UNIVERSIDADE FEDERAL DE VIÇOSA

Departamento de Informática

INF310 – PROGRAMAÇÃO CONCORRENTE E DISTRIBUÍDA

APOSTILA DE EXERCÍCIOS

Tópicos: Funções S e P, Análise de Protocolos, Sincronização e Ordem de Execução

---

SEÇÃO 1: FUNÇÕES S e P - CONSTRUÇÃO DE GRAFOS

Exercício 1.1

Dado a especificação de concorrência através das funções S e P:

S(P(S(T1, T2), S(T3, P(T4, T5))), T6)

a) Desenhe o grafo de precedência correspondente.
b) Identifique se o grafo é propriamente aninhado (PN). Justifique sua resposta.
c) Quantas threads no mínimo são necessárias para executar essa especificação?

Exercício 1.2

Considere a especificação:

P(S(T1, P(T2, T3), T4), S(T5, T6))

a) Construa o grafo de precedência.
b) O grafo é propriamente aninhado? Demonstre.
c) Qual seria o tempo total de execução se cada tarefa Ti leva 10 unidades de tempo e temos 3 processadores disponíveis?

Exercício 1.3

Dada a especificação complexa:

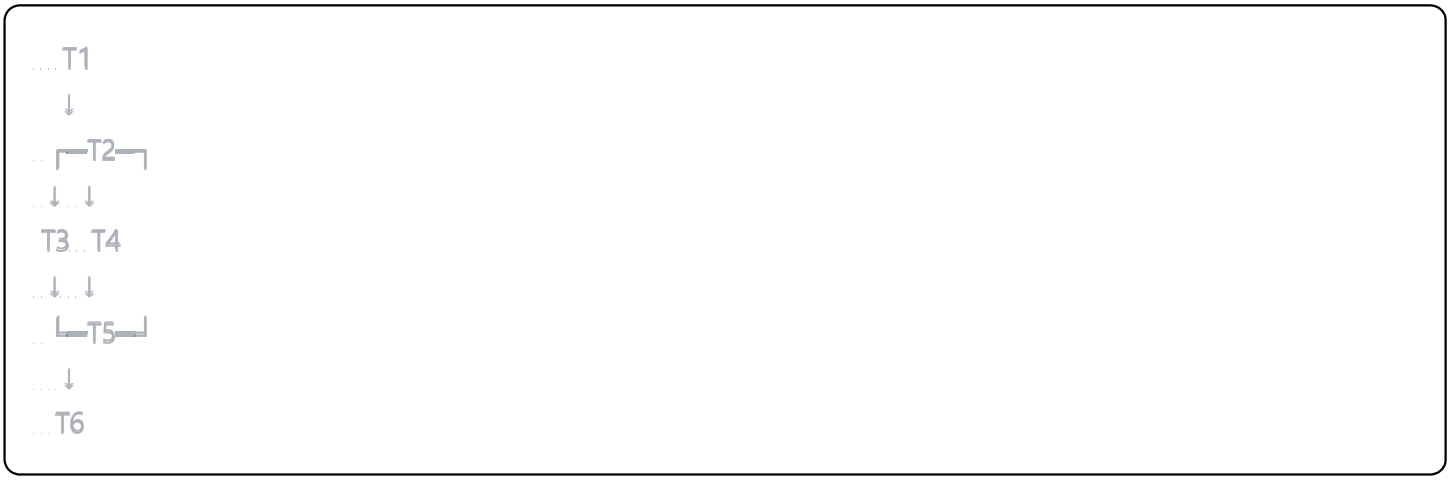
S(P(S(T1, T2), T3), P(S(T4, P(T5, T6)), T7), T8)

a) Desenhe o grafo de precedência.
b) Verifique se é PN e justifique.
c) Identifique todos os pontos de sincronização necessários.

SEÇÃO 2: FUNÇÕES S e P - ANÁLISE DE GRAFOS

Exercício 2.1

Observe o seguinte grafo de precedência:



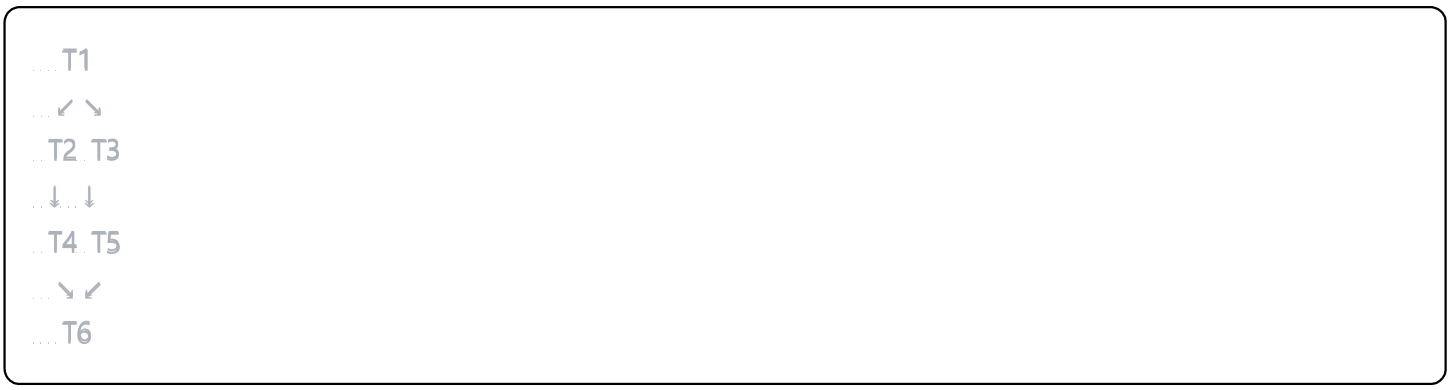
a) Escreva a especificação usando funções S e P.

b) O grafo é propriamente aninhado? Justifique.

c) Desenhe um grafo alternativo que NÃO seja PN para as mesmas 6 tarefas.

Exercício 2.2

Dado o grafo:



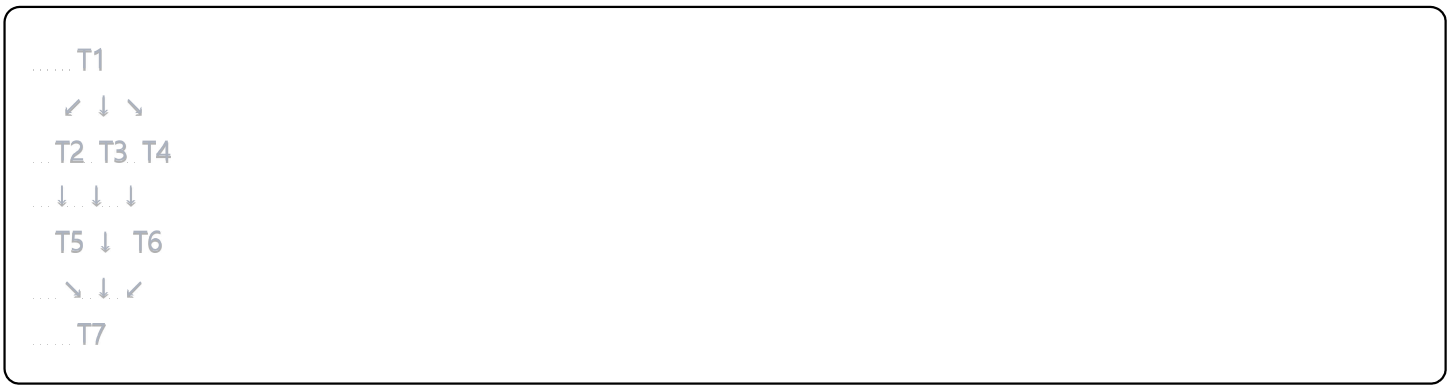
a) Expresse usando funções S e P.

b) Identifique se é PN.

c) Se não for PN, modifique minimamente o grafo para torná-lo PN e reescreva a função.

Exercício 2.3

Analise este grafo mais complexo:



a) Determine a função S e P correspondente.

b) Verifique a propriedade PN.

c) Calcule o grau de paralelismo máximo deste grafo.

SEÇÃO 3: FUNÇÕES S e P - ANÁLISE DE CÓDIGO

Exercício 3.1

Analise o código C++ a seguir:

```cpp
void tarefa(int id) {
    cout << "T" << id << " ";
}

int main() {
    thread t1(tarefa, 1);
    thread t2(tarefa, 2);
    t1.join();
    thread t3(tarefa, 3);
    t2.join();
    tarefa(4);
    t3.join();
    thread t5(tarefa, 5);
    t5.join();
    return 0;
}
```

a) Desenhe o grafo de precedência.

b) Escreva a especificação usando funções S e P.

c) O grafo resultante é PN? Justifique.

Exercício 3.2

Considere este código:

```cpp
void func(int n) { cout << n << " "; }

int main() {
    thread t1(func, 1);
    thread t2(func, 2);
    thread t3(func, 3);

    t1.join();
    func(4);

    thread t5(func, 5);
    t2.join();
    t3.join();

    func(6);
    t5.join();
    return 0;
}
```

a) Construa o grafo de precedência.

b) Determine a função S e P.

c) É propriamente aninhado? Se não, explique onde está o problema.

Exercício 3.3

Examine o código:

```cpp
```

```cpp
void processo(int id) {
    cout << "P" << id << endl;
}

int main() {
    thread t1(processo, 1);
    processo(2);
    t1.join();

    thread t3(processo, 3);
    thread t4(processo, 4);
    thread t5(processo, 5);

    t3.join();
    processo(6);
    t4.join();
    t5.join();

    processo(7);
    return 0;
}
```

a) Faça o grafo de precedência.

b) Escreva usando S e P.

c) Analise se é PN e quantas threads são necessárias no mínimo.

SEÇÃO 4: ANÁLISE DOS 4 PRINCÍPIOS DE EXCLUSÃO MÚTUA

Exercício 4.1

Analise o seguinte protocolo para 2 threads:

cpp

```cpp
// Variáveis globais
bool flag[2] = {false, false};
int turn = 0;

// Thread i (onde j = 1-i)
flag[i] = true;
turn = j;
while(flag[j] && turn == j);
// REGIÃO CRÍTICA
flag[i] = false;
```

Verifique se este protocolo garante: a) Exclusão mútua b) Ausência de deadlock
c) Ausência de atraso desnecessário d) Entrada eventual (ausência de starvation)

Para cada item, justifique com cenários específicos.

Exercício 4.2

Considere este protocolo alternativo:

```cpp
// Variáveis globais
bool wanting[2] = {false, false};
int priority = 0;

// Thread i
wanting[i] = true;
while(wanting[1-i]) {
    if(priority == 1-i) {
        wanting[i] = false;
        while(priority == 1-i);
        wanting[i] = true;
    }
}
// REGIÃO CRÍTICA
priority = 1-i;
wanting[i] = false;
```

Analise todos os 4 princípios fundamentais. Se algum for violado, forneça um cenário específico demonstrando a violação.

Exercício 4.3

Examine este protocolo para N threads:

```cpp
```

```cpp
// Variáveis globais
int level[N]; // todos iniciados em 0
int waiting[N-1]; // valores irrelevantes

// Thread i
for(int L = 1; L < N; L++) {
    level[i] = L;
    waiting[L-1] = i;

    do {
        bool wait = false;
        for(int k = 0; k < N; k++) {
            if(k != i && level[k] >= L) {
                wait = true;
                break;
            }
        }
        if(!wait || waiting[L-1] != i) break;
    } while(true);
}
// REGIÃO CRÍTICA
level[i] = 0;
```

Faça uma análise completa dos 4 princípios. Este é o algoritmo da padaria modificado - identifique se há problemas na implementação.

SEÇÃO 5: MODIFICAÇÃO DE CÓDIGO COM SINCRONIZAÇÃO

Exercício 5.1

O código a seguir possui problemas de sincronização:

cpp

```
int saldo = 1000; // variável global

void depositar(int valor) {
    for(int i = 0; i < 10; i++) {
        int temp = saldo;
        temp += valor;
        saldo = temp;
    }
}

void sacar(int valor) {
    for(int i = 0; i < 5; i++) {
        if(saldo >= valor) {
            int temp = saldo;
            temp -= valor;
            saldo = temp;
        }
    }
}
```

Modifique o código adicionando **apenas** operações `lock()`, `unlock()`, `block()` e `wakeup(tid)` para:

- Eliminar condições de corrida
- Garantir que o saldo nunca fique negativo
- Evitar starvation entre operações de depósito e saque

Restrição: Não altere as linhas existentes, apenas adicione operações de sincronização.

Exercício 5.2

Considere este código problemático:

```
cpp
```

```
int buffer[10];
int count = 0;

void produtor(int id) {
    for(int i = 0; i < 20; i++) {
        if(count < 10) {
            buffer[count] = i + id * 100;
            count++;
        }
    }
}

void consumidor(int id) {
    for(int i = 0; i < 15; i++) {
        if(count > 0) {
            count--;
            int item = buffer[count];
            cout << "Consumido: " << item << endl;
        }
    }
}
```

Adicione sincronização usando `lock()`, `unlock()`, `block()` e `wakeup(tid)` para:

- Eliminar condições de corrida
- Garantir que produtores bloqueiem quando buffer cheio
- Garantir que consumidores bloqueiem quando buffer vazio
- Implementar sincronização eficiente sem espera ocupada

Exercício 5.3

Este código implementa um contador compartilhado:

```cpp
```

```
int contador = 0;
int limite_superior = 100;
int limite_inferior = 0;

void incrementar() {
    while(true) {
        if(contador < limite_superior) {
            contador++;
            cout << "Inc: " << contador << endl;
        }
    }
}

void decrementar() {
    while(true) {
        if(contador > limite_inferior) {
            contador--;
            cout << "Dec: " << contador << endl;
        }
    }
}

void resetar() {
    while(true) {
        // Espera um tempo aleatório
        contador = 50; // valor médio
        cout << "Reset para 50" << endl;
    }
}
```

Adicione sincronização para:

- Eliminar condições de corrida
- Garantir que apenas uma operação ocorra por vez
- A função resetar deve ter prioridade sobre as outras
- Evitar starvation de qualquer função

SEÇÃO 6: ANÁLISE DE ORDEM DE SAÍDA

Exercício 6.1

Analise este código e determine todas as possíveis saídas:

```
cpp
```

```cpp
int x = 5;

int main() {
    if(fork() == 0) {
        x += 3;
        cout << x;
        if(fork() == 0) {
            x *= 2;
            cout << x;
        } else {
            x--;
            cout << x;
        }
    } else {
        x += 10;
        cout << x;
        if(fork() == 0) {
            x /= 3;
            cout << x;
        } else {
            x += 5;
            cout << x;
        }
    }
    return 0;
}
```

a) Liste todas as saídas possíveis.

b) Existe condição de corrida? Justifique.

c) Quantos processos são criados no total?

Exercício 6.2

Para este código com threads:

```cpp
```

```cpp
int global = 0;
mutex m;

void funcao(int id) {
    m.lock();
    global += id;
    cout << global << " ";
    m.unlock();
}

int main() {
    thread t1(funcao, 1);
    thread t2(funcao, 2);
    thread t3(funcao, 3);

    t2.join();
    cout << "MID ";
    t1.join();
    t3.join();
    cout << "FIM" << endl;

    return 0;
}
```

a) Quais saídas são possíveis?

b) O que garante a sincronização correta?

c) A palavra "MID" sempre aparecerá na mesma posição relativa? Justifique.

Exercício 6.3

Examine este código complexo:

```cpp
```

```
void tarefa(int n) {
    cout << n;
}

int main() {
    if(fork() == 0) {
        cout << "A";
        thread t1(tarefa, 1);
        cout << "B";
        t1.join();
        cout << "C";
    } else {
        thread t2(tarefa, 2);
        cout << "D";
        if(fork() == 0) {
            cout << "E";
        } else {
            t2.join();
            cout << "F";
        }
    }
    cout << "G";
    return 0;
}
```

a) Desenhe a árvore de processos/threads.

b) Liste pelo menos 5 saídas possíveis diferentes.

c) Identifique quais caracteres podem aparecer juntos sem separação.

SEÇÃO 7: IMPLEMENTAÇÃO EM ALTO NÍVEL

Exercício 7.1

Problema dos Leitores-Escritores com Prioridade

Implemente uma solução para o problema dos leitores-escritores onde:

- Múltiplos leitores podem acessar simultaneamente
- Apenas um escritor pode acessar por vez
- Escritores têm prioridade sobre leitores
- Não deve haver starvation de leitores

Use apenas `block()`, `wakeup(tid)`, `lock()`, `unlock()` e variáveis globais. Implemente as funções:

- `iniciar_leitura()`
- `finalizar_leitura()`
- `iniciar_escrita()`
- `finalizar_escrita()`

Exercício 7.2

Problema do Barbeiro Dorminhoco Estendido

Um salão tem 2 barbeiros e 5 cadeiras de espera. Implemente a sincronização para:

- Clientes chegam e sentam se há cadeira livre, senão vão embora
- Se não há clientes, barbeiros dormem
- Apenas um cliente pode ser atendido por barbeiro
- Cliente deve acordar barbeiro se necessário

Implemente usando `block()`, `wakeup(tid)`, `lock()`, `unlock()`:

- `cliente_chega(int id)`
- `barbeiro_trabalha(int barbeiro_id)`
- `cliente_vai_embora(int id)`

Exercício 7.3

Problema da Ponte Estreita com Sentido

Uma ponte suporta no máximo 3 veículos simultaneamente, mas veículos só podem atravessar no mesmo sentido. Implemente:

- Veículos do Norte e do Sul querem atravessar
- No máximo 3 veículos na ponte simultaneamente
- Todos na ponte devem ir no mesmo sentido
- Evitar starvation de qualquer direção
- Após 5 veículos consecutivos de um lado, dar prioridade ao outro

Use `block()`, `wakeup(tid)`, `lock()`, `unlock()` para implementar:

- `veiculo_norte_entra(int id)`
- `veiculo_norte_sai(int id)`
- `veiculo_sul_entra(int id)`
- `veiculo_sul_sai(int id)`

Exercício 7.4

Sistema de Reserva de Recursos com Deadlock Prevention

Implemente um sistema onde:

- 4 tipos de recursos (A, B, C, D) com quantidades limitadas
- Processos podem solicitar múltiplos recursos
- Implementar prevenção de deadlock por ordenação
- Processo deve adquirir todos os recursos ou nenhum (atomicidade)

Quantidades disponíveis: A=3, B=2, C=4, D=1

Implemente usando `block()`, `wakeup(tid)`, `lock()`, `unlock()`:

- `solicitar_recursos(int processo_id, int qtd_A, int qtd_B, int qtd_C, int qtd_D)`
- `liberar_recursos(int processo_id)`
- `verificar_disponibilidade(int qtd_A, int qtd_B, int qtd_C, int qtd_D)`

INSTRUÇÕES GERAIS:

1. Para cada exercício, justifique suas respostas detalhadamente
2. Quando solicitado código, implemente em alto nível mas seja preciso na lógica
3. Considere sempre os 4 princípios da exclusão mútua ao analisar protocolos
4. Em grafos, deixe claro os nós de início e fim
5. Para saídas de código, considere todas as possibilidades de intercalação
6. Use terminologia técnica correta em suas explicações

Boa prática e estudos!