Apostila de Exercícios - Concorrência e Distribuição

Sessão 1: Conceitos Teóricos Fundamentais

Questão 1

Explique detalhadamente a diferença entre concorrência e paralelismo. Forneça um exemplo prático para cada conceito e discuta por que um programa concorrente nem sempre é paralelo.

Questão 2

A seguinte afirmação está **incorreta**: "Em um sistema monoprocessador, nunca pode haver verdadeiro paralelismo, portanto threads são inúteis nesse tipo de sistema."

Explique por que esta afirmação é falsa e cite pelo menos três cenários onde threads são vantajosas mesmo em sistemas monoprocessador.

Questão 3

Descreva o papel dos registradores PC (Program Counter), PSW (Program Status Word) e SP (Stack Pointer) no contexto de multithreading. Como o sistema operacional utiliza esses registradores durante uma mudança de contexto?

Questão 4

Compare e contraste multiprocessamento simétrico (SMP) com multiprocessamento assimétrico. Quais são as vantagens e desvantagens de cada abordagem? Em que cenários você escolheria uma sobre a outra?

Questão 5

Analise a seguinte situação: "Um sistema possui 4 cores físicos com hyperthreading, resultando em 8 threads lógicos. Um programa cria 12 threads de trabalho."

Discuta como o sistema operacional gerenciará essas threads, quais são as implicações de performance e como o escalonador pode afetar a execução.

Questão 6

Explique o conceito de interrupções no contexto de sistemas concorrentes. Como as interrupções diferem de system calls? Forneça exemplos práticos de quando cada uma é utilizada.

Questão 7

A afirmação "System calls sempre bloqueiam o processo chamador até que a operação seja completada" está correta? Justifique sua resposta e explique os diferentes tipos de system calls em termos de

comportamento de bloqueio.

Questão 8

Descreva o funcionamento interno de uma mudança de contexto entre threads no mesmo processo versus mudança de contexto entre processos diferentes. Quais estruturas de dados são preservadas/alteradas em cada caso?

Questão 9

Explique como o sistema operacional gerencia a pilha (stack) em ambientes multithread. Por que cada thread precisa de sua própria pilha? O que aconteceria se duas threads compartilhassem a mesma região de pilha?

Questão 10

Discuta os trade-offs entre criar muitas threads versus poucas threads em uma aplicação. Como fatores como overhead de criação, mudança de contexto, contenção de recursos e arquitetura do hardware influenciam essa decisão?

Questão 11

A seguinte afirmação é verdadeira ou falsa: "Em sistemas distribuídos, a concorrência sempre implica em paralelismo real"? Justifique sua resposta considerando diferentes arquiteturas de sistemas distribuídos.

Questão 12

Explique como o conceito de "falso compartilhamento" (false sharing) pode afetar a performance em sistemas multicore. Como isso se relaciona com a hierarquia de cache e como pode ser mitigado?

Sessão 2: Grafos de Fluxo e Linguagem Join/Fork

Questão 13

Dado o seguinte código em linguagem join/fork:

```
A
fork B, C
B; join
C; fork D, E
D; join
E; join
F
```

Desenhe o grafo de fluxo correspondente e identifique todos os possíveis caminhos de execução paralela.

Questão 14

Analise o seguinte grafo de fluxo:

```
A → fork(B,C)
B → D
C → fork(E,F)
D → join → G
E → join → G
F → join → G
G → H
```

Escreva o código equivalente em linguagem join/fork e determine quantas threads estarão executando simultaneamente no pico de paralelismo.

Questão 15

A seguinte sequência de saída seria possível para o grafo abaixo? Justifique sua resposta.

```
Grafo: A → fork(B,C) → B,C → join → D
Saída proposta: A, C, B, D
```

Questão 16

Crie um grafo de fluxo que represente o seguinte cenário: "Processar um arquivo em paralelo dividindo-o em 3 partes, onde cada parte é processada independentemente, mas o resultado final só pode ser gerado após todas as partes serem processadas."

Questão 17

Dado o código:

```
START
fork TASK1, TASK2, TASK3
TASK1; fork SUBTASK1A, SUBTASK1B
SUBTASK1A; join
SUBTASK1B; join
TASK2; join
TASK3; join
END
```

Identifique o ponto de maior paralelismo e explique por que certas threads devem aguardar outras antes de prosseguir.

Questão 18

Analise se a seguinte execução é válida para o grafo dado:

```
Grafo: A → fork(B,C,D) → B,C,D → join → E → fork(F,G) → F,G → join → H
Execução: A, B, C, D, E, F, G, H
```

Se for inválida, explique por que e forneça uma execução válida alternativa.

Questão 19

Desenhe um grafo que permita que as tarefas A, B, C e D executem em paralelo, mas E só pode começar após A e B terminarem, F só pode começar após C e D terminarem, e G só pode começar após E e F terminarem.

Questão 20

Considere o seguinte código com aninhamento:

```
MAIN
fork BRANCH1, BRANCH2
BRANCH1; fork SUB1, SUB2
SUB1; join
SUB2; fork LEAF1, LEAF2
LEAF1; join
LEAF2; join
BRANCH2; join
FINAL
```

Calcule o número máximo de threads que podem executar simultaneamente e identifique em que ponto isso ocorre.

Questão 21

A saída "MAIN, BRANCH1, SUB1, SUB2, LEAF1, LEAF2, BRANCH2, FINAL" seria possível para o código da questão anterior? Explique detalhadamente por que sim ou não.

Questão 22

Projete um grafo de fluxo para implementar um padrão "pipeline" com 4 estágios, onde cada estágio pode processar uma unidade de dados enquanto o próximo estágio processa a unidade anterior.

Sessão 3: Implementação e Análise de Código

Questão 23

Implemente em C++ usando pthreads um programa que:

- Crie 4 threads trabalhadoras
- Cada thread calcule a soma de um quarto de um vetor de 1000 elementos
- A thread principal colete os resultados parciais e compute o resultado final
- Use pthread_join adequadamente

Questão 24

Analise o seguinte código C++ e explique o que ele faz:

```cpp
#include <thread>
#include <vector>
#include <iostream>

void worker(int id, std::vector<int>& data, int start, int end) {
    for(int i = start; i < end; i++) {
        data[i] *= 2;
    }
    std::cout << "Thread " << id << " completed\n";
}

int main() {
    std::vector<int> data(100, 1);
    std::vector<std::thread> threads;

    for(int i = 0; i < 4; i++) {
        threads.emplace_back(worker, i, std::ref(data), i*25, (i+1)*25);
    }

    for(auto& t : threads) {
        t.join();
    }

    return 0;
}
```

Identifique possíveis problemas e como corrigi-los.

Questão 25

Usando std::promise e std::future, implemente uma função que:

- Lance uma thread para calcular o fatorial de um número
- A thread principal continue executando outras tarefas

- Eventualmente colete o resultado do fatorial
- Trate o caso onde o cálculo pode falhar

Questão 26

A seguinte saída seria possível para o código abaixo? Justifique.

```cpp
std::mutex mtx;
int counter = 0;

void increment(int times) {
    for(int i = 0; i < times; i++) {
        std::lock_guard<std::mutex> lock(mtx);
        counter++;
        std::cout << counter << " ";
    }
}

int main() {
    std::thread t1(increment, 3);
    std::thread t2(increment, 2);
    t1.join();
    t2.join();
}
```

Saída proposta: "1 3 2 4 5"

Questão 27

Implemente uma função usando std::move que:

- Receba um std::vectorstd::string por valor
- Crie uma thread que processe esse vector (conte quantas strings têm mais de 5 caracteres)
- Mova o vector para a thread evitando cópia desnecessária
- Retorne o resultado usando std::future

Questão 28

Analise este código e explique todos os possíveis problemas de concorrência:

```cpp
```

```cpp
class Counter {
    int value = 0;
public:
    void increment() { value++; }
    int get() { return value; }
};

Counter global_counter;

void worker() {
    for(int i = 0; i < 1000; i++) {
        global_counter.increment();
    }
}

int main() {
    std::vector<std::thread> threads;
    for(int i = 0; i < 10; i++) {
        threads.emplace_back(worker);
    }

    for(auto& t : threads) {
        t.join();
    }

    std::cout << global_counter.get() << std::endl;
    return 0;
}
```

Questão 29

Usando pthread_create e pthread_join, implemente um programa que:

- Crie uma thread que leia números de um arquivo
- Crie outra thread que processe esses números (calcule média)
- Use uma estrutura compartilhada para passar dados entre as threads
- Implemente sincronização adequada

Questão 30

O seguinte código pode produzir a saída "Thread 2 finalizada, Thread 1 finalizada, Thread 0 finalizada"? Explique por que.

```cpp
```

```cpp
void task(int id) {
    std::this_thread::sleep_for(std::chrono::milliseconds(id * 100));
    std::cout << "Thread " << id << " finalizada, ";
}

int main() {
    for(int i = 0; i < 3; i++) {
        std::thread(task, i).detach();
    }
    std::this_thread::sleep_for(std::chrono::seconds(1));
    return 0;
}
```

Questão 31

Implemente uma classe ThreadSafeQueue usando std::mutex e std::condition_variable que suporte:

- push(): adicionar elemento à fila
- pop(): remover e retornar elemento (deve bloquear se fila vazia)
- empty(): verificar se fila está vazia (thread-safe)
- size(): retornar tamanho atual (thread-safe)

Questão 32

Analise este código com std::async e determine todas as possíveis sequências de execução:

cpp

```cpp
auto future1 = std::async(std::launch::async, []() {
    std::cout << "Task 1 start ";
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
    std::cout << "Task 1 end ";
    return 1;
});

auto future2 = std::async(std::launch::async, []() {
    std::cout << "Task 2 start ";
    std::this_thread::sleep_for(std::chrono::milliseconds(50));
    std::cout << "Task 2 end ";
    return 2;
});

int result = future1.get() + future2.get();
std::cout << "Result: " << result;
```

Questão 33

Implemente um padrão Producer-Consumer usando:

- Uma thread produtora que gera números de 1 a 100
- Três threads consumidoras que processam esses números
- Um buffer limitado (capacidade 10)
- Sincronização adequada para evitar race conditions e deadlocks

Questão 34

Dado o seguinte código, explique por que ele pode não funcionar conforme esperado e como corrigi-lo:

```cpp
bool ready = false;
int data = 0;

void producer() {
    data = 42;
    ready = true;
}

void consumer() {
    while (!ready) {
        std::this_thread::yield();
    }
    std::cout << data << std::endl;
}
```

Notas Finais

Esta apostila abrange os principais conceitos de concorrência e distribuição estudados, desde fundamentos teóricos até implementações práticas. As questões foram elaboradas para testar diferentes níveis de compreensão:

- **Conceitual**: Entendimento dos princípios fundamentais
- **Analítico**: Capacidade de analisar grafos e fluxos de execução
- **Prático**: Habilidade de implementar soluções usando as ferramentas estudadas

Para obter o máximo benefício, tente responder as questões sem consultar material de apoio inicialmente, depois verifique e aprofunde seus conhecimentos conforme necessário.

Lembre-se de que em concorrência e distribuição, muitas vezes não existe uma única resposta "correta" - o importante é demonstrar compreensão dos trade-offs e justificar suas decisões técnicas.