UNIVERSIDADE FEDERAL DE VIÇOSA

Departamento de Informática

INF310 – Programação Concorrente e Distribuída

Prova 3 – Valor: 25 pontos

Nome: _____ **Matrícula:** _____

QUESTÕES TEÓRICAS

Questão 1 (2 pontos)

Explique a diferença entre **busy waiting** (espera ocupada) e **bloqueio** no contexto de sincronização de threads. Compare o uso de semáforos com uma implementação que utilize apenas variáveis compartilhadas e loops de verificação. Qual abordagem é mais eficiente e por quê?

Questão 2 (2 pontos)

Em um monitor, quando uma thread chama `wait()` em uma variável de condição:

a) O que acontece com o mutex associado ao monitor?

b) Por que é necessário usar `while` ao invés de `if` ao verificar a condição antes do `wait()`?

c) Qual é a diferença entre `notify_one()` e `notify_all()` e quando cada um deve ser usado?

Questão 3 (2 pontos)

Considere um sistema onde há 4 threads tentando acessar 3 recursos compartilhados (R1, R2, R3). Explique:

a) O que é deadlock e quais são as 4 condições necessárias para que ele ocorra?

b) Se uma thread precisa adquirir múltiplos semáforos, qual estratégia simples pode prevenir deadlock?

c) Por que a ordem de aquisição de recursos é importante em programação concorrente?

QUESTÕES DE IMPLEMENTAÇÃO - MONITORES

Questão 4 (3 pontos)

Uma academia possui um vestiário com **N armários** numerados e **M chaves** (onde M < N). Para usar o vestiário, um aluno precisa:

1. Pegar uma chave (se houver disponível)
2. Escolher um armário livre
3. Usar o vestiário
4. Devolver a chave

As regras são:

- Se não há chaves disponíveis, o aluno deve esperar
- Se não há armários livres, o aluno deve esperar (mesmo tendo pego a chave)
- Um aluno só pode devolver a chave após usar o vestiário
- **Importante:** Alunos VIP (identificados por um flag) têm **prioridade** para pegar chaves quando elas são devolvidas

Implemente um monitor que sincronize o acesso ao vestiário. O monitor deve ter:

- `pegar_chave(bool is_vip, int id_aluno)`: bloqueia até conseguir chave
- `pegar_armario(int id_aluno)`: bloqueia até conseguir armário livre
- `devolver_chave_e_armario(int id_aluno)`: libera chave e armário

Esboço das threads:

cpp

```
aluno:
loop {
    vestiario.pegar_chave(is_vip, id);
    vestiario.pegar_armario(id);
    usar_vestiario();
    vestiario.devolver_chave_e_armario(id);
}
```

Questão 5 (3 pontos)

Um sistema de impressão compartilhada possui:

- **3 impressoras** coloridas (caras, recurso limitado)
- **5 impressoras** preto-e-branco (mais comuns)

Usuários chegam com dois tipos de trabalhos:

- **Trabalho Premium**: requer impressora colorida (apenas)
- **Trabalho Normal**: aceita qualquer impressora, mas **prefere** colorida se disponível

As regras são:

- Se há impressora colorida livre e há trabalhos Premium esperando, priorizar Premium
- Trabalhos Normais só podem usar colorida se não há Premium esperando
- Trabalhos Normais sempre podem usar preto-e-branco
- Após impressão, impressora deve ser liberada

Implemente um monitor com:

- `imprimir_premium(int id_trabalho)`: bloqueia até conseguir impressora colorida
- `imprimir_normal(int id_trabalho)`: tenta colorida, senão usa P&B
- `liberar_impressora(int id_trabalho, tipo_impressora)`: libera recurso

Esboço das threads:

cpp

```
trabalho_premium:
loop {
    sistema.imprimir_premium(id);
    // imprime (demora)
    sistema.liberar_impressora(id, COLORIDA);
}

trabalho_normal:
loop {
    sistema.imprimir_normal(id);
    // imprime (demora)
    sistema.liberar_impressora(id, tipo_usada);
}
```

QUESTÕES DE IMPLEMENTAÇÃO - SEMÁFOROS

Questão 6 (3 pontos)

Um sistema de controle de tráfego aéreo gerencia o pouso e decolagem de aviões em um aeroporto com as seguintes restrições:

- Há **1 pista única** que pode ser usada para pouso OU decolagem
- No máximo **2 aviões** podem estar na "fila de espera" próxima à pista (área de taxiamento)
- **Prioridade:** Aviões pousando têm prioridade absoluta sobre aviões decolando (questão de segurança - combustível)
- Um avião decolando deve esperar se há aviões querendo pousar
- Após pousar, o avião deve sair da pista para liberar para o próximo

Implemente usando semáforos a sincronização entre threads representando aviões pousando e decolando.

Garanta que:

- No máximo 1 avião usa a pista por vez
- No máximo 2 aviões no taxiamento
- Aviões pousando têm prioridade
- Não há starvation para decolagens (quando não há pousos esperando)

Esboço das threads:

cpp

```
aviao_pousando:
    // solicitar pouso (prioridade!)
    // entrar na área de taxiamento (max 2)
    // usar pista para pousar
    // liberar pista
    // sair do taxiamento

aviao_decolando:
    // solicitar decolagem (baixa prioridade)
    // entrar na área de taxiamento (max 2)
    // usar pista para decolar
    // liberar pista
    // sair do taxiamento
```

Dica: Use semáforos para contar aviões querendo pousar e controlar prioridade.

Questão 7 (3 pontos)

Uma fábrica possui uma linha de montagem com o seguinte processo:

1. **Estação A**: 3 robôs podem trabalhar simultaneamente (preparação de peças)
2. **Estação B**: Apenas 1 robô por vez (montagem final, gargalo)
3. **Estação C**: 2 robôs podem trabalhar simultaneamente (controle de qualidade)

Um produto deve passar pelas 3 estações **nesta ordem obrigatória**: A → B → C.

Cada robô (thread) processa produtos continuamente. Quando termina uma estação, imediatamente tenta entrar na próxima.

Implemente usando semáforos o controle de acesso às estações. Garanta que:

- No máximo 3 robôs em A, 1 em B, 2 em C simultaneamente
- Um robô só avança para a próxima estação quando há espaço
- Não há deadlock entre as estações

Esboço da thread robô:

cpp

```
robo:
loop {
   // entrar estação A (max 3)
   processar_em_A();
   // sair de A, entrar em B (max 1)
   processar_em_B();
   // sair de B, entrar em C (max 2)
   processar_em_C();
   // sair de C
}
```

Desafio: Pense na ordem de liberação e aquisição de semáforos para evitar deadlock!

QUESTÕES DE ANÁLISE DE CÓDIGO

Questão 8 (2 pontos)

Considere o código abaixo proposto como solução para um sistema de controle de acesso a uma sala de servidores que suporta no máximo 5 técnicos simultaneamente:

cpp

```
/* variáveis globais */
int dentro = 0;
sem_t mutex, acesso;
// inicializados com 0

void tecnico() {
    sem_wait(&mutex);
    if (dentro < 5) {
        dentro++;
        sem_post(&mutex);
        sem_post(&acesso);

        // trabalha na sala de servidores

        sem_wait(&acesso);
        sem_wait(&mutex);
        dentro--;
        sem_post(&mutex);
    } else {
        sem_post(&mutex);
        // vai embora
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    sem_init(&acesso, 0, 0);
    // criar threads técnicos
}
```

Analise o código e responda:

a) Identifique e explique **pelo menos 2 problemas** de sincronização neste código.

b) O código garante que no máximo 5 técnicos entram? Justifique.

c) Há possibilidade de deadlock? Se sim, descreva o cenário.

Questão 9 (2 pontos)

Analise o seguinte código que implementa um buffer limitado com produtor e consumidor:

cpp

```c
#define N 10
int buffer[N];
int count = 0;
sem_t mutex;

void produtor() {
    while(true) {
        int item = produzir();

        if (count < N) {
            sem_wait(&mutex);
            buffer[count] = item;
            count++;
            sem_post(&mutex);
        }
    }
}

void consumidor() {
    while(true) {
        int item;
        if (count > 0) {
            sem_wait(&mutex);
            count--;
            item = buffer[count];
            sem_post(&mutex);

            consumir(item);
        }
    }
}

int main() {
    sem_init(&mutex, 0, 1);
    // criar threads
}
```

Analise e responda:

a) Identifique o principal problema de race condition neste código.

b) Além do problema de race condition, há outro problema de sincronização. Qual é?

c) Como este código deveria ser corrigido? (descreva a solução, não precisa implementar)

Questão 10 (2 pontos)

Para sincronizar o acesso de múltiplas threads a uma impressora compartilhada, um estudante implementou o seguinte código usando monitor:

cpp

```cpp
class Impressora {
private:
    bool ocupada = false;
    mutex mtx;
    condition_variable cv;

public:
    void imprimir(string documento) {
        unique_lock<mutex> lock(mtx);

        if (ocupada) {
            cv.wait(lock);
        }

        ocupada = true;
        lock.unlock();

        // imprime documento (demora 5 segundos)
        cout << "Imprimindo: " << documento << endl;
        sleep(5);

        lock.lock();
        ocupada = false;
        cv.notify_one();
    }
};
```

Analise e responda:

a) Há algum problema com o uso de `if` ao invés de `while` antes do `wait()`? Justifique.

b) O código garante exclusão mútua durante a impressão? Por quê?

c) Se 10 threads chamarem `imprimir()` simultaneamente, o que pode acontecer?

Questão 11 (3 pontos)

Para descobrir a ordem de execução das threads em um sistema, um estudante executou um programa e obteve a seguinte saída no console:

```
Thread 5 liberada. Thread 4 esperada
Thread 6 liberada. Thread 5 esperada
Thread 7 liberada. Thread 6 esperada
Thread 8 liberada. Thread 7 esperada
Thread 4 liberada. Thread 8 esperada
```

O código fonte relevante é mostrado abaixo (incompleto):

cpp

```cpp
sem_t s1, s2;
int v = 0;

void func(int id) {
    sem_post(&s1);
    sem_wait(&s2);
    if (id != v)
        printf("Thread %d liberada. Thread %d esperada\n", id, v);
    v++;
    sem_post(&s2);
}

int main() {
    sem_init(&s1, 0, 0);
    sem_init(&s2, 0, ?);  // valor inicial oculto

    vector<thread> threads;
    for (int i = 0; i < 10; i++) {
        thread t(func, i);
        threads.push_back(move(t));
    }

    sem_wait(&s1);
    // ... resto do main
}
```

Com base na saída mostrada, responda:

a) Qual é o valor inicial do semáforo `s2`? Justifique baseado na saída.

b) Explique o papel de cada semáforo (`s1` e `s2`) neste código.

c) Por que a ordem de saída não é sequencial (4, 5, 6, 7, 8)?

Questão 12 (3 pontos)

Um monitor foi implementado para controlar o acesso a recursos compartilhados:

cpp

```cpp
class RecursoCompartilhado {
private:
    int id_atual = -1;
    bool acessando = false;
    mutex mtx;
    condition_variable pode_acessar;
    condition_variable terminou;

public:
    void acessar(int id) {
        unique_lock<mutex> lock(mtx);

        while (acessando) {
            pode_acessar.wait(lock);
        }

        id_atual = id;
        acessando = true;
    }

    void liberar() {
        unique_lock<mutex> lock(mtx);

        acessando = false;
        id_atual = -1;

        pode_acessar.notify_all();
        terminou.notify_one();
    }

    void esperar_recurso_livre() {
        unique_lock<mutex> lock(mtx);

        while (acessando) {
            terminou.wait(lock);
        }
    }
};
```

Considere o seguinte uso:

cpp

```
RecursoCompartilhado recurso;

// Thread A
recurso.acessar(1);
// trabalha por 10 segundos
recurso.liberar();

// Thread B (executando simultaneamente)
recurso.esperar_recurso_livre();
cout << "Recurso ficou livre!" << endl;
```

Analise e responda:

a) Qual é o propósito da função `esperar_recurso_livre()` e por que ela usa uma condition_variable separada?

b) Por que `liberar()` usa `notify_all()` em `pode_acessar` mas `notify_one()` em `terminou`?

c) Há algum problema de concorrência se múltiplas threads chamarem `acessar()` simultaneamente? Justifique.

BOA PROVA!

Observações:

- Justifique todas as suas respostas
- Código sem explicação não terá pontuação completa
- Considere sempre: race conditions, deadlocks e starvation
- Boa organização e clareza também são avaliadas