Apostila de Exercícios — Programação Concorrente e Distribuída

Baseada na sua lista (INF310 – Lista 1).

> Observação: os enunciados abaixo não trazem gabarito — conforme solicitado. Cada sessão contém pelo menos 4 exercícios. Os trechos de código foram revisados para evitar erros óbvios de sintaxe e tornar os enunciados consistentes.

Sessão 1 — Saídas possíveis, grafos de precedência e aninhamento (mistura das questões 3, 4 e 5)

Instruções gerais: para cada item abaixo: (a) indique quais das saídas propostas são possíveis; (b) considere se o grafo de precedência é *propriamente aninhado*; (c) represente a relação de precedência usando a notação `S(...)` e `P(...)` quando aplicável; (d) desenhe (esboce) o grafo de precedência.

Questão 1.1

```
// Exemplo 1
void func(int id) {
    std::cout << "p" << id << " ";
}
int main() {
    std::thread t1(func,1);
    std::thread t2(func,2);
    func(3);
    t2.join();
    std::thread t4(func,4);
    t1.join();
    t4.join();
    func(5);
    std::cout << std::endl;
}
```

Lista de saídas candidatas (analise cada uma):

a) `p1 p3 p2 p4 p5`

b) `p2 p3 p4 p1 p5`

c) `p1 p2 p4 p3 p5`

d) `p3 p2 p1 p5 p4`

e) `ppp123  p4 p5` *(uma linha propositalmente estranha — trate-a como inválida e justifique por que)*

Questão 1.2

```
// Exemplo 2 — alteração da ordem de join
void func(int id) { std::cout << id; }
int main() {
    std::thread a(func,1);
    std::thread b(func,2);
    a.join();
    func(3);
    b.join();
    return 0;
}
```

Proponha 5 saídas possíveis. Para cada saída, explique porque é (im)possível e represente em `S` / `P`.

Questão 1.3 (fork)

```
int main() {
    if (fork() == 0) {
        std::cout << "1";
        if (fork() == 0)
            std::cout << "2";
        else
            std::cout << "3";
    }
    else
        std::cout << "4";
    if (fork() == 0)
        std::cout << "5";
    else
        std::cout << "6";
    return 0;
}
```

Dado o trecho acima, analise cuidadosamente e diga quais das seguintes sequências podem ser produzidas (justifique):

a) `123456`

b) `14235656`

c) `1234555666`

d) `4321565656`

e) `12233445566`

f) `1251264546135136`

Questão 1.4 (misto threads + fork)

```
// Exemplo 4
void f(int id) { std::cout << id; }
int main() {
    std::thread t(f,1);
    if (fork() == 0) {
        std::cout << 2;
    }
    t.join();
    std::cout << 3;
}
```

Liste saídas possíveis e desenhe o grafo de precedência (considere que `fork()` duplica o processo junto com as threads do momento do fork). Diga se o grafo resultante é propriamente aninhado.

Sessão 2 — Algoritmos de exclusão mútua para duas threads (baseada na questão 8)

Instruções gerais: para cada algoritmo abaixo, verifique/justifique se o algoritmo satisfaz os critérios: - Exclusão mútua (apenas uma thread na RC ao mesmo tempo) - Ausência de deadlock - Ausência de starvation - Fairness (justiça) - Ausência de atraso desnecessário (se apenas uma thread quer entrar, ela não deve ficar esperando sem necessidade)

Considere que `quer` é um array global iniciado com `false` e `vez` é uma variável global inteira inicializada adequadamente. As threads se identificam por `0` e `1`. Em suas análises, descreva cenários (sequências de passos) que comprovem as violações quando existirem.

Questão 2.1 — Código A (variação do enunciado original)

```
// Variáveis globais: bool quer[2] = {false,false}; int vez = 0;
// trecho executado por 'eu' (0 ou 1), com 'outro' = 1-eu
quer[eu] = true;
while (vez == outro) {
    while (quer[outro]);
    vez = eu;
}
// REGIÃO_CRÍTICA
vez = outro;
quer[eu] = false;
```

Analise: quais propriedades são satisfeitas/violadas? Explique passo a passo.

Questão 2.2 — Código B (variação do enunciado original)

```
quer[eu] = true;
while (quer[outro])
    ;
if (vez == outro) {
    quer[eu] = false;
    while (vez != eu);
    quer[eu] = true;
}
// REGIÃO_CRÍTICA
vez = outro;
quer[eu] = false;
```

Analise e discuta violação de princípios (se houver).

Questão 2.3 — Código C (Peterson)

```
// Peterson para 2 threads
bool flag[2] = {false,false};
int turn;

// trecho para 'i'
flag[i] = true;
turn = 1-i;
while (flag[1-i] && turn == 1-i) ;
// REGIÃO_CRÍTICA
flag[i] = false;
```

Mostre por que (ou por que não) este esquema garante os requisitos pedidos.

Questão 2.4 — Código D (turno simples, falho para algumas propriedades)

```
int vez = 0;
// para 'eu'
while (vez != eu) ;
// REGIÃO_CRÍTICA
vez = 1-eu;
```

Discuta: quais propriedades são atendidas/violadas? Explique cenários concretos que mostrem deadlock, starvation ou atraso desnecessário (se existirem).

Questão 2.5 — Proposta e correção Dado um dos algoritmos que você apontou como inválido, proponha uma modificação mínima (pseudocódigo) que o torne válido para duas threads. Justifique por

que sua alteração resolve os problemas detectados (não é necessário provar formalmente, mas justifique com cenários e argumentos).

Sessão 3 — Implementação de especificações S / P (semelhança às questões 10 e 11)

Instruções gerais: para cada especificação dada em termos de `S(...)` e `P(...)`, desenhe um esboço alto-nível do programa multithread em C++ (ou pseudocódigo) que satisfaça a especificação. Para os exercícios que exigem uso apenas de `join`, não use mutex, block/wakeup, etc.; para os exercícios que pedem block/wakeup, use `block()` e `wakeup(id)` explicitamente (ou explique como fariam com condition variables). Não é necessário código compilável, mas os esboços devem ser claros o suficiente para execução.

Questão 3.1 — (join-only) Especificação: `S(P(S(P(t1,t2),t3),t4),t5)`

- Escreva um esboço de implementação em C++ usando `std::thread` e `join` apenas. Indique claramente quais threads executam quais tarefas (uma mesma thread pode executar mais de uma tarefa, quando possível).

Questão 3.2 — (block/wakeup) Especificação: `S(P(S(T1, P(T2, T3)), T4), T5)`

- Escreva um esboço que utilize `block()` e `wakeup(id)` para implementar a precedência. Mostre as ações de bloqueio e acordar no lugar correto.

Questão 3.3 — (join-only) Especificação: `P(S(t1,t2), S(t3, P(t4, t5)))`

- Esboce o programa usando apenas `join`. Explique como você reutilizaria threads (quando aplicável) para evitar criar threads desnecessárias.

Questão 3.4 — (barreira / condition variables) Especificação: `S(P(t1,t2,t3), P(t4, S(t5,t6)))` — suponha que `P` com mais de dois argumentos seja paralelo entre todos os listados.

- Mostre um esboço que use *barreira* ou *condition variables* para sincronizar o ponto após `P(t1,t2,t3)` e antes de `P(t4, S(t5,t6))`.

Questão 3.5 — análise extra Para uma das implementações acima, descreva possíveis otimizações (reduzir número de threads, usar pools, evitar join desnecessários) e as trade-offs envolvidas.

Sessão 4 — Erros em código paralelo e correções sem alterar linhas existentes (baseada nas questões 12 e 13)

Instruções gerais: em cada exercício a seguir você recebe um trecho de código que será executado por várias threads. **Você não deve alterar as linhas já implementadas**; em vez disso, indique *exatamente* quais operações de sincronização (por exemplo: `mutex.lock()`, `mutex.unlock()`, `barrier.wait()`, `block()`, `wakeup(id)`, `condition_variable.wait(...)`, etc.) devem ser inseridas **em quais números de linha** para corrigir os problemas apontados. Justifique por que cada inserção corrige o problema. Nos casos em que exista mais de uma solução válida, indique pelo menos duas alternativas e discuta vantagens/desvantagens.

Questão 4.1 — Partição e redução (variação da questão 12)

```
1| void f(int tid, int num_threads, int *dados, int tam, int *res, int &n) {
2|     int ini = tid * tam / num_threads;
3|     int fim = (tid+1) * tam / num_threads;
4|     int soma = 0;
5|
6|     for (int i = ini; i < fim; i++) {
7|         soma += dados[i];
8|     }
9|
10|     res[n++] = soma; // soma de cada partição (não necessariamente
ordenado)
11|
12|     if (tid == 0) {
13|         n = 0;
14|         for (int i = 0; i < num_threads; i++) {
15|             n += res[i];
16|         }
17|     }
18|
19|     for (int i = ini; i < fim; i++) {
20|         dados[i] = n - dados[i]; // soma final dos dados menos o valor
original
21|     }
22| }
```

- Identifique todas as condições de corrida e inconsistências. Para cada ponto problemático, indique a operação (ou conjunto de operações) de sincronização a ser inserida e o número de linha *exato* onde inserir (p.ex. "antes de 10: mutex.lock(); depois de 10: mutex.unlock();"). - Proponha duas estratégias distintas: (A) usar mutex/locks e (B) usar barreira + um único responsable (thread 0) para calcular `n`. Discuta complexidade e desempenho de ambas.

Questão 4.2 — Escrita concorrente em arrays

```
1| void worker(int tid, int nthreads, int *arr, int len) {
2|     int start = tid * len / nthreads;
3|     int end = (tid+1) * len / nthreads;
4|     for (int i = start; i < end; ++i) {
5|         arr[i] = arr[i] + 1; // operação não-atomica sobre arr[i]
6|     }
7| }
```

- Mostre uma forma segura de sincronizar sem alterar as linhas 1–6 (ou seja, indicando onde aplicar locks/atomics). Discuta por que `arr[i] = arr[i] + 1` pode falhar em presença de concorrência e alternativas eficientes (atomics, chunk-local, etc.).

Questão 4.3 — Atualização de variável com restrições (variante da questão 13)

> Requisitos: $0 \leq x \leq 10$ sempre; qualquer thread pode atualizar x quando `x` estiver em `[1,9]`; não deve haver condição de corrida; espera ocupada é proibida; `block()` e `wakeup(id)` existem e um `mutex` também está disponível. IDs das threads: `1` e `2`.

```
int x = 0; // variável global

void thread1() {
    while (true) {
        x = x + 1;
    }
}

void thread2() {
    while (true) {
        x = x - 1;
    }
}
```

- Reescreva/atualize o esqueleto acima **sem modificar as linhas originais**: em vez disso, apresente todas as inserções de sincronização (com números de linha imaginários se desejar) para garantir as propriedades pedidas. Mostre uma solução usando `block()` / `wakeup(id)` e uma alternativa usando `condition_variable` (ou semáforo). Explique por que a espera ocupada está evitada.

Questão 4.4 — Remoção de busy-wait Apresente um exemplo curto (código com linhas numeradas) que inicialmente utilize busy-wait para sincronizar duas threads em um ponto; então reescreva indicando as inserções necessárias para transformar a espera ocupada em `block()` / `wakeup()` ou `condition_variable`. Explique por que a segunda versão é preferível em sistemas multitarefa.

Questão 4.5 — Avaliação crítica Escolha **uma** das soluções propostas nas questões anteriores (4.1–4.4) e escreva um pequeno texto apontando possíveis falhas residuais (por exemplo: deadlocks induzidos por ordens incorretas de lock, contenção alta em mutex único, overhead de barreira) e proponha mitigação para cada falha apontada.

FIM DA APOSTILA

Se quiser, posso: - Gerar o gabarito (soluções) para esta apostila, - Transformar isto em um PDF imprimível, - Criar variações (exercícios adicionais) com níveis de dificuldade.

Diga qual dessas opções deseja ou peça alterações na apostila (ex.: mais exercícios, enunciados em inglês, etc.).