UNIVERSIDADE FEDERAL DE VIÇOSA

INF310 – PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

Lista de Exercícios 3 - Problemas Contextualizados

Questões com Monitores

Questão 1 (Monitor): Estacionamento Inteligente

Um estacionamento possui **N vagas numeradas** (1 a N) e um sistema automatizado de controle de acesso. Motoristas chegam e desejam estacionar, mas há regras especiais:

- **Motoristas VIP** podem estacionar em qualquer vaga disponível
- **Motoristas Regular** só podem estacionar nas vagas com numeração **ímpar**
- **Motoristas Comum** só podem estacionar nas vagas com numeração **par**

Quando um motorista chega e não há vaga apropriada disponível, ele deve esperar. Quando ele sai, a vaga é liberada para outros motoristas.

Implemente um monitor que sincronize as threads representando os motoristas. O monitor deve garantir:

- Não há mais de N carros estacionados simultaneamente
- Motoristas só ocupam vagas permitidas para sua categoria
- Não há desperdício: se há vaga apropriada, algum motorista esperando deve ser notificado
- As funções `estacionar(tipo, id_motorista)` e `sair(id_motorista)` devem retornar apenas quando concluídas

Esboço das threads:

cpp

```
motorista tipo T:
loop {
  estacionamento.estacionar(T, id);
  // usa o estacionamento por um tempo
  estacionamento.sair(id);
  // vai embora, volta depois
}
```

Questão 2 (Monitor): Sala de Reunião Compartilhada

Uma empresa possui uma sala de reunião que pode ser usada de duas formas:

- **Modo Individual**: apenas 1 pessoa usa a sala (apresentações importantes)
- **Modo Grupo**: até 5 pessoas podem usar simultaneamente (reuniões de equipe)

As regras são:

- Se alguém está usando no modo individual, ninguém mais pode entrar
- Se há pessoas no modo grupo, mais pessoas podem entrar (até o limite de 5)
- Modo individual tem **prioridade** sobre modo grupo
- Uma pessoa no modo grupo NÃO pode entrar se há alguém esperando para modo individual

Implemente um monitor que sincronize o acesso à sala. O monitor deve ter:

- `entrar_individual(id)`: bloqueia até conseguir acesso exclusivo
- `sair_individual(id)`: libera a sala
- `entrar_grupo(id)`: bloqueia até conseguir entrar no grupo (se não há individual esperando/usando)
- `sair_grupo(id)`: sai do grupo

Esboço das threads:

cpp

```
pessoa_individual:
loop {
    sala.entrar_individual(id);
    // usa a sala sozinho
    sala.sair_individual(id);
}

pessoa_grupo:
loop {
    sala.entrar_grupo(id);
    // participa da reunião em grupo
    sala.sair_grupo(id);
}
```

Questão 3 (Monitor): Ponte Estreita

Uma ponte pode suportar no máximo **K veículos** simultaneamente e permite tráfego em ambas as direções, mas **não ao mesmo tempo**. Ou seja:

- Múltiplos veículos do **Norte→Sul** podem atravessar juntos (até K)
- Múltiplos veículos do **Sul→Norte** podem atravessar juntos (até K)
- Veículos em direções opostas **NÃO** podem estar na ponte simultaneamente
- Quando a ponte está vazia, a próxima direção é a que tem veículos esperando há mais tempo

Implemente um monitor que controle o acesso à ponte:

- `entrar_norte_sul(id)`: veículo do norte quer atravessar para o sul
- `sair_norte_sul(id)`: veículo completou a travessia
- `entrar_sul_norte(id)`: veículo do sul quer atravessar para o norte
- `sair_sul_norte(id)`: veículo completou a travessia

Critérios:

- Não mais que K veículos na ponte
- Não misturar direções

- Evitar starvation (alternar direções quando possível)

Esboço das threads:

cpp

```
veiculo_norte:
loop {
    ponte.entrar_norte_sul(id);
    // atravessa a ponte (leva tempo)
    ponte.sair_norte_sul(id);
}

veiculo_sul:
loop {
    ponte.entrar_sul_norte(id);
    // atravessa a ponte (leva tempo)
    ponte.sair_sul_norte(id);
}
```

Questão 4 (Monitor): Sistema de Empréstimo de Livros

Uma biblioteca possui **M cópias** de um livro popular. Há dois tipos de usuários:

- **Professores**: podem emprestar o livro por tempo ilimitado e têm prioridade
- **Alunos**: podem emprestar, mas se um professor está esperando e não há cópias disponíveis, novos alunos não podem pegar o livro

As regras são:

- Se há cópias disponíveis, qualquer usuário pode emprestar
- Se não há cópias e há professores esperando, alunos devem esperar todos os professores serem atendidos primeiro
- Quando alguém devolve, deve-se priorizar professores esperando

Implemente um monitor com as seguintes funções:

- `emprestar_professor(id)`: professor tenta emprestar
- `devolver_professor(id)`: professor devolve
- `emprestar_aluno(id)`: aluno tenta emprestar
- `devolver_aluno(id)`: aluno devolve

Esboço das threads:

cpp

```
professor:
loop {
    biblioteca.emprestar_professor(id);
    // usa o livro
    biblioteca.devolver_professor(id);
}

aluno:
loop {
    biblioteca.emprestar_aluno(id);
    // usa o livro
    biblioteca.devolver_aluno(id);
}
```

Questões com Semáforos

Questão 5 (Semáforo): Laboratório de Química

Um laboratório de química possui:

- **3 bancadas** de trabalho
- **5 jalecos** de proteção
- **2 óculos de proteção**

Para realizar um experimento, um estudante precisa de:

- 1 bancada
- 1 jaleco
- 1 óculos

Depois de realizar o experimento, o estudante devolve todos os equipamentos.

Implemente usando semáforos a sincronização das threads que representam os estudantes. Garanta que:

- Não há mais recursos sendo usados do que os disponíveis
- Estudantes só realizam experimento quando conseguem TODOS os recursos
- Recursos são liberados após o uso

Esboço da thread estudante:

cpp

```
estudante:
loop {
    // pegar recursos (bancada, jaleco, óculos)
    realizar_experimento();
    // devolver recursos
}
```

Dica: Cuidado com a ordem de aquisição de recursos para evitar deadlock!

Questão 6 (Semáforo): Restaurante Self-Service

Um restaurante self-service tem um sistema de controle de acesso:

- A **área de buffet** suporta no máximo **10 clientes** simultaneamente
- Há **15 pratos** disponíveis para os clientes usarem
- Há **3 balanças** para pesar a comida

O processo do cliente é:

1. Entrar na área do buffet (se não estiver cheia)
2. Pegar um prato
3. Servir-se (pode demorar)
4. Usar uma balança para pesar
5. Devolver o prato
6. Sair da área do buffet

Implemente usando semáforos a sincronização. Garanta que:

- No máximo 10 clientes na área do buffet
- Clientes só pegam prato se houver disponível
- Clientes só pesam se houver balança disponível
- Todos os recursos são devidamente devolvidos

Esboço da thread cliente:


cpp

```
cliente:
loop {
    // entrar no buffet
    // pegar prato
    servir_se();
    // pesar na balança
    // devolver prato
    // sair do buffet
    comer();
}
```

Questão 7 (Semáforo): Parque de Diversões

Um parque de diversões tem um brinquedo radical com as seguintes características:

- O brinquedo funciona em **ciclos**: carrega passageiros, opera, descarrega
- Cada ciclo acomoda exatamente **N passageiros** (nem mais, nem menos)
- O operador deve esperar que exatamente N passageiros estejam prontos
- Os passageiros devem esperar que o brinquedo complete o ciclo para desembarcar
- Após desembarcar, novos passageiros podem embarcar para o próximo ciclo

Implemente usando semáforos a sincronização entre:

- 1 thread operador
- M threads passageiros (M > N)

O monitor deve garantir:

- Brinquedo só opera com exatamente N passageiros
- Passageiros não desembarcam antes do ciclo terminar
- Não há race condition no embarque/desembarque

Esboço das threads:

cpp

```
operador:
loop {
    // esperar N passageiros embarcarem
    operar_brinquedo();
    // liberar N passageiros para desembarcarem
}

passageiro:
loop {
    embarcar();
    // esperar o ciclo completar
    desembarcar();
    // descansar antes de voltar
}
```

Questão 8 (Semáforo): Centro de Vacinação

Um centro de vacinação tem o seguinte processo:

- Há **K cabines** de vacinação
- Há **1 sala de observação** com capacidade para **M pessoas**
- Processo: Cidadão recebe vacina em uma cabine → vai para sala de observação por 15 minutos → sai

Regras:

- Cidadão só entra na cabine se houver cabine livre

- Após vacina, cidadão só vai para observação se houver espaço
- Se sala de observação está cheia, cidadão espera NA CABINE (ocupando-a)
- Após 15 minutos de observação, cidadão libera o espaço

Implemente usando semáforos considerando:

- Múltiplas threads cidadãos
- 1 thread por cabine (simula o enfermeiro aplicando vacina)

Garanta que:

- No máximo K vacinações simultâneas
- No máximo M pessoas na sala de observação
- Cidadão não libera cabine se não conseguir entrar na observação

Esboço das threads:

cpp

```
cidadao:
// entrar na cabine
// receber vacina (enfermeiro aplica)
// sair da cabine e entrar na observação
observar_por_15_minutos();
// sair da observação

enfermeiro_cabine_i:
loop {
   // esperar cidadão na cabine i
   aplicar_vacina();
   // liberar cidadão
}
```

Dica: Este é o mais desafiador! Pense bem na ordem de aquisição dos semáforos.

Critérios de Avaliação

Para todas as questões, seu código será avaliado considerando:

1. **Corretude (40%)**
   - Não há deadlock
   - Não há race conditions
   - Sincronização está correta
2. **Completude (30%)**
   - Todas as regras do problema foram implementadas
   - Não há recursos sendo usados além do limite
   - Não há starvation evitável
3. **Qualidade do Código (20%)**
   - Código legível e bem comentado
   - Nomes de variáveis/semáforos significativos

    - Boa organização
4. **Testes (10%)**
    - Demonstra funcionamento com prints informativos
    - Testa cenários de concorrência

Dicas Gerais

Para Monitores:

- Use `mutex` para proteger todas as variáveis compartilhadas
- Use `condition_variable` para bloquear/acordar threads
- Sempre use predicados nos `wait()` para evitar spurious wakeups
- Pense em QUEM deve ser notificado quando algo acontece

Para Semáforos:

- Identifique todos os recursos compartilhados
- Um semáforo contador para cada tipo de recurso
- Cuidado com a ordem de `wait()` para evitar deadlock
- Todo `wait()` deve ter um `post()` correspondente
- Semáforos de sinalização começam em 0
- Semáforos de recursos começam com a quantidade disponível

Evitando Deadlock:

- **Ordem consistente**: Sempre adquira recursos na mesma ordem
- **Liberação antecipada**: Libere recursos que não está mais usando
- **Timeout**: Considere desistir após um tempo (em sistemas reais)
- **Teste exaustivamente**: Rode com muitas threads por muito tempo

---

Formato de Entrega

Para cada questão, entregue:

1. Código completo e compilável em C++
2. Breve explicação (comentário) da sua estratégia de sincronização
3. Exemplo de saída mostrando o funcionamento correto
4. Análise de por que sua solução evita deadlock

Compile com:

bash

```
g++ -std=c++17 -pthread questao_X.cpp -o questao_X
```

Teste com:

bash

```
./questao_X
# Deixe rodar por pelo menos 30 segundos
# Observe se há deadlock ou comportamento incorreto
```

Boa sorte e bom código! 🚀