Exercício Prático: Análise de Performance de Contadores Concorrentes

Sistemas Operacionais

12 de agosto de 2025

Objetivos de Aprendizagem

- Aplicar a API de threads POSIX (`pthreads`) para criar e gerenciar múltiplas threads.
- Implementar e comparar duas estratégias de concorrência para um contador: uma com lock de granularidade grossa (simples/preciso) e uma escalável (aproximado).
- Utilizar ferramentas de medição de tempo para conduzir uma análise de performance empírica.
- Analisar e explicar os trade-offs entre contenção de lock, escalabilidade e precisão.

Contexto Teórico

Este exercício é uma aplicação direta dos conceitos discutidos na Seção 29.1 do Capítulo 29 ("Lock-based Concurrent Data Structures"). O objetivo é recriar os experimentos que geraram os gráficos das Figuras 29.5 e 29.6, permitindo que você observe empiricamente como diferentes designs de sincronização afetam a performance de uma estrutura de dados simples sob contenção.

Fase 1: Implementação do Contador Simples (Preciso)

Baseado na Figura 29.2 (Capítulo 29, página 3), você deverá implementar as funções para a estrutura `counter_precise_t`. Esta implementação deve usar um **único lock global** para proteger o acesso ao contador, garantindo que cada incremento seja atômico.

Fase 2: Implementação do Contador Escalável (Aproximado)

Baseado na Figura 29.4 (Capítulo 29, página 5), você deverá implementar as funções para a estrutura `counter_approx_t`. Esta implementação utiliza locks e contadores locais (um por núcleo de CPU) para minimizar a contenção, e um lock global para atualizações

periódicas. A lógica de atualização do contador global deve ser acionada por um **limiar (threshold `S`)**.

Esqueleto de Código (Ponto de Partida)

Utilize o arquivo de cabeçalho abaixo como base para sua implementação. Você precisará criar um arquivo `counter.c` para implementar as funções e um `main.c` para realizar os testes de performance.

Listing 1: Arquivo de Esqueleto: counter.h

```
#ifndef __COUNTER_H__
#define __COUNTER_H__

#include <pthread.h>

// Defina de acordo com seu sistema, ou um valor fixo.
#define NUMCPUS 8

// ESTRUTURA PARA FASE 1
typedef struct __counter_precise_t {
    int value;
    pthread_mutex_t lock;
} counter_precise_t;

// ESTRUTURA PARA FASE 2
typedef struct __counter_approx_t {
    int global;
    pthread_mutex_t glock;
    int local[NUMCPUS];
    pthread_mutex_t llocks[NUMCPUS];
    int threshold;
} counter_approx_t;

// Funcoes para o contador preciso
void Counter_Init(counter_precise_t *c);
void Counter_Increment(counter_precise_t *c);
int  Counter_GetValue(counter_precise_t *c);

// Funcoes para o contador aproximado
void CounterApprox_Init(counter_approx_t *c, int threshold);
void CounterApprox_Update(counter_approx_t *c, int threadID, int
   amount);
int  CounterApprox_GetValue(counter_approx_t *c);

#endif // __COUNTER_H__
```

O Que Entregar

1. **Código-Fonte:** Os arquivos `counter.c` e `main.c` completos e funcionais.

2. **Relatório de Análise (PDF):** Um breve relatório contendo:

    - **Gráfico 1 (Escalabilidade):** Um gráfico comparando o tempo de execução do Contador Preciso vs. Aproximado em função do número de threads. (Similar à Figura 29.5)
    - **Gráfico 2 (Influência do Limiar S):** Um gráfico mostrando como o tempo de execução do Contador Aproximado varia em função do valor de `S`, com um número fixo de threads. (Similar à Figura 29.6)
    - **Análise Escrita:** Uma breve análise explicando os resultados observados em ambos os gráficos, discutindo os conceitos de contenção de lock, escalabilidade e o trade-off entre performance e precisão.