

# Apostila de Programação Concorrente e Distribuída

## Sessão 1: Análise de Concorrência e Grafos de Precedência

### Questão 1.1

Dado o código C++ a seguir:

```
cpp

void func(int id) {
    ... cout << "p" << id << " ";
}

int main() {
    thread t1(func, 1);
    ... func(2);
    ... thread t3(func, 3);
    t1.join();
    ... t3.join();
    ... func(4);
    ... cout << endl;
}
```

a) Identifique quais das seguintes saídas são possíveis:

- 
- 
- 
- 
- 

b) O grafo resultante é propriamente aninhado? Justifique.

c) Se for propriamente aninhado, escreva-o usando funções S (sequencial) e P (paralelo).

d) Desenhe o grafo de precedência correspondente.

### Questão 1.2

Analise o seguinte código usando .

```
cpp
```

```

int main() {
    if (fork() == 0) {
        cout << "A";
        if (fork() == 0)
            cout << "B";
        else
            cout << "C";
    } else {
        cout << "D";
    }

    if (fork() == 0)
        cout << "E";
    else
        cout << "F";

    return 0;
}

```

a) Quais das seguintes saídas são válidas?

- ABCDEF
- DABCEF
- DACBEF
- DFABCE
- ADBCEF
- ABDCEF

b) Desenhe a árvore de processos gerada pelo código.

c) Quantos processos são criados no total?

### Questão 1.3

Considere o código multithread abaixo:

cpp

```

void task(int id) {
    ... cout << "T" << id << " ";
}

int main() {
    ... thread t1(task, 1);
    ... thread t2(task, 2);
    ... t1.join();
    ... task(3);
    ... thread t4(task, 4);
    ... t2.join();
    ... t4.join();
    ... task(5);
    ... cout << endl;
}

```

a) Determine se as seguintes execuções são possíveis e justifique:

- T1 T2 T3 T4 T5
- T2 T1 T3 T4 T5
- T1 T3 T2 T4 T5
- T2 T3 T1 T4 T5
- T1 T2 T4 T3 T5

b) Represente o grafo de precedência.

c) Identifique se há paralelismo verdadeiro no código.

## Questão 1.4

Examine este código com múltiplos `fork()`:

cpp

```

int main() {
    cout << "X";

    if (fork() == 0) {
        cout << "Y";
        if (fork() == 0) {
            cout << "Z";
        }
    } else {
        if (fork() == 0) {
            cout << "W";
        }
    }

    cout << "V";
    return 0;
}

```

- a) Identifique todas as saídas possíveis.
- b) Construa o diagrama de execução mostrando todos os processos.
- c) O código apresenta algum tipo de sincronização? Justifique.

## Sessão 2: Algoritmos de Exclusão Mútua

### Questão 2.1

Analise o seguinte algoritmo de exclusão mútua para duas threads:

cpp

```
bool flag[2] = {false, false};
int turn = 0;

// Thread 0
flag[0] = true;
while (flag[1] && turn == 1);
// REGIÃO CRÍTICA
flag[0] = false;

// Thread 1
flag[1] = true;
while (flag[0] && turn == 0);
// REGIÃO CRÍTICA
flag[1] = false;
```

- a) Este algoritmo garante exclusão mútua? Justifique.
- b) Pode ocorrer deadlock? Demonstre com um cenário.
- c) Há possibilidade de starvation? Explique.
- d) O algoritmo é fair (justo)? Analise.
- e) Existe atraso desnecessário? Em que situação?

## Questão 2.2

Considere esta proposta de exclusão mútua:

cpp

```
bool want[2] = {false, false};
```

```
// Thread 0
```

```
want[0] = true;
```

```
while (want[1]) {
```

```
... want[0] = false;
```

```
... // pequeno delay
```

```
... want[0] = true;
```

```
}
```

```
// REGIÃO CRÍTICA
```

```
want[0] = false;
```

```
// Thread 1
```

```
want[1] = true;
```

```
while (want[0]) {
```

```
... want[1] = false;
```

```
... // pequeno delay
```

```
... want[1] = true;
```

```
}
```

```
// REGIÃO CRÍTICA
```

```
want[1] = false;
```

Avalie se o algoritmo satisfaz os seguintes critérios: **a)** Exclusão mútua **b)** Ausência de deadlock **c)** Ausência de starvation  
**d)** Fairness **e)** Ausência de atraso desnecessário

## Questão 2.3

Examine este algoritmo modificado:

```
cpp
```

```

bool ready[2] = {false, false};
int favor = 0;

// Thread 0
ready[0] = true;
favor = 1;
while (ready[1] && favor == 1);
// REGIÃO CRÍTICA
ready[0] = false;

// Thread 1
ready[1] = true;
favor = 0;
while (ready[0] && favor == 0);
// REGIÃO CRÍTICA
ready[1] = false;

```

- a) Trace a execução quando ambas as threads tentam entrar simultaneamente na região crítica.
- b) Verifique todos os critérios de corretude para exclusão mútua.
- c) Compare com o algoritmo de Peterson. Há diferenças?

## Questão 2.4

Analise esta tentativa de implementação:

```

cpp

int ticket = 0;
int serving = 0;

// Thread 0
int my_ticket0 = ticket++;
while (my_ticket0 != serving);
// REGIÃO CRÍTICA
serving++;

// Thread 1
int my_ticket1 = ticket++;
while (my_ticket1 != serving);
// REGIÃO CRÍTICA
serving++;

```

- a) Identifique o problema principal nesta implementação.
- b) Este código pode funcionar corretamente em algum cenário? Qual?

c) Como você modificaria para garantir exclusão mútua?

d) Quais operações deveriam ser atômicas?

---

## Sessão 3: Funções S e P - Grafos de Dependência

### Questão 3.1

Dada a especificação de concorrência:  $S(P(T1, T2), P(T3, S(T4, T5)))$

a) Desenhe o grafo de dependência correspondente.

b) Identifique quais tarefas podem executar em paralelo.

c) Qual é o caminho crítico da execução?

d) Esboce o código C++ usando threads e joins para implementar esta especificação.

### Questão 3.2

Para a expressão:  $P(S(T1, P(T2, T3)), S(T4, T5))$

a) Construa o diagrama de precedência.

b) Liste todas as possíveis ordens de execução válidas.

c) Implemente usando threads em C++, considerando que cada tarefa  $T_i$  imprime apenas o número  $i$ .

d) Quantas threads no mínimo são necessárias para esta execução?

### Questão 3.3

Considere a especificação:  $S(T1, P(S(T2, T3), S(T4, P(T5, T6))))$

a) Desenhe o grafo de precedência.

b) Identifique os pontos de sincronização necessários.

c) Escreva um esboço em alto nível do código usando `block()` e `wakeup()`.

d) Determine o grau máximo de paralelismo desta especificação.

### Questão 3.4

Dada a função:  $P(S(P(T1, T2), T3), S(T4, P(T5, T6)))$

a) Represente graficamente as dependências.

b) Quais tarefas devem necessariamente executar em sequência?



c) Implemente usando apenas joins (sem mutex, block ou wakeup).

d) Analise a eficiência: qual seria o speedup teórico máximo com processadores infinitos?

---

## Sessão 4: Sincronização de Threads e Correção de Código

### Questão 4.1

Analise o seguinte código que será executado por múltiplas threads:

cpp

```
1| void processar(int tid, int n_threads, int* dados, int size, int* resultado) {
2|     int inicio = tid * size / n_threads;
3|     int fim = (tid + 1) * size / n_threads;
4|     int soma_local = 0;
5|
6|     for (int i = inicio; i < fim; i++) {
7|         soma_local += dados[i];
8|     }
9|
10|    resultado[tid] = soma_local;
11|
12|    if (tid == 0) {
13|        int total = 0;
14|        for (int i = 0; i < n_threads; i++) {
15|            total += resultado[i];
16|        }
17|        cout << "Soma total: " << total << endl;
18|    }
19|}
```

a) Identifique os problemas de concorrência no código.

b) Proponha soluções usando mutex, barriers, ou outras primitivas de sincronização.

c) Indique em quais linhas as operações de sincronização devem ser inseridas.

d) Explique como cada solução resolve os problemas identificados.

### Questão 4.2

Considere este código para atualização de contador compartilhado:

cpp

```

int contador = 0;
mutex mtx;

void thread_A() {
    while (true) {
        if (contador < 100) {
            contador = contador + 2;
            cout << "A: " << contador << endl;
        }
    }
}

void thread_B() {
    while (true) {
        if (contador > 0) {
            contador = contador - 1;
            cout << "B: " << contador << endl;
        }
    }
}

```

**a)** Identifique as condições de corrida.

**b)** Modifique o código para garantir que:

- $0 \leq \text{contador} \leq 100$
- Não haja condição de corrida
- Não use espera ocupada

**c)** Use `block()` e `wakeup()` para implementar a sincronização.

### Questão 4.3

Examine este código de produtor-consumidor:

cpp

```

1| int buffer[10];
2| int count = 0;
3| int in = 0, out = 0;
4|
5| void producer(int id) {
6|     for (int i = 0; i < 5; i++) {
7|         int item = id * 10 + i;
8|         buffer[in] = item;
9|         in = (in + 1) % 10;
10|        count++;
11|        cout << "Produzido: " << item << endl;
12|    }
13| }
14|
15| void consumer(int id) {
16|     for (int i = 0; i < 5; i++) {
17|         if (count > 0) {
18|             int item = buffer[out];
19|             out = (out + 1) % 10;
20|             count--;
21|             cout << "Consumido por " << id << ": " << item << endl;
22|         }
23|     }
24| }

```

- a) Liste todos os problemas de sincronização.
- b) Corrija usando mutex e condition variables.
- c) Garanta que produtores esperem quando buffer estiver cheio.
- d) Garanta que consumidores esperem quando buffer estiver vazio.

## Questão 4.4

Analise este código de leitores-escretores:

```
cpp
```

```

1| int dados = 0;
2| int leitores_ativos = 0;
3| bool escritor_ativo = false;
4|
5| void leitor(int id) {
6|     leitores_ativos++;
7|     if (leitores_ativos == 1) {
8|         // primeiro leitor
9|     }
10|
11|     cout << "Leitor " << id << " leu: " << dados << endl;
12|
13|     leitores_ativos--;
14|     if (leitores_ativos == 0) {
15|         // último leitor
16|     }
17| }
18|
19| void escritor(int id, int valor) {
20|     escritor_ativo = true;
21|     dados = valor;
22|     cout << "Escritor " << id << " escreveu: " << valor << endl;
23|     escritor_ativo = false;
24| }

```

- a) Identifique as violações de sincronização.
- b) Implemente a solução clássica de leitores-escritores.
- c) Use mutex para proteger variáveis compartilhadas.
- d) Garanta que escritores tenham acesso exclusivo e múltiplos leitores possam acessar simultaneamente.
- e) Como evitar starvation de escritores?