# UNIVERSIDADE FEDERAL DE VIÇOSA

## INF310 – PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

### Lista de Exercícios 3

---

## QUESTÕES ABERTAS

**1.** Explique detalhadamente como a operação wait (ou Down) de um semáforo garante atomicidade mesmo quando múltiplas threads tentam decrementar o contador simultaneamente. Quais estruturas de baixo nível são utilizadas para implementar essa atomicidade?

**2.** Monitores utilizam variáveis de condição para coordenar threads. Explique a diferença entre os modelos de sinalização "signal-and-wait" e "signal-and-continue", incluindo as vantagens e desvantagens de cada abordagem.

**3.** Por que semáforos binários não são suficientes para resolver todos os problemas de sincronização que semáforos contadores podem resolver? Apresente um exemplo concreto que demonstre essa limitação.

**4.** Descreva o problema da inversão de prioridade que pode ocorrer em sistemas que utilizam monitores ou semáforos. Como protocolos de herança de prioridade podem mitigar esse problema?

**5.** Explique por que a operação spurious wakeup pode ocorrer quando threads aguardam em variáveis de condição e por que é necessário sempre verificar a condição em um loop ao invés de usar um simples if após o wait.

---

## QUESTÕES DE MÚLTIPLA ESCOLHA

**1.** Um semáforo inicializado com valor 3 sofre as seguintes operações na ordem: wait(), wait(), post(), wait(), wait(). Quantas threads ficam bloqueadas?

- a) 0
- b) 1
- c) 2
- d) 3

**2.** Em um monitor, quando uma thread chama notify() em uma variável de condição vazia (sem threads aguardando), o comportamento esperado é:

- a) A thread chamadora é bloqueada
- b) Uma exceção é lançada

- c) A notificação é perdida
- d) A notificação é enfileirada para a próxima thread

**3.** Qual estrutura de sincronização é mais adequada para implementar um pool de recursos com quantidade fixa?

- a) Mutex
- b) Semáforo contador
- c) Semáforo binário
- d) Barreira

**4.** A principal vantagem de monitores sobre semáforos é:

- a) Maior eficiência computacional
- b) Encapsulamento e menor propensão a erros
- c) Suporte nativo a prioridades
- d) Menor overhead de contexto

**5.** Um semáforo com valor inicial 0 é usado para:

- a) Exclusão mútua
- b) Sincronização de eventos
- c) Contagem de recursos
- d) Gerenciamento de memória

**6.** Em C++, por que std::condition_variable requer std::unique_lock ao invés de std::lock_guard?

- a) std::lock_guard não suporta múltiplas threads
- b) std::unique_lock permite unlock temporário durante wait
- c) std::lock_guard é mais lento
- d) std::unique_lock tem menor overhead

**7.** Qual operação NÃO é atômica por natureza e requer sincronização?

- a) Leitura de um bool
- b) Incremento de um inteiro
- c) Atribuição de um ponteiro
- d) Todas as anteriores podem requerer sincronização

**8.** O problema do jantar dos filósofos pode ser resolvido eficientemente com:

- a) Um mutex global

- b) Semáforos por filósofo e ordenação de recursos
- c) Apenas variáveis atômicas
- d) Nenhuma sincronização necessária

**9.** Quando múltiplas threads aguardam em uma variável de condição e notify_all() é chamado:

- a) Apenas uma thread é acordada
- b) Todas as threads são acordadas mas apenas uma adquire o lock
- c) Todas as threads executam simultaneamente
- d) A operação causa deadlock

**10.** Um semáforo binário difere de um mutex principalmente porque:

- a) Semáforo binário não tem proprietário
- b) Mutex é mais rápido
- c) Semáforo binário não pode ser usado para exclusão mútua
- d) Mutex não é thread-safe

**11.** O padrão produtor-consumidor com buffer limitado requer, no mínimo:

- a) 1 semáforo
- b) 2 semáforos
- c) 3 semáforos
- d) 4 semáforos

**12.** Em um monitor, o lock implícito é liberado quando:

- a) Uma thread chama wait() em uma variável de condição
- b) Uma thread termina a execução de um método do monitor
- c) Uma thread chama notify()
- d) Alternativas a e b estão corretas

**13.** Se uma thread executa post() em um semáforo com 5 threads bloqueadas, qual thread é acordada?

- a) A primeira que chamou wait()
- b) A última que chamou wait()
- c) Uma thread aleatória
- d) Depende da implementação e política de escalonamento

**14.** Qual problema clássico de sincronização modela compartilhamento de recursos com leitura concorrente mas escrita exclusiva?

- a) Produtor-consumidor
- b) Leitores-escritores
- c) Jantar dos filósofos
- d) Barbeiro dorminhoco

**15.** A inversão de prioridade ocorre quando:

- a) Uma thread de baixa prioridade executa antes de uma de alta prioridade
- b) Todas as threads têm a mesma prioridade
- c) O escalonador falha
- d) Semáforos são usados incorretamente

**16.** Para garantir que no máximo N threads executem uma seção crítica simultaneamente, deve-se usar:

- a) Mutex
- b) Semáforo contador inicializado com N
- c) Semáforo binário
- d) Variável de condição

**17.** O problema de starvation pode ocorrer quando:

- a) Threads são sempre acordadas em ordem FIFO
- b) Não há política justa de acesso a recursos
- c) Usa-se apenas mutexes
- d) Todas as threads têm mesma prioridade

**18.** Qual operação em um monitor é equivalente a post() em semáforo?

- a) wait()
- b) notify()
- c) lock()
- d) unlock()

**19.** Um sistema com 3 threads executando wait() em um semáforo inicializado com 2 resultará em:

- a) Deadlock
- b) 1 thread bloqueada

- c) Todas executam simultaneamente
- d) Erro de compilação

**20.** A principal diferença entre notify() e notify_all() é:

- a) notify() é mais rápido
- b) notify_all() acorda todas as threads aguardando
- c) notify() causa deadlock
- d) Não há diferença prática

---

## QUESTÕES DE IMPLEMENTAÇÃO

### Semáforos

**1.** Implemente um sistema de estacionamento com 5 vagas usando semáforos. Carros chegam e tentam estacionar; se não houver vaga, aguardam. Quando um carro sai, libera a vaga.

**2.** Implemente o problema dos leitores-escritores usando semáforos, onde múltiplos leitores podem acessar simultaneamente, mas escritores precisam de acesso exclusivo. Priorize leitores.

**3.** Um sistema de impressão tem 3 impressoras idênticas. Implemente usando semáforos a sincronização onde N threads (documentos) competem pelas impressoras. Cada impressão leva tempo variável.

**4.** Implemente uma barreira de sincronização para 4 threads usando apenas semáforos. Todas as threads devem chegar à barreira antes que qualquer uma possa prosseguir.

**5.** Um jogo multiplayer aguarda que exatamente 5 jogadores estejam prontos antes de iniciar uma partida. Implemente usando semáforos a sincronização entre a thread do servidor e as threads dos jogadores.

### Monitores

**6.** Implemente um monitor para gerenciar um buffer circular limitado (tamanho 10) no padrão produtor-consumidor. Múltiplos produtores e consumidores devem ser suportados.

**7.** Implemente um monitor que controla o acesso a um recurso compartilhado onde existem 3 níveis de prioridade. Threads de alta prioridade sempre devem ter preferência sobre as de baixa prioridade.

**8.** Um sistema de reserva de salas tem 2 salas disponíveis. Implemente um monitor onde threads tentam reservar uma sala por um tempo determinado. Se ambas estiverem ocupadas, a thread aguarda.

**9.** Implemente um monitor para o problema da ponte estreita: a ponte suporta tráfego em apenas uma direção por vez, mas múltiplos carros podem cruzar simultaneamente na mesma direção. Carros

nas duas direções competem pelo acesso.

**10.** Implemente um monitor que simula um sistema de pedágios com 3 cabines. Veículos chegam, pagam (processo que leva tempo), e liberam a cabine. Se todas as cabines estiverem ocupadas, veículos aguardam em fila.

---

### Observações:

- Para as questões de implementação, use C++ com threads padrão (std::thread)
- Para semáforos, pode usar std::counting_semaphore (C++20) ou implementar usando mutex e condition_variable
- Para monitores, use std::mutex e std::condition_variable
- Considere condições de corrida e possibilidades de deadlock
- Todas as implementações devem ser thread-safe