Laboratório 4
Comunicação entre threads via memória compartilhada
Corrida de dados e Violação de atomicidade
Alocação dinâmica de tarefa para as threads

Programação Concorrente (ICP-361) 2025-2
Profa. Silvana Rossetto

[1]Instituto de Computação/UFRJ

Introdução

O objetivo deste Laboratório é introduzir o uso de variáveis compartilhadas para permitir a comunicação entre as threads de uma aplicação e mostrar quais benefícios e desafios essa comunicação pode trazer.

Para cada atividade, siga o roteiro proposto e responda às questões colocadas.

Atividade 1

Objetivo: Mostrar um exemplo simples de programa com uma variável compartilhada entre threads e o problema de **corrida de dados**.

Roteiro:

1. Abra o arquivo **soma.c** e entenda o que ele faz. **Qual saída é esperada para o programa (valor final da variável *soma*)?**
2. Execute o programa **várias vezes** e observe os resultados impressos na tela (acompanhe a explanação da professora).
3. **Os valores impressos foram diferentes a cada execução? Por que?**
4. **É possível que a variável soma termine com valor acima de 200000 quando executamos com 2 threads? Por que?**

Atividade 2

Objetivo: Introduzir o uso de *locks* provido pela biblioteca Pthreads e mostrar seu uso para tratar o problema de **corrida de dados**.

Roteiro:

1. Abra o arquivo **soma-lock.c** e compreenda como locks são usados para implementar a **exclusão mútua** entre as threads (acompanhe a explanação da professora).
2. Execute o programa **várias vezes**. **Os valores impressos foram sempre o valor esperado? Por que?**
3. Altere o número de threads e avalie os resultados.

Atividade 3

Objetivo: Mostrar um exemplo de programa com erro de **violação de atomicidade** e como usar exclusão mútua com *locks* para tratá-lo.

Roteiro:

1. Abra o arquivo **soma-lock-atom.c** e compreenda o que a thread `extra` faz (acompanhe a explanação da professora).
2. **Quais valores devem ser impressos na saída do programa?** (todos os múltiplos de 10 dentro da faixa de valores assumidos pela variável 'soma'? alguns múltiplos de 10? os primeiros 10000 múltiplos de 10? os últimos 10000 múltiplos de 10?).
3. Execute o programa **várias vezes** e observe os resultados. **Os valores impressos foram os valores esperados? Por que?**
4. Altere o programa usando locks para resolver o problema de **violação de atomicidade**. **Agora quais valores espera-se que sejam impressos na saída do programa?**
5. Execute o programa **várias vezes** e observe os resultados. **O problema foi resolvido? A saída varia de uma execução para outra? Justifique.**

Atividade 4

Objetivo: Praticar o uso da concorrência. Dada uma sequência consecutiva de números naturais (inteiros positivos) de 1 a n (n muito grande), identificar todos os **números primos** dessa sequência e retornar a **quantidade total de números primos encontrados**. Use a função abaixo para verificar a primalidade de um número:

```
int ehPrimo(long long int n) {
   int i;
      if (n<=1) return 0;
      if (n==2) return 1;
      if (n%2==0) return 0;
      for (i=3; i<sqrt(n)+1; i+=2)
         if(n%i==0) return 0;
      return 1;
```

Roteiro

1. Implemente uma **versão concorrente** para esse problema, dividindo a tarefa entre as threads. O número de elementos da sequência (n) e o número de threads (t) devem ser informados na linha de comando.
   IMPORTANTE: Distribua a tarefa entre as threads de forma dinâmica (como discutido na aula do dia 26 de agosto): **cada thread pega o próximo número da sequência para avaliar a sua primalidade** ATENÇÃO: os valores da série NÃO devem ser colocados em um vetor, devem ser obtidos incrementalmente em tempo de execução.
2. OBS.: defina a variável n do tipo **long long int** e use a função **atoll() para converter o valor recebido do usuário (string) para long long int**.
3. **Certifique-se que a sua solução está correta**.
4. Inclua **tomadas de tempo no código** (toda a parte concorrente da solução).
5. Execute o programa várias vezes, variando o valor de N (experimente 10^3 e 10^6 e outros se for o caso); e para cada valor de n, varie a quantidade de threads.
6. Para cada configuração, execute o programa ao menos 5 vezes e registre o tempo médio dessas execuções.

7. Calcule a **aceleração e a eficiência** alcançada em cada configuração, tomado como base (execução sequencial) a execução com 1 thread. Para avaliar o ganho de desempenho, execute o programa concorrente com 2, 4 e 8 threads. Calcule a **aceleração (A)** fazendo:

$$A(n,t) = T_s(n,1)/T_p(n,t)$$

   e a **eficiência (E)** fazendo:

$$E(n,t) = A(n,t)/t$$

   n é a quantidade de elementos da sequência e t é a quantidade de threads usadas na execução.
8. Registre todos os dados levantados e calculados em uma TABELA, gere os **gráficos de tempo, aceleração e eficiência**. AVALIE: **os resultados e traga suas observações para próxima aula.**

Entrega do laboratório:

Este laboratório não tem entrega.