

Tarefa 9: Regiões críticas nomeadas e Locks explícitos

Descrição

Este programa demonstra o uso de regiões críticas nomeadas e locks explícitos em OpenMP para realizar inserções concorrentes em múltiplas listas encadeadas, garantindo a integridade dos dados e evitando condições de corrida.

Locks Explícitos (Regiões Críticas Nomeadas)

O que são Locks?

Lock (ou mutex - mutual exclusion) é um mecanismo de sincronização que garante acesso **mutuamente exclusivo** a um recurso compartilhado. Funciona como uma "chave digital" que apenas uma thread pode possuir por vez.

Conceito Fundamental

Um lock possui dois estados:

- 🔓 **Desbloqueado (unlocked)**: Recurso disponível, qualquer thread pode adquirir
- 🔒 **Bloqueado (locked)**: Recurso ocupado, outras threads ficam em espera

Operações Básicas de Lock

```
// 1. Declaração do lock
omp_lock_t meu_lock;

// 2. Inicialização
omp_init_lock(&meu_lock);
```

```
// 3. Adquirir o lock (bloqueia se necessário)
omp_set_lock(&meu_lock);
// SEÇÃO CRÍTICA - apenas uma thread executa
omp_unset_lock(&meu_lock);

// 4. Destruição (limpeza)
omp_destroy_lock(&meu_lock);
```

Exemplo Simples de Uso: Contador Compartilhado

```
int contador = 0;
omp_lock_t lock_contador;

// Thread segura incrementando contador
omp_set_lock(&lock_contador);
contador++; // Seção crítica protegida
printf("Contador: %d\n", contador);
omp_unset_lock(&lock_contador);
```

Implementação no Nosso Projeto

- Cada lista possui seu próprio lock (`omp_lock_t`)
- Inserções em listas diferentes podem ocorrer simultaneamente
- Inserções na mesma lista são serializadas automaticamente
- Uso de `omp_set_lock()` e `omp_unset_lock()` para controle explícito

Vantagem dos Locks vs Regiões Críticas


Flexibilidade: Locks podem ser criados dinamicamente em runtime, enquanto regiões críticas nomeadas devem ser definidas estaticamente no código.

Por que Locks Explícitos são Necessários?

Limitações das Regiões Críticas Tradicionais

1. Problema da Serialização Global

```
// Abordagem INADEQUADA com região crítica global
#pragma omp critical
{
    insert_element(random_list, value); // TODAS as inserções
}
```



Problema: Mesmo inserindo em listas diferentes, threads ficam bloqueadas.

2. Regiões Críticas Nomeadas - Limitação Estática

```
// Abordagem LIMITADA com nomes fixos
#pragma omp critical(list1)
{
    insert_element(&list1, value);
}

#pragma omp critical(list2)
{
    insert_element(&list2, value);
}
```

Problema: Funciona apenas para número fixo e predefinido de listas.

Solução com Locks Explícitos

3. Locks Dinâmicos - Solução Escalável

```
//SOLUÇÃO: lock por lista, dinamicamente alocado
omp_set_lock(&lists[chosen_list].lock);
insert_element(&lists[chosen_list], value);
omp_unset_lock(&lists[chosen_list].lock);
```

Vantagens dos Locks Explícitos

1. **Granularidade Dinâmica:** Cada lista tem seu lock individual
2. **Escalabilidade:** Funciona com qualquer número N de listas
3. **Paralelismo Máximo:** Threads em listas diferentes não se bloqueiam
4. **Flexibilidade:** Número de recursos protegidos determinado em runtime

5. Performance: Contenção mínima entre threads

Análise Comparativa

Abordagem	Paralelismo	Escalabilidade	Flexibilidade	Performance
#pragma omp critical	✗ Serialização total	✗ Não escala	✗ Inflexível	✗ Baixa
#pragma omp critical(nome)	✓ Parcial	✗ Limitado	✗ Estático	● Média
Locks Explícitos	✓ Máximo	✓ Ilimitado	✓ Dinâmico	✓ Alta

Cenário Prático

Com 4 threads e 3 listas:

- Região crítica global: Máximo 1 thread ativa (25% uso)
- Locks explícitos: Até 3 threads ativas simultaneamente (75% uso)

Características do Programa

1. Estrutura de Dados

- Lista Encadeada: Cada lista possui sua própria estrutura com:
 - Ponteiro para o primeiro nó (head)
 - Lock exclusivo (omp_lock_t)
 - Contador de elementos (count)
 - Identificador único (id)

2. Funcionalidades

Programa Interativo Principal

- Aceita número de listas definido pelo usuário
- Distribui inserções aleatoriamente entre todas as listas
- Escalável para qualquer número de listas
- Interface interativa para configuração personalizada
- Execução direta sem demonstrações preliminares

Exemplo de Saída

TAREFA 9: Regiões críticas nomeadas e Locks explícitos

=====

=== TESTE INTERATIVO ===

Digite o número de listas: 2

Digite o número de inserções: 20

Digite o número de threads: 3

=== PROGRAMA COM 2 LISTAS ===

Número de inserções: 20

Número de threads: 3

Thread 0 inserindo 968 na Lista 2

Thread 1 inserindo 421 na Lista 2 ← Simultâneo!

Thread 0 inserindo 727 na Lista 2

Thread 2 inserindo 649 na Lista 2

Thread 0 inserindo 717 na Lista 1 ← Mudança de lista!

...

Resultados após 20 inserções em 2 listas:

Lista 1 (11 elementos): 432 471 349 384 ...

Lista 2 (9 elementos): 283 102 819 541 ...

Tempo total: 0.0152 segundos

Total de elementos: 20

Programa concluído com sucesso!

Conceitos Demonstrados

1. Locks Explícitos vs Regiões Críticas

- **Diferença fundamental:** Locks permitem granularidade dinâmica
- **Escalabilidade:** Funciona com N listas determinado em runtime
- **Performance:** Paralelismo real entre recursos diferentes

2. Thread Safety Dinâmica

- **Proteção específica:** Cada lista protegida individualmente
- **Contenção mínima:** Threads só competem pela mesma lista
- **Sincronização eficiente:** Locks apenas quando necessário

3. Paralelismo de Granularidade Fina

- **Múltiplos recursos:** Várias listas acessadas simultaneamente
- **Balanceamento:** Distribuição aleatória equilibra carga
- **Escalabilidade:** Performance melhora com mais listas

Vantagens da Abordagem

1. **Granularidade Dinâmica:** Locks específicos por lista, número variável
2. **Paralelismo Máximo:** Inserções simultâneas em listas diferentes
3. **Flexibilidade Total:** Funciona com qualquer configuração N listas
4. **Segurança Garantida:** Integridade dos dados sem race conditions
5. **Performance Ótima:** Throughput superior a alternativas estáticas
6. **Demonstração Clara:** Visualização do comportamento paralelo

Código Fonte



```

// Estrutura do nó da lista encadeada
typedef struct Node {
    int data; // Dados armazenados no nó
    struct Node* next; // Ponteiro para o próximo nó
} Node;

// Estrutura para representar uma lista com seu lock
typedef struct {
    Node* head; // Ponteiro para o primeiro nó da lista
    omp_lock_t lock; // Lock exclusivo para esta lista específica
    int count; // Contador de elementos na lista
    int id; // Identificador único da lista
} LinkedList;

// Função para criar um novo nó
Node* create_node(int data) {
    Node* new_node = (Node*)malloc(sizeof(Node)); // Aloca memória para o novo nó
    if (new_node == NULL) { // Verifica se a alocação foi bem-sucedida
        fprintf(stderr, "Erro ao alocar memória para novo nó\n");
        exit(1); // Termina o programa em caso de erro
    }
    new_node->data = data; // Define o valor do nó
    new_node->next = NULL; // Inicializa o ponteiro next como NULL
    return new_node; // Retorna o nó criado
}

// Função para inicializar uma lista
void init_list(LinkedList* list, int id) {
    list->head = NULL; // Lista começa vazia
    list->count = 0; // Contador inicia em zero
    list->id = id; // Define o identificador da lista
    omp_init_lock(&list->lock); // Inicializa o lock exclusivo da lista
}

// Função para destruir uma lista e liberar memória
void destroy_list(LinkedList* list) {
    Node* current = list->head; // Começa pelo primeiro nó
    while (current != NULL) { // Percorre todos os nós
        Node* temp = current; // Guarda referência do nó atual
        current = current->next; // Avança para o próximo nó
        free(temp); // Libera memória do nó atual
    }
    omp_destroy_lock(&list->lock); // Destroi o lock da lista
}

// Função para inserir um elemento na lista (thread-safe)
void insert_element(LinkedList* list, int data) {
    Node* new_node = create_node(data); // Cria o novo nó a ser inserido

    // Região crítica nomeada para esta lista específica
    omp_set_lock(&list->lock); // Adquire lock exclusivo da lista

    printf("Thread %d inserindo %d na Lista %d\n",
        omp_get_thread_num(), data, list->id); // Mostra qual thread está inserindo

    // Inserção no início da lista para simplicidade
    new_node->next = list->head; // Novo nó aponta para o antigo primeiro
    list->head = new_node; // Novo nó torna-se o primeiro
    list->count++; // Incrementa contador de elementos

    // Simula algum processamento durante a inserção
    usleep(1000); // 1ms de delay para visualizar paralelismo

    omp_unset_lock(&list->lock); // Libera o lock da lista
}

// Função para imprimir os elementos de uma lista
void print_list(LinkedList* list) {
    printf("Lista %d (%d elementos): ", list->id, list->count); // Cabeçalho da lista
    Node* current = list->head; // Começa pelo primeiro nó
    while (current != NULL) { // Percorre todos os nós
        printf("%d ", current->data); // Imprime o valor do nó atual
        current = current->next; // Avança para o próximo nó
    }
    printf("\n"); // Nova linha ao final
}

// Programa com duas listas
void program_two_lists(int num_insertions, int num_threads) {
    printf("\n=== PROGRAMA COM DUAS LISTAS ===\n");
    printf("Número de inserções: %d\n", num_insertions);
    printf("Número de threads: %d\n\n", num_threads);

    LinkedList list1, list2; // Declara duas listas
    init_list(&list1, 1); // Inicializa lista 1
    init_list(&list2, 2); // Inicializa lista 2

    double start_time = omp_get_wtime(); // Marca tempo de início

    #pragma omp parallel num_threads(num_threads) // Inicia região paralela
    {

```



```

    unsigned int seed = time(NULL) + omp_get_thread_num(); // Seed única por thread

#pragma omp for // Distribui iterações entre threads
for (int i = 0; i < num_insertions; i++) {
    // Escolhe aleatoriamente entre lista 1 ou 2
    int choice = rand_r(&seed) % 2; // 0 ou 1 para escolher lista
    int value = rand_r(&seed) % 1000; // Valor aleatório entre 0-999

    if (choice == 0) {
        insert_element(&list1, value); // Insere na lista 1
    } else {
        insert_element(&list2, value); // Insere na lista 2
    }
}

double end_time = omp_get_wtime(); // Marca tempo de fim

printf("\nResultados após %d inserções:\n", num_insertions);
print_list(&list1); // Mostra conteúdo da lista 1
print_list(&list2); // Mostra conteúdo da lista 2
printf("Tempo total: %.4f segundos\n", end_time - start_time); // Calcula tempo decorrido
printf("Total de elementos: %d\n", list1.count + list2.count); // Soma total de elementos

destroy_list(&list1); // Libera memória da lista 1
destroy_list(&list2); // Libera memória da lista 2
}

// Programa generalizado para N listas
void program_n_lists(int num_lists, int num_insertions, int num_threads) {
    printf("\n=== PROGRAMA COM %d LISTAS ===\n", num_lists);
    printf("Número de inserções: %d\n", num_insertions);
    printf("Número de threads: %d\n", num_threads);

    // Aloca memória para array de listas dinamicamente
    LinkedList* lists = (LinkedList*)malloc(num_lists * sizeof(LinkedList));
    if (lists == NULL) { // Verifica se alocação foi bem-sucedida
        fprintf(stderr, "Erro ao alocar memória para as listas\n");
        exit(1); // Termina programa em caso de erro
    }

    // Inicializa todas as listas
    for (int i = 0; i < num_lists; i++) {
        init_list(&lists[i], i + 1); // Cada lista tem ID sequencial (1, 2, 3...)
    }

    double start_time = omp_get_wtime(); // Marca tempo de início

#pragma omp parallel num_threads(num_threads) // Inicia região paralela
    {
        unsigned int seed = time(NULL) + omp_get_thread_num(); // Seed única por thread

#pragma omp for // Distribui iterações entre threads
        for (int i = 0; i < num_insertions; i++) {
            // Escolhe aleatoriamente uma das N listas
            int list_choice = rand_r(&seed) % num_lists; // Índice da lista escolhida
            int value = rand_r(&seed) % 1000; // Valor aleatório entre 0-999

            insert_element(&lists[list_choice], value); // Insere na lista escolhida
        }
    }

    double end_time = omp_get_wtime(); // Marca tempo de fim

    printf("\nResultados após %d inserções em %d listas:\n", num_insertions, num_lists);
    int total_elements = 0; // Contador total de elementos
    for (int i = 0; i < num_lists; i++) {
        print_list(&lists[i]); // Mostra conteúdo de cada lista
        total_elements += lists[i].count; // Soma elementos de todas as listas
    }

    printf("Tempo total: %.4f segundos\n", end_time - start_time); // Calcula tempo decorrido
    printf("Total de elementos: %d\n", total_elements); // Mostra total de elementos

    // Libera memória das listas
    for (int i = 0; i < num_lists; i++) {
        destroy_list(&lists[i]); // Destroi cada lista individualmente
    }
    free(lists); // Libera array de listas
}

int main() {
    printf("TAREFA 9: Regiões críticas nomeadas e Locks explícitos\n");

    srand(time(NULL)); // Inicializa gerador de números aleatórios

    // Teste interativo para que o usuário defina o número de listas
    printf("\n=== TESTE INTERATIVO ===\n");
    int num_lists, num_insertions, num_threads; // Variáveis para entrada do usuário

    printf("Digite o número de listas: ");
    if (scanf("%d", &num_lists) != 1 || num_lists < 1) { // Lê e valida número de listas
        fprintf(stderr, "Número de listas inválido\n");
        return 1; // Retorna erro se entrada inválida
    }
}

```

```
printf("Digite o número de inserções: ");
if (scanf("%d", &num_insertions) != 1 || num_insertions < 1) { // Lê e valida número de inserções
    fprintf(stderr, "Número de inserções inválido\n");
    return 1; // Retorna erro se entrada inválida
}

printf("Digite o número de threads: ");
if (scanf("%d", &num_threads) != 1 || num_threads < 1) { // Lê e valida número de threads
    fprintf(stderr, "Número de threads inválido\n");
    return 1; // Retorna erro se entrada inválida
}

program_n_lists(num_lists, num_insertions, num_threads); // Executa programa principal

printf("\nPrograma concluído com sucesso!\n");
return 0; // Retorna sucesso
}
```

Figura 1: Código fonte completo da Tarefa 9 - Demonstração de regiões críticas nomeadas e locks explícitos em OpenMP