Nome:

. . . . . .

Número:

☐0	☐0	☐0	☐0	☐0
☐1	☐1	☐1	☐1	☐1
☐2	☐2	☐2	☐2	☐2
☐3	☐3	☐3	☐3	☐3
☐4	☐4	☐4	☐4	☐4
☐5	☐5	☐5	☐5	☐5
☐6	☐6	☐6	☐6	☐6
☐7	☐7	☐7	☐7	☐7
☐8	☐8	☐8	☐8	☐8
☐9	☐9	☐9	☐9	☐9

ENGENHARIA INFORMÁTICA – UNIVERSIDADE DO MINHO

Exame de Sistemas Distribuídos

9 de fevereiro de 2022 – Duração: 2h00

Instruções: *Preencha o nome e o número de aluno nesta folha pintando completamente as caixas correspondentes a cada algarismo; em cada pergunta de escolha múltipla há sempre uma ou mais respostas certas; para as assinalar pinte completamente as caixas correspondentes; não use as áreas sombreadas; preencha também o nome e número em cada folha de exame adicional.*

Grupo I

Responda a este grupo no próprio enunciado.

1. Considere um *lock* implementado com o algoritmo de Peterson para exclusão mútua entre *threads*. É verdade que:

- ☐ se usado por apenas um *thread*, ele nunca conseguirá entrar na secção crítica
- ☐ se usado por três *threads*, um deles nunca conseguirá entrar na secção crítica
- ☐ não pode ser usado com mais do que dois *threads* porque poderiam entrar vários simultaneamente na secção crítica
- ☐ precisa de designar um *thread* como "vítima" para evitar impasses (*deadlock*)

2. Considere um sistema que permite inscrições em turnos da uma UC segundo o modelo cliente/servidor. Numa sessão entre eles espera encontrar mensagens

- ☐ semelhantes enviadas n vezes do servidor para o cliente, cada uma com uma string descrevendo cada turno (dia e hora)
- ☐ do cliente para o servidor indicando para cada um dos n turnos o número atualizado de alunos inscritos
- ☐ do cliente para o servidor pedindo para adquirir um *lock* para evitar corridas durante a inscrição
- ☐ do servidor para o cliente, contendo cada uma o número n de turnos disponíveis e n pares (dia,hora), todos como números inteiros

3. Considere um modelo de concorrência *1-thread-por-pedido* na implementação de um servidor para um cliente *multi-threaded*. Este modelo:

- ☐ exige que cada resposta seja etiquetada, referindo o pedido correspondente
- ☐ permite que multiplos *threads* num mesmo cliente façam progresso concorrentemente em invocações ao servidor
- ☐ dispensa a utilização de *locks* na lógica da aplicação com estado partilhado do servidor
- ☐ dispensa a utilização de *locks* na manipulação do estado de sessão no servidor

4. Um sistema de resolução de nomes hierárquico como o DNS:

- ☐ é adequada a um sistema administrativamente descentralizado
- ☐ pode usar resolução recursiva para reduzir a carga no nó da raiz
- ☐ permite obter uma resposta consultando menos nós que uma DHT Chord de dimensão semelhante
- ☐ pode usar resolução iterativa, mas tende a aumentar a latência

5. Considere um sistema de partilha e distribuição de ficheiros como o BitTorrent, em que um grande número de participantes espalhados pelo globo procuram e descarregam conteúdos de grande dimensão. Identifique aquele que lhe parece ser o problema de sistemas distribuídos mais importante e identifique uma solução típica. Discuta a adequação dessa solução a casos extremos de ficheiros impopulares (com apenas um pequeno número de cópias existentes) e extremamente populares (em que pode haver milhares de tentativas simultâneas de pesquisa e descarregamento).

☐0 ☐.1 ☐.2 ☐.3 ☐.4 ☐.5 ☐.6 ☐.7 ☐.8 ☐.9 ☐1 *correção*

Grupo II *Responda a cada pergunta deste grupo numa folha de exame separada.*

Considere um sistema cliente/servidor de apoio à gestão de uma sala de reuniões de uma associação de estudantes. Diferentes listas candidatas a uma futura eleição (assuma números de lista, de 1 a L) podem aparecer para reuniões informais, sem marcação prévia, em que os membros podem chegar ou abandonar a reunião a qualquer momento. Cada reunião, em que todos os membros são da mesma lista, tem a garantia de poder ocupar a sala sem ser interrompida por membros de outras listas, que terão que esperar até que a sala esteja livre.

6. Apresente uma classe Java (para ser usada no servidor) que implemente a interface abaixo, tendo em conta que os seus métodos serão invocados num ambiente *multi-threaded*.

```
interface Reunião {
  void participa(int lista);
  void abandona(int lista);
  int naSala();
  int aEspera();
}
```

O método `participa` deve bloquear até o membro poder entrar na sala; o método `abandona` marca o abandono da sala pelo membro; os métodos `naSala` e `aEspera` devolvem o número de membros atualmente na sala e à espera de entrar, respetivamente.

Valorização: Não assuma que o valor de L é previamente conhecido. Quando a sala ficar vazia, dê preferência à lista com mais pessoas à espera. Minimize os *threads* que são acordados.

7. Considere um serviço ao qual clientes se ligam por TCP, para participarem numa reunião. Ao chegar, um cliente envia o seu número de lista, devendo o servidor responder com `ENTRE` quando a sala estiver disponível. O cliente poderá então enviar `ABANDONEI`, indicando que abandonou a sala.

Implemente só o programa servidor usando *threads*, *sockets* TCP, e a classe desenvolvida na pergunta anterior. Use um protocolo o mais simples possível, por exemplo, baseado em linhas de texto.

Valorização: Permita que, enquanto está à espera de entrar na sala, um cliente pergunte quantos participantes ainda há na reunião atual e quantos estão à espera.