Universidade Federal de Viçosa

Departamento de Informática

INF310 – Programação Concorrente e Distribuída

Prova 2 – Valor: 25 pontos

Nome: _____ **Matrícula:** _____

Questões Teóricas

1. (5 pts) Sobre sincronização e primitivas de concorrência, responda:

a) (2 pts) Explique o conceito de atomicidade em operações concorrentes. Por que operações como `contador++` não são atômicas e quais problemas isso pode causar em um ambiente multithreaded?

b) (1,5 pts) Compare as vantagens e desvantagens entre o uso de semáforos binários e mutexes para garantir exclusão mútua. Em que situações um seria preferível ao outro?

c) (1,5 pts) O que é uma race condition? Dê um exemplo prático de como ela pode ocorrer em um programa concorrente e explique por que é um problema crítico.

---

2. (5 pts) Sobre monitores e variáveis de condição:

a) (2 pts) Explique a diferença fundamental entre `sem_wait()` e `sem_post()` em semáforos versus `wait()` e `signal()` em variáveis de condição de monitores. Por que variáveis de condição sempre devem ser usadas dentro de um mutex?

b) (1,5 pts) O que significa "spurious wakeup" no contexto de variáveis de condição? Por que recomenda-se usar `while` ao invés de `if` ao verificar condições antes de um `wait()`?

c) (1,5 pts) Descreva o problema de inversão de prioridade em sistemas concorrentes. Como semáforos podem contribuir para esse problema?

---

3. (5 pts) Sobre deadlocks e sincronização:

a) (2 pts) Quais são as quatro condições necessárias para que um deadlock ocorra? Explique cada uma delas brevemente.

b) (1,5 pts) Considere um sistema onde múltiplas threads precisam adquirir dois recursos (A e B) em ordem diferente. Explique como isso pode causar deadlock e proponha uma estratégia simples para preveni-lo.

c) (1,5 pts) O que é starvation em sistemas concorrentes? Ela é diferente de deadlock? Justifique sua resposta com um exemplo.

Questões de Implementação - Monitores

4. (3 pts) Implemente um monitor para gerenciar um estacionamento com capacidade limitada.

O estacionamento tem **N vagas** (você pode assumir N=10). Veículos chegam e tentam entrar (função `entrar()`), e após algum tempo, saem (função `sair()`). Se o estacionamento estiver cheio, os veículos devem esperar até que haja vaga

disponível.

Implemente a classe `Estacionamento` utilizando monitores (mutex e variáveis de condição) para garantir sincronização correta. O código deve ser thread-safe.

Estrutura base:

cpp

```cpp
class Estacionamento {
private:
    // Seus atributos aqui
public:
    void entrar(int idVeiculo);
    void sair(int idVeiculo);
};
```

---

5. (3 pts) Implemente um monitor para o problema dos leitores-escritores com prioridade para escritores.

Múltiplas threads podem ler simultaneamente, mas apenas uma thread pode escrever por vez, e quando há um escritor escrevendo, nenhum leitor pode ler. **Prioridade para escritores:** se há escritores esperando, novos leitores não devem começar a ler.

Implemente as funções `iniciar_leitura()`, `terminar_leitura()`, `iniciar_escrita()` e `terminar_escrita()` usando monitores.

Estrutura base:

cpp

```cpp
class LeitorEscritor {
private:
    // Seus atributos aqui
public:
    void iniciar_leitura();
    void terminar_leitura();
    void iniciar_escrita();
    void terminar_escrita();
};
```

Questões de Implementação - Semáforos

6. (3 pts) Implemente uma barreira de sincronização usando semáforos.

Uma barreira é um ponto de sincronização onde **N threads** devem esperar até que todas as N threads tenham chegado antes de prosseguir. Implemente a classe `Barreira` com uma função `esperar()` que bloqueia a thread até que todas as N threads tenham chamado essa função.

Estrutura base:

cpp

```cpp
class Barreira {
private:
    int n;          // número total de threads
    int contador;   // contador de threads que chegaram
    // Seus semáforos aqui
public:
    Barreira(int numThreads);
    void esperar();
};
```

---

7. (3 pts) Implemente o problema do jantar dos filósofos usando semáforos, evitando deadlock.

Há 5 filósofos sentados em uma mesa circular com 5 garfos (um entre cada par de filósofos). Cada filósofo precisa de 2 garfos (esquerdo e direito) para comer. Implemente a solução usando semáforos de forma que **não ocorra deadlock**.

Você pode usar a estratégia de limitar o número de filósofos que podem tentar pegar garfos simultaneamente ou outra estratégia de sua escolha, mas deve explicar brevemente sua abordagem em comentários.

Estrutura base:

cpp

```cpp
sem_t garfos[5];
sem_t limite;  // se necessário

void filosofo(int id) {
    // Implementar: pensar, pegar garfos, comer, soltar garfos
}
```

Questões de Análise de Código

8. (3 pts) Analise o código abaixo que implementa um contador compartilhado:

cpp

```cpp
int contador_global = 0;
mutex mtx;

void incrementar(int vezes) {
    for (int i = 0; i < vezes; i++) {
        mtx.lock();
        int temp = contador_global;
        temp++;
        contador_global = temp;
        mtx.unlock();
    }
}

void decrementar(int vezes) {
    for (int i = 0; i < vezes; i++) {
        int temp = contador_global;
        mtx.lock();
        temp--;
        contador_global = temp;
        mtx.unlock();
    }
}

int main() {
    thread t1(incrementar, 1000);
    thread t2(decrementar, 1000);
    t1.join();
    t2.join();
    cout << "Contador final: " << contador_global << endl;
    return 0;
}
```

a) (1,5 pts) Identifique e explique todos os problemas de concorrência presentes no código acima.

b) (1,5 pts) Corrija o código para garantir que o resultado final seja sempre 0 (assumindo igual número de incrementos e decrementos).

---

9. (3 pts) Analise o código abaixo que implementa um buffer circular limitado:

cpp

```
#define TAM 5
int buffer[TAM];
int in = 0, out = 0, count = 0;
sem_t mutex, cheio, vazio;

void produtor() {
    while(true) {
        int item = produzir_item();
        sem_wait(&vazio);

        buffer[in] = item;
        in = (in + 1) % TAM;
        count++;

        sem_post(&mutex);
        sem_post(&cheio);
    }
}

void consumidor() {
    while(true) {
        sem_wait(&cheio);
        sem_wait(&mutex);

        int item = buffer[out];
        out = (out + 1) % TAM;
        count--;

        sem_post(&vazio);
        consumir_item(item);
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&cheio, 0, 0);
    sem_init(&vazio, 0, TAM);

    thread prod(produtor);
    thread cons(consumidor);
    prod.join();
    cons.join();
```

```
    return 0;
}
```

a) (2 pts) Identifique e explique os problemas de sincronização no código. Há possibilidade de race condition? Há possibilidade de deadlock? Justifique.

b) (1 pt) Corrija o código para funcionar corretamente com múltiplos produtores e consumidores.

Boa prova!