ISCTE – Instituto Universitário de Lisboa

Programação Concorrente e Distribuída
1ª Época
2016-2017

Licenciatura em Engenharia Informática
Licenciatura em Informática e Gestão de Empresas
Licenciatura em Engenharia de Telecomunicações e Informática

Duração: 2:30 horas
Sem Consulta

1.1. **Complete** a sua identificação em todas as folhas, com nome, número e curso.
1.2. **A prova é sem consulta. Em cima da sua mesa deve ter apenas uma esferográfica e um documento de identificação. Os telemóveis devem estar desligados.**
1.3. **Qualquer troca de informações com colegas implica a anulação imediata da prova e o procedimento disciplinar correspondente.**
1.4. Serão dados 10 minutos no início da prova para esclarecimento de dúvidas. Se tiver dúvidas em relação a alguma pergunta considere os pressupostos que entender de forma a justificar a sua resposta. Indique os pressupostos que assumir.
1.5. Responda às **questões 1 a 4** numa folha de respostas e às **questões 5 e 6** numa outra.

ServerSocket

	ServerSocket(int port) Creates a server socket, bound to the specified port.
`Socket`	accept() throws IOException Listens for a connection to be made to this socket and accepts it. The method blocks until a connection is made. A new Socket s is created.

InetAddress

`static InetAddress`	getByName(String host) Determines the IP address of a host, given the host's name.

Socket

	Socket(InetAddress address, int port) Creates a stream socket and connects it to the specified port number at the specified IP address.
InputStream	**getInputStream**() Returns an input stream for this socket.
OutputStream	**getOutputStream()** Returns an output stream for this socket.

ObjectInputStream

	ObjectInputStream(InputStream) Creates an ObjectInputStream that reads from the specified InputStream.
`Object`	readObject() throws IOException, ClassNotFoundException Read an object from the ObjectInputStream.

ObjectOutputStream

	ObjectOutputStream(OutputStream) Creates an ObjectOutputStream that writes to the specified OutputStream.

void	writeObject(Object obj) throws IOException Write the specified object to the ObjectOutputStream.

1. Descreva o que acontece a uma Thread quando invoca o método wait de um determinado objecto. [1.5 valores]

A *thread* liberta o cadeado intrínseco do objecto e fica no estado **NOT-RUNNABLE** até que uma outra *thread* invoque o método *notify* ou *notifyAll* desse objecto. O método *wait* lança uma *InterruptedException* caso a flag de interrupção da *thread* esteja ou fique activa.

2. Descreva o ciclo de vida de uma Thread. [1.5 valores]

Uma *thread* quando é criada está no estado **NEW**.

Assim que arranca (após invocado o método *start*) passa para o estado **RUNNABLE** onde está disponível para correr.

Quando lhe é atribuído, pelo *scheduler*, tempo de execução, a *thread* passa de **RUNNABLE** para o estado **RUNNING** e assim que é retirada do processador volta ao estado **RUNNABLE**.

Caso seja invocado um método bloqueante (por exemplo *wait* ou *join* entre outros) a *thread* passa para o estado **NOT-RUNNABLE** onde não pode ser agendada pelo *scheduler* para execução. Quando o método retornar a *thread* volta ao estado **RUNNABLE**.

Quando a *thread* termina passa para o estado **DEAD** de onde não pode sair.

3. Indique o que será escrito na consola decorrente da execução do seguinte programa (Barreira e SleepMessages). [1 valor]

```
11  public class Barreira {
12      private int n = 0;
13
14      public synchronized void p() throws InterruptedException {
15          while (n > 0) {
16              wait();
17          }
18          n++;
19      }
20
21      public synchronized void r() {
22          n = 0;
23          notifyAll();
24      }
25  }
26

50  public class SleepMessages extends Thread {
51      private Barreira u;
52      private long startTime;
53
54      public SleepMessages(String name, Barreira u, long startTime) {
55          setName(name);
56          this.startTime = startTime;
57          this.u = u;
58      }
59
60      public void run() {
61          String importantInfo[] = { "Red", "Green" };
62          try {
63              for (int i = 0; i < importantInfo.length; i++) {
64                  u.p();
65                  try {
66                      sleep(2000);
67                      System.out.println(getName() + "-" + importantInfo[i] + "-"
68                              + getTime());
69                  } catch (InterruptedException e) {
```

```
70                      System.out.println(getName() + "- OPSS -" + getTime());
71                  }
72               }
73            } catch (InterruptedException e) {
74               System.out.println(getName() + "- Dead -" + getTime());
75            }
76         }
77
78         private String getTime() {
79            return (System.currentTimeMillis() - startTime) + "s";
80         }
81
82         public static void main(String args[]) throws InterruptedException {
83            long sTime = System.currentTimeMillis();
84            Barreira u = new Barreira();
85            Thread t1 = new SleepMessages("t1", u, sTime);
86            Thread t2 = new SleepMessages("t2", u, sTime);
87            System.out.println("Here they go!...");
88            t1.start();
89            sleep(1000);
90            t2.start();
91            sleep(1000);
92            t2.interrupt();
93            sleep(1000);
94            u.r();
95            t1.join();
96            System.out.println("Main - Main done - "
97                  + (System.currentTimeMillis() - sTime) + "s");
98         }
99      }
```

OUTPUT:

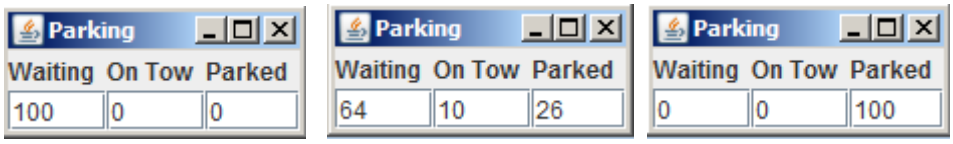
```
Here they go!...
t1-Red-2s
t2- Dead -2s
t1-Green-5s
Main - Main done - 5s
```

```
Here they go!...
t2- Dead -2s
t1-Red-2s
t1-Green-5s
Main - Main done - 5s
```

4. [6 valores]

Simule o seguinte cenário: Há um conjunto de (100) aviões (`Plane`) parados à espera de serem estacionados (uma fila de aviões na classe Airport) e um conjunto de (10) reboques (`Tow`) que irão em paralelo iniciar o estacionamento de aviões num estacionamento limitado a um número fixo de lugares (`ParkingSpot`).

Esta evolução é acompanhada em tempo real em três campos de texto de uma pequena aplicação:



a) b) c)

Fig. 1 Imagens do estado do sistema no início (com 100 aviões por estacionar) a meio do processo, e no final com todos os aviões estacionados.

No final deve aparecer na consola a mensagem "*All planes parked*", escrita pelo método `run()` da classe aeroporto, depois de todos os reboques terem terminado a sua execução com sucesso.

Se, a meio do processo, um reboque detectar que há aviões por estacionar e não há lugares para os estacionar, este deve interromper a espera do aeroporto pela finalização da tarefa dos reboques e o aeroporto deve escrever na consola: "*Not all planes parked*" e terminar, deixando os restantes reboques estacionar os aviões que estão a rebocar, mas sem esperar por estes nem escrever a mensagem "*All planes parked*".

Defina, tendo em atenção que a classe que determina o comportamento dos reboques (`Tow`) é um `Thread` (deve correr em paralelo) e irá estar a aceder a recursos partilhados:

1. A classe GUI que define a interface gráfica
2. O método `run()` da classe `Airport` (bem como todos os métodos que não estejam definidos e forem necessários ao funcionamento das outras classes)
3. A classe `Tow` que está continuamente a tirar um avião da lista aviões em espera e a colocá-lo no parque (use um `sleep()` aleatório entre 1 a 2 segundos para simular o tempo de parqueamento).

Considere como base de trabalho o seguinte código da classe `Airport` (não precisa repetir no exame este código):

```
public class Airport extends Thread {
        private static final int N_TOWS = 10;
        private static final int N_PLANES = 100;
        private static final int N_PARKINGSPOTS = ...;

        private Queue<Plane> planes = new LinkedList<Plane>();
        private List<Thread> tows = new LinkedList<Thread>();
        private List<ParkingSpot> parkingSpots = new LinkedList<ParkingSpot>();
        private GUI gui;

        public Airport() {
                for (int i = 0; i != N_TOWS; i++) {
                        Tow t = new Tow(this);
                        tows.add(t);
                }
                for (int i = 0; i != N_PLANES; i++) {
                        planes.add(new Plane(i));
                }
                for (int i = 0; i != N_PARKINGSPOTS; i++) {
                        parkingSpots.add(new ParkingSpot(i));
                }
        }

        public void run() {
                // …
        }
        public static void main(String[] args) {
                new Airport().start();
        }
        // …
}
public class Plane {
        private String id;
        public Plane(int i) {
                id = String.valueOf(i);
        }
        public String id() { return id; }
}

public class ParkingSpot {
        private String id;
        public ParkingSpot(int i) {
                id = String.valueOf(i);
        }
        public String id() { return id; }
}
```

SOLUÇÃO:

```
public void run() {
        gui.go();
        for (Thread t : tows) {
                t.start();
        }
        try {
                for (Thread t : tows) {
                        t.join();
                }
                System.out.println("All planes parked");
        } catch (InterruptedException e) {
                System.out.println("Not all Planes parked, planes left " + planes.size());
```

```
        }
}
public synchronized void towing() {
        gui.towing();
}
public synchronized void parked() {
        gui.parked();
}
public synchronized Plane getPlane() {
        return planes.poll();
}
public synchronized ParkingSpot getParkingSpot() {
        if (parkingSpots.isEmpty())
                return null;
        return parkingSpots.remove(0);
}

public class Tow extends Thread {
        private Airport airport;
        public Tow(Airport airport) {
                this.airport = airport;
        }

        @Override
        public void run() {
                Plane p = airport.getPlane();
                ParkingSpot s = airport.getParkingSpot();
                while (s != null && p != null) {
                        park(p, s);

                        s = airport.getParkingSpot();
                        p = airport.getPlane();
                }
                if (p != null) {
                        airport.interrupt();
                }
        }
        private void park(Plane p, ParkingSpot s) {
                airport.towing();
                try {
                        sleep((long) (1000 + Math.random() * 1000));
                } catch (InterruptedException e) {
                }
                airport.parked();
        }
}

public class GUI {

        private JFrame frame = new JFrame("Parking");
        private int waiting;
        private int towing;
        private int parked;
        private JTextField a;
        private JTextField b;
        private JTextField c;

        public GUI(int nPlanes) {
                waiting = nPlanes;
                towing = 0;
                parked = 0;

                JPanel g = new JPanel();

                g.setLayout(new GridLayout(2, 3));

                g.add(new JLabel("Waiting  "));
                g.add(new JLabel("On Tow  "));
                g.add(new JLabel("Parked  "));
```

```
                a = new JTextField("  ");
                b = new JTextField("  ");
                c = new JTextField("  ");

                g.add(a);
                g.add(b);
                g.add(c);

                frame.add(g);
        }

        public void go() {
                a.setText(String.valueOf(waiting));
                b.setText(String.valueOf(towing));
                c.setText(String.valueOf(parked));
                frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
                frame.pack();
                frame.setVisible(true);
                try {
                        Thread.sleep(3000);
                } catch (InterruptedException e) {
                }
        }

        public void towing() {
                waiting--;
                towing++;
                set();
        }
        public void parked() {
                parked++;
                towing--;
                set();
        }

        private void set() {// não seria necessário usar o SwingUtilities.invokeLater
                SwingUtilities.invokeLater(new Runnable() {
                        @Override
                        public void run() {
                                a.setText(String.valueOf(waiting));
                                b.setText(String.valueOf(towing));
                                c.setText(String.valueOf(parked));
                        }
                });

        }
}
```

5. [6 valores]

Desenvolva uma classe, StringBlockingBuffer, cujos objetos representam um *buffer* de caracteres (String) que suporta acessos concorrentes. As *threads* que manipulam o *buffer* podem acrescentar e consumir sequências de caracteres. Tenha em conta as seguintes características:

1. O *buffer* está vazio no momento em que é criado (i.e. sem caracteres, equivalente à String vazia).
    a. Existe uma operação (getSize) que devolve o tamanho do *buffer* (equivalente ao comprimento da String guardada).
2. Existe uma operação de leitura (read) onde é passado como argumento o tamanho da sequência de carateres a ler (*n*), os quais ao serem lidos são removidos do *buffer* e devolvidos numa String. Os caracteres a consumir são os que foram adicionados há mais tempo.
    a. Caso não existam pelo menos *n* caracteres no *buffer* a chamada bloqueia, retornando só quando houverem caracteres disponíveis suficientes para consumir.
    b. Caso a *thread* que está bloqueada na chamada seja interrompida, a operação é abortada e a exceção InterruptedException é lançada (cabe à classe que invoca tratá-la).
    c. Existe uma operação para saber o número total de caracteres que estão pendentes para leitura. Este valor corresponde ao somatório de todos os valores *n* passados pelas *threads* que se encontram em espera.
3. Existe uma operação (append) para acrescentar caracteres aos existentes no *buffer* passando uma String. Esta operação não deverá ser bloqueante. Ao serem acrescentados carateres, outras *threads* em espera podem desbloquear caso passe a haver um número de caracteres suficientes para satisfazer o(s) pedido(s) de leitura.

```
public class StringBlockingBuffer {

   private String content;
   private int pendingReads;

   public StringBlockingBuffer() {
        content = "";
        pendingReads = 0;
   }

   public synchronized int getContentLength() {
        return content.length();
   }

   public synchronized String read(int n) throws InterruptedException {
        pendingReads += n;
        while(content.length() < n) {
               try {
                      wait();
               } catch (InterruptedException e) {
                      pendingReads -= n;
                      throw e;
               }
        }
        String s = content.substring(0, n);
        content = content.substring(n);
        pendingReads -= n;
        return s;
   }

   public synchronized int getPendingReads() {
        return pendingReads;
   }

   public synchronized void append(String s) {
        content += s;
        notifyAll();
   }
}
```

6. [4 valores]

Considere um sistema de gestão de uma empresa de carros de aluguer. Para poder planear a atribuição de serviços, a empresa necessita de, diariamente, conhecer a duração dos turnos de cada um dos motoristas que lhe prestam serviço.
Cada motorista tem assim à sua disposição uma pequena aplicação que, todos os dias, transmite ao servidor os dados relativos às suas horas de trabalho.
Implemente a classe relativas **ao servidor**, tomando em conta o seguinte:

- quando o cliente é ativado, deve ligar-se ao servidor, estabelecendo um canal de escrita, onde os dados serão transmitidos em forma de objetos (ObjectOutputStream).
- imediatamente, o cliente deve enviar ao servidor uma ou mais instâncias da classe *Turno*, caracterizada pela hora de início e de fim (os minutos não são considerados). Para exemplificar o funcionamento, considere o caso de um motorista que tem três turnos: das 6:00 às 9:00, das 11:00 às 16:00 e das 20:00 às 22:00.
- quando o servidor recebe a informação, deve colocá-la num atributo que será uma lista de *Turnos*, bem como exibir a informação, para controle, na consola.
- o servidor deve exibir a máxima disponibilidade, aceitando novas ligações mesmo quando esteja, eventualmente, a receber dados de outros clientes.

Assuma que está definida a classe Turno. É necessário definir o método main(), bem como todos os outros relevantes.

```
public class Servidor {

   public final static int port =8090;
   private ServerSocket server;

   public void init() throws IOException {
        server =new ServerSocket(port);
   }
   private List<Turno>turnos=new ArrayList<>();

   public void serve() throws IOException{

        while(true){
                Socket s=server.accept();
                new TrataCliente(s.getInputStream()).start();
        }
   }

   public synchronized void adicionaTurno(Turno t){
        turnos.add(t);
   }

   public static void main(String[] args){
        final Servidor s = new Servidor();
        try {
                s.init();
                s.serve();
        } catch (IOException e) {
                // TODO Auto-generated catch block
                e.printStackTrace();
        }
   }

   public class TrataCliente extends Thread {
        private ObjectInputStream in;

        public TrataCliente(InputStream in) throws IOException {
                super();
                this.in = new ObjectInputStream(in);
```

```
        }

        @Override
        public void run() {
                try {
                        while(true){
                                Turno t=(Turno)in.readObject();
                                adicionaTurno(t);
                                System.out.println("Recebido:"+t);
                        }
                } catch (ClassNotFoundException e) {
                } catch (IOException e) {
                        // Não fazer nada... Leitura acabou
                        System.out.println("Cliente desligou-se.");
                } finally{
                        try {
                                in.close();
                        } catch (IOException e) {
                        }
                }
        }
   }
}
```