   private int entraram;
   private int sairam;
   private ReentrantLock lock;
   private Condition maior200;
   public Controlador() {
      this.nrCurrEnc = 1;
      this.kgCurrEnc = 0;
      this.entraram = 0;
      this.sairam = 0;
      this.lock = new ReentrantLock();
      this.maior200 = lock.newCondition();
   }
   public int pedido(int kg) {
      int ret = -1;
      this.lock.lock();
      this.entraram++;
      try {
         this.kgCurrEnc += kg;
         while(this.kgCurrEnc < 200) {
            maior200.await();
         }
         maior200.signalAll();
         this.sairam++;
         ret = this.nrCurrEnc;
         if (this.entraram == this.sairam) novaEncomenda();
      }
      catch(InterruptedException e) {
         e.printStackTrace();
      }
      finally {
         this.lock.unlock();
      }
      return ret;
   }
   public void novaEncomenda() {
      this.lock.lock();
      try {
         this.kgCurrEnc = 0;
         this.nrCurrEnc++;
      }
      finally {
         this.lock.unlock();
      }
   }
}
```

```
class TreatClient implements Runnable {
   private Socket s;
   private Controlador c;
   TreatClient(Controlador c, Socket s) {
      this.c = c;
      this.s = s;
   }
   public void run() {
      try (Socket s = this.s;
          PrintWriter out = new PrintWriter(s.getOutputStream(), true);
          BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));)
      {
         out.println("Quantos kg tem a sua encomenda?");
         int kg = Integer.parseInt(in.readLine());
         int nr = c.pedido(kg);
         out.println("O nmero do transporte em que a sua encomenda seguiu foi o " + nr);
      }
      catch(IOException e) {
         e.printStackTrace();
      }
   }
}
public class Servidor {
   public static void main(String[] args) throws IOException {
      ServerSocket ss = new ServerSocket(12345);
      Socket s = null;
      Controlador c = new Controlador();
      while(true) {
         s = ss.accept();
         Thread t = new Thread(new TreatClient(c, s));
         t.start();
      }
   }
}
```

5. Teste 5

O monitor é uma primitiva estruturada de controlo de concorrência que consideramos de "alto nıvel". Comente esta afirmação e indique uma vantagem e uma desvantagem deste elevado nıvel de abstracção.

R:Um monitor é uma primitiva estruturada do controlo de concorrência. Oferece um tipo de dados com controlo de concorrência implícito em todas as operações de exclusão mútua. Com um modelo de concorrência baseado em monitores, o compilador pode inserir mecanismos de exclusão mútua transparente, isto é, sem o programador ter que aceder às primitivas de controlo e realizar o bloqueio e libertação de recursos manuais. Em java a utilização de monitores, é realizada através de métodos e blocos synchronyzed que fazem uso de ReentrantLocks.

Vantagens: Controlo de concorrência implícita (transparente);

Desvantagens: Os locks são reentrantes e podem levar à starvation.

Considere um serviço distribuıdo de presenças em que os seus utilizadores (identificados por nome) registam sempre a sua entrada num determinado espaço (também identificado por nome). Cada espaço é controlado por um servidor local especıfico. Opcionalmente, os utilizadores podem registar a saıda desse espaço no referido servidor. Os utilizadores podem, portanto, entrar, migrar ou abandonar espaços controlados pelo sistema de registo de presença. Além das operações de entrada e saıda de utilizadores, os servidores devem ainda suportar a operação de localização de um utilizador informando qual o id do espaço (e respectiva data e hora de entrada) onde poderá ser encontrado. Implemente o servidor de presença descrito assumindo que recebe como argumentos da sua linha de comando, o id do espaço que deverá controlar, o porto de escuta e os endereços de todos os servidores do sistema. Assuma que todos os servidores são recipro- camente acessıveis, e a data e hora local estão suficientemente sincronizadas. Recorra a System.currentTimeMillis() para obter a data e hora expressa em milisegundos. Procure maximizar a concorrência potencial da sua solução. "$ servidor <espaco-id> <porto-ip> <endereco-ip-serv-1> ... <endereco-ip-serv-n>"

```
public class Presenca {
   private String user;
   private int espaco_id;
   private LocalDateTime entrada;
   private LocalDateTime saida;
   Presenca(String user, int espaco_id) {
      this.user = user;
      this.espaco_id = espaco_id;
      this.entrada = LocalDateTime.now();
      this.saida = null;
   }
   public String getUser() {
      return this.user;
   }
   public int getEspacoId() {
```

```
      return this.espaco_id;
   }
   public String getEntrada() {
      return this.entrada.toString();
   }
   void sair() {
      this.saida = LocalDateTime.now();
   }
}
//
//
public class Presencas {
   public static Map<String, ArrayList<Presenca>> map = new HashMap<String,
       ArrayList<Presenca>>();
}
```

```
class TreatClient implements Runnable {
   private Socket s;
   private int espaco_id;
   private int port_id;
   private ArrayList<String> serverList;
   private PrintWriter out;
   private BufferedReader in;
   private boolean connected;
   TreatClient(Socket s, int espaco_id, int port_id, ArrayList<String> servers) {
      this.s = s;
      this.espaco_id = espaco_id;
      this.port_id = port_id;
      this.serverList = servers;
      this.connected = true;
   }
   void regista_presenca() {
      String username = null;
      try {
         out.println("O seu nome?");
         username = in.readLine();
      }
      catch(IOException e) {
         e.printStackTrace();
      }
      synchronized(Presencas.map) {
         Presenca p = new Presenca(username, this.espaco_id);
         ArrayList<Presenca> pres = Presencas.map.get(username);
         if(pres == null)
            pres = new ArrayList<Presenca>();
         pres.add(p);
         Presencas.map.put(username, pres);
      }
   }
   void server_list() {
      for(String s : serverList)
         out.println(s);
   }
   void encontra_user(String user) {
      ArrayList<Presenca> pres = null;
      synchronized(Presencas.map) {
         pres = Presencas.map.get(user);
      }
      if(pres == null)
         out.println("O utilizador especificado no existe!");
      else {
         Presenca p = pres.get(pres.size() - 1);
         synchronized(p) {
            out.println(p.getUser() + ":");
            out.println("Servidor: " + p.getEspacoId());
            out.println("Hora entrada: " + p.getEntrada());
         }
      }
   }
   void mudar_server(String server) {
      out.println("A mudar o servidor...");
      try {
         this.connected = false;
         new Socket(server, 9999);
      }
      catch(IOException e) {
         e.printStackTrace();
```

```
      }
   }
   void processa_pedido(String msg) {
      String[] line = msg.split(" ");
      switch(line[0]) {
         case "server_list" : server_list();
                              break;
         case "mudar_server" : mudar_server(line[1]);
                               break;
         case "encontra_user" : encontra_user(line[1]);
                                break;
      }
   }
   public void run() {
      String input = null;
      try (Socket s = this.s;)
      {
         this.out = new PrintWriter(s.getOutputStream(), true);
         this.in = new BufferedReader(new InputStreamReader(s.getInputStream()));
         System.out.println(espaco_id);
         regista_presenca();
         while(connected && (input = in.readLine()) != null) {
            processa_pedido(input);
         }
         this.in.close();
         this.out.close();
      }
      catch(IOException e) {
         e.printStackTrace();
      }
   }
}
public class Servidor {
   public static void main(String[] args) throws IOException {
      int espaco_id = Integer.parseInt(args[0]);
      int porto_ip = Integer.parseInt(args[1]);
      ArrayList<String> servers = new ArrayList<String>();
      for(int i = 2; i < args.length; i++)
         servers.add(args[i]);
      ServerSocket ss = new ServerSocket(porto_ip);
      Socket s = null;
      while(true) {
         s = ss.accept();
         Thread t = new Thread(new TreatClient(s, espaco_id, porto_ip, servers));
         t.start();
      }
   }
}
```

6. Teste 6

Explique como utiliza a operação de wait(cond, lock) e por que razão é necessário associar o argumento lock.

R: A operação de wait utiliza-se em problemas de ordem de execução, isto é quando uma thread não pode avançar enquanto um determinado predicado, associado a um região de memória partilhada, se verifique. Uma vez que se trata de memória partilhada é necessário que se controle a concorrência de threads no acesso a tal região do código, motivo pelo qual cada variável de condição está associada a um lock. Esta operação é utilizada da seguinte forma:

```
lock.lock()
while(!PREDICADO){
wait(cond,lock);
}
//(FAZER ALGO)
lock.unlock();
```

Antes de verificar o predicado é necessário obter o lock.Por esta razão e de forma a que mais threads possam verificar o predicado enquanto estamos bloqueados, é necessário passar o lock ao wait. A operação de wait inclui pois as operações de libertações e re-obtenção do lock.

Descreva sucintamente as funcionalidades de um servidor de objectos em sistemas de objetos distribuıdos.

R: A principal funcionalidade de um servidor de objetos é a gestão de um conjunto de objetos e intermediar a realização de pedidos aos objetos. Os objetos que controla são detentores dos variados serviços que oferece.Este é responsável por verificar se um determinado pedido se destina a um dos objetos que possui, por assegurar que tal objeto se encontre carregado, por realizar o pedido ao objeto e por retornar a resposta ao cliente. Este deve ainda remover da cache objetos que já não são utilizados há muito tempo.

Considere uma biblioteca de apoio a um concurso para ser usada num ambiente multi-threaded. Esta deve permitir serem adicionadas questões (pares pergunta-resposta), e oferecer a possibilidade de threads competirem para tentarem responder. Devem ser implementadas as seguintes interfaces:

```
interface Controlador {
void adiciona(String pergunta, String resposta);
Questao obtem(int id);
}
interface Questao {
String responde(String resposta);
int id();
}
```

A operação adiciona introduz um novo par pergunta-resposta, criando um objecto Questao, etiquetado por um id numérico crescente (1, 2, 3, . . .). A operação obtem devolve um objecto que representa uma questão que tiver sido previamente adici- onada, com id maior do que o argumento,

e que se encontre ainda disponıvel: não tenha ainda sido respondida correctamente nem tenha sido sujeita a mais de 10 tentativas de resposta. Caso não exista nenhuma, deverá bloquear até tal ser possıvel. O método responde deverá devolver "R", "C", ou "E", conforme a questão já tiver sido previamente respondida correctamente, a resposta esteja certa, ou a resposta esteja errada, respectivamente. Tenha cuidado para evitar o uso continuamente crescente e desnecessário de memória (memory leak).

```
public class Controlador implements ControladorInterface {
   private Map<Integer, Questao> questoes;
   private int nrQuestoes;
   public Controlador() {
      this.questoes = new HashMap<Integer, Questao>();
      this.nrQuestoes = 0;
   }
   public int getNrQuestoes() {
      return this.nrQuestoes;
   }
   /*
   A  operao  adiciona introduz um novo par pergunta-resposta, criando um objecto
   Questao, etiquetado por um id numrico crescente (1, 2, 3, . . . ).
    */
   public synchronized void adiciona(String pergunta, String resposta) {
      this.questoes.put(nrQuestoes, new Questao(nrQuestoes, pergunta, resposta));
      nrQuestoes++;
      this.notifyAll();
   }
   /*
   A  operao  obtem devolve um objecto que representa uma questo que tiver
   sido previamente adicionada, com id maior do que o argumento, e que se encontre
   ainda  disponvel :  no  tenha ainda sido respondida correctamente nem tenha sido sujeita
   a mais de 10 tentativas de resposta. Caso no exista nenhuma, dever bloquear
    at  tal ser  possvel .
    */
   public synchronized Questao obtem(int id) {
      Questao q = null;
      int i = id;
      for(; i < nrQuestoes; i++) {
         q = this.questoes.get(i);
         if (q != null && q.disponivel()) return q;
         if (q != null && !q.disponivel()) this.questoes.remove(id);
      }
      try {
         while((q = this.questoes.get(nrQuestoes - 1)) == null
            || !q.disponivel()) {
            wait();
         }
      }
      catch(InterruptedException e) {
         e.printStackTrace();
      }
      return q;
   }
}
public class Questao implements QuestaoInterface {
   private int id;
   private String pergunta;
   private String resposta;
   private int tentativas;
   private boolean jaAcertaram;
   private ReentrantLock lock;
   /*
   A  operao  obtem devolve um objecto que representa uma questo que tiver sido
   previamente adicionada, com id maior do que o argumento, e que se encontre ainda
    disponvel :  no  tenha ainda sido respondida correctamente nem tenha sido sujeita a
   mais de 10 tentativas de resposta.
    */
   public Questao(int id, String pergunta, String resposta) {
      this.id = id;
      this.pergunta = pergunta;
      this.resposta = resposta;
      this.tentativas = 10;
      this.jaAcertaram = false;
      this.lock = new ReentrantLock();
```

```
   }
   public String getPergunta() {
      return this.pergunta;
   }
   boolean disponivel() {
      return (!this.jaAcertaram) && (this.tentativas > 0);
   }
   public String responde(String resposta) {
      /* O mtodo responde dever devolver R , C , ou E , conforme a questo
         j tiver sido previamente respondida correctamente, a resposta esteja certa,
         ou a resposta esteja errada, respectivamente.
       */
      this.lock.lock();
      try {
         this.tentativas--;
         if(this.resposta.equals(resposta) && this.jaAcertaram)
            return "R";
         else if (!this.resposta.equals(resposta))
            return "E";
         else {
            this.jaAcertaram = true;
            return "C";
         }
      }
      finally {
         this.lock.unlock();
      }
   }
   public int id() {
      return this.id;
   }
}
```

```
public class QuestaoMaker implements Runnable {
   Controlador c;
   public QuestaoMaker(Controlador c) {
      this.c = c;
   }
   @Override
   public void run() {
      String[] questao = new String[2];
      int i = 0;
      try {
         while(true) {
            //questao = Util.novaPergunta();
            //c.adiciona(questao[0], questao[1]);
            c.adiciona("lala" + i, "lala" + i);
            i++;
            Thread.sleep(10 * 1000);
         }
      }
      catch(InterruptedException e) {
         e.printStackTrace();
      }
   }
}
public class Server {
   public static void main(String[] args) throws IOException {
      Controlador c = new Controlador();
      ServerSocket ss = new ServerSocket(12345);
      Socket s = null;
      Thread qMaker = new Thread(new QuestaoMaker(c));
      qMaker.start();
      while(true) {
         s = ss.accept();
         Thread client = new Thread(new TreatClient(c, s));
         client.start();
      }
   }
}
public class TreatClient implements Runnable {
   private Controlador c;
   private Socket s;
   private ArrayList<Integer> respondidas;
   public TreatClient(Controlador c, Socket s) {
      this.c = c;
      this.s = s;
      this.respondidas = new ArrayList<>();
   }
```

```
   public void veredito(String veredito, PrintWriter out) {
      switch(veredito) {
         case "R": out.println("Respondida");
                   break;
         case "C": out.println("Certa");
                   break;
         case "E": out.println("Errada");
                   break;
      }
   }
   /*
   Cada cliente ligado, at fechar a  ligao ,  poder  em ciclo: enviar  Pergunta  ,
   esperar pelo enunciado de uma pergunta, enviar a resposta e esperar pelo resultado,
   que  dever  ser  Respondida  ,  Certa  , ou  Errada
    */
   @Override
   public void run() {
      String line = null, resposta = null, veredito = null;
      int id = -1;
      try (BufferedReader in = new BufferedReader(new InputStreamReader(s.getInputStream()));
          PrintWriter out = new PrintWriter(s.getOutputStream(), true);
          Socket s = this.s;) {
         while((line = in.readLine()) != null) {
            if (!line.equals("Pergunta"))
               continue;
            Questao q = c.obtem(id);
            out.println(q.getPergunta());  // mostra pergunta ao cliente
            resposta = in.readLine();     // recebe resposta
            veredito = q.responde(resposta); // verifica resposta
            if(!veredito.equals("E")) id = q.id();
            veredito(veredito, out);
            respondidas.add(q.id());
         }
      }
      catch(IOException e) {
         e.printStackTrace();
      }
   }
}
```

7. Perguntas teóricas extra

Diferenças ao assegurar exclusão mútua num Sistema Distribuído centralizado e não centralizado

R: Num sistema distribuído centralizado existe um processo coordenado de acesso à zona crítica, ao qual todos os outros processos devem enviar um pedido caso pretenda executar a zona crítica. O coordenador decide quando os processos podem executar, enviando-lhes uma resposta. Os outros processos quando recebem a resposta começam a executar, enviando uma mensagem a libertar a zona crítica, aquando do seu término. Num ambiente não centralizado um processo tem de enviar uma mensagem a todos os outros processos, só podendo executar quando receber todas as mensagens de resposta.

Exclusão mútua vs Ordem de execução R: Ambas se tratam da situações de controlo de concorrência. Exclusão mútua ocorre quando vários processos concorrem no acesso a recursos partilhados, onde os processos se podem encontrar temporariamente impedidos de continuar. Nesta situação utilizam-se fundamentalmente locks, para fazer o controlo de concorrência. Na situação de ordem de exeução, existem padrões de cooperação e de dependência entre ações de processos. Os processos bloqueiam-se voluntariamente e são libertados explicitamente por outras(um processo poderá não poder prosseguir até uma dada ação de outro processo. Situações como estas são na sua maioria resolvidos por variáveis de condição.

Qual a principal vantagem de escolher o sistema operativo na concretização de primitivas de controlo de concorrência ?

R: A principal vantagem de escolher o sistema operativo no controlo de concorrência é a eficiência o nível de recursos, nomeadamente, de CPU, ao diminuir a ocorrência de esperas ativas.Isto é possível pela disponibilização dos locks e variáveis de condição que permitem o adormecimento de processos até que os recursos se encontrem disponíveis.

Implementar exclusão mútua com test&set

R: Test-and-set corresponde a uma instrução usada para atribuir o valor verdadeiro a uma variável em memória e devolver o seu antigo valor de forma totalmente atómica. Facilmente se implementa exclusão mútua com esta instrução, mantendo num ciclo uma dada thread até que o valor recebido por esta instrução seja falso. Contudo envolve esperas ativas.

```
while(TestAndSet(&lock)==true);
//( sesso  crtica )
lock=false
```

Diga o que entende por Starvation

R: Starvation é um problema que pode ser encontrado em ambientes de programação concorrente, onde um a processo é perpetuamente negado o acesso aos recursos necessários à sua continuação de trabalho. Problemas deste género tendem a ocorrer com a utilização de locks re-entrantes, erros em algoritmos de escalonamento ou exclusão mútua.

Secções críticas e os problemas relacionados

Uma secção crítica corresponde a um segmento de código que acede a recursos partilhados por novas threads. Neste tipo de regiões é necessário garantir:

- exclusão mútua, apenas um processo pode executar a secção crítica num dado momento;
- ausência de deadlock, se vários processos estão a tentar entrar na secção crítica um deles deve inevitavelmente conseguir.
- ausência de starvation, se um processo tenta entrar na secção crítica, inevitavelmente conseguirá.

Vantagens e desvantagens da comunicação orientada à conexão e por datagramas

R: A grande vantagem de comunicação por datagramas, ou seja, não orientada à conexão e a eficiência. Isto, quando não ocorrem perdas de mensagens ou corrupção de dados, podendo-se nestes casos traduzir em overheads enormes. A eficiência de que falamos resulta da ausência de conexão(negociação) entre o emissor e recetor, os pacotes são simplesmente transmitidos de uma origem para um destino.

Comunicações orientados à conexão são mais complexas, existindo uma negociação inicial antes de iniciar a transmissão de dados. Por outro lado, também são mais confiáveis, existindo controlo de perdas e corrompimento.

Connection-oriented(TCP):

+complexo;

+fiável;

-eficiente

Connectionless(UDP):

+eficiente;

+simples;

-fiável

Distinga os paradigmas de memória partilhada e troca de mensagens na construção de aplicações concorrentes e distribuídas

R: Os paradigmas divergem na forma que os componentes concorrentes comunicam entre si. No paradigma de troca de mensagens, como o nome indica, tal comunicação é efetuada pela troca de mensagens, podendo a sua leitura ser feita síncrona, traduzindo-se numa espera da receção da mensagem por parte do emissor, ou assincronamente, não ocorrendo esperas. Este tipo de comunicação tende a ser mais simples, e é considerado uma forma mais robusta de programação.

Em paradigmas de memória partilhada, a comunicação é efetuada ao alterar o conteúdo de áreas de memória partilhadas, recorrendo à utilização de primitivas de sincronização para controlar o acesso aos dados e evitar inconsistências(locks, semáforos, variáveis de condição, monitores). As primitivas utilizadas em trocas de mensagens são básicas do tipo send/receive.