Trabalho Parcial 01

Sistemas Operacionais

Prof. Pedro Botelho

26º Junho de 2025

1. (6 pontos) Um problema clássico na sincronização de processos é o problema **Produtor-Consumidor**. Esse problema envolve dois ou mais processos (ou threads): um ou mais **produtores** que geram dados e os colocam em um *buffer* (uma região de memória, como um vetor ou uma lista), e um ou mais **consumidores** que retiram dados desse *buffer*. O desafio principal é garantir que o produtor não tente adicionar dados a um *buffer* cheio e que o consumidor não tente retirar dados de um *buffer* vazio, evitando condições de corrida e inconsistências de dados.

   (a) Seu trabalho será projetar um sistema com múltiplas *threads* de um **gerenciador centralizado de caixas** de uma loja. Sempre que um caixa se conecta ao servidor, uma *thread* é criada para gerenciá-lo. De forma a simplificar, o sistema já inicia com os três caixas conectados (devendo apenas criar suas *threads*), além da *thread* de gerenciamento, não sendo preciso tratar conexão, apenas a sincronização entre *threads*. Sendo assim, existem 5 *threads*: 3 produtoras e 1 consumidora. A única função da *thread* principal é criar as *threads* produtoras e consumidoras e esperar que elas terminem.

   (b) Um *buffer* de 5 espaços deve ser usado (um vetor simples pode ser usado), de forma que as *threads* produtoras possam inserir os valores das vendas e a consumidora possa processá-lo (não esqueça de gerenciar seu espaço usando um **semáforo**). O processamento deverá ser sincronizado, de forma que o acesso ao *buffer* seja protegido (via **mutex**). Ainda, a *thread* consumidora deve ser avisada pelas *threads* produtoras sempre que dados novos forem inseridos no *buffer*, para que possa processá-los (vide ***pthread_cond***).

   (c) Cada uma das ***threads*** **produtoras** deve produzir entre 20 e 30 valores (passe a quantidade como parâmetro para as *threads* na sua criação, e recupere na função, dessa forma apenas uma função pode ser usada para as três *threads*) relativos ao preço dos produtos, entre 1 a 1000 reais (use valores aleatórios, via funções **srand()** e **rand()**) e colocá-los no *buffer* (se não cheio), devendo sinalizar a *thread* consumidora e imprimir na tela o valor do produto, seu TID (*thread* ID) e o número da iteração atual, indicando também que é produtora. Ao final, as *threads* produtoras devem esperar um certo tempo antes de produzirem novos valores (use valores aleatórios como *delay*, entre 1 e 5 segundos). Quando as *threads* produtoras produzirem todos os valores, devem finalizar e informar à *thread* consumidora que finalizaram.

(d) A ***thread* consumidora** irá consumir os valores postos no *buffer* compartilhado quando as *threads* produtoras sinalizarem, porém com restrições: essa *thread* irá retirar os 5 elementos do *buffer* (quando o buffer estiver cheio) e realizar uma média aritmética. Assim, irá imprimir a média na tela, junto a seu TID e a iteração corrente, voltando a esperar o *buffer* ficar cheio novamente.

(e) Abaixo está um exemplo da estrutura geral do código:

```c
void *producer(void *args) {
    int n = *((int *) args);

    while(n-- > 0) {
        // acessar buffer compartilhado (produzir)
        // sinalizar dados
        // imprimir TID/dados
        // esperar por um tempo aleatorio
    }
    // imprimir que finalizou
}

void *consumer(void *args) {
    while( /* tem produtoras ainda? */ ) {
        // esperar 5 dados
        // acessar buffer compartilhado (consumir)
        // imprimir TID/dados
    }
    // imprimir que finalizou
}

int main(void) {
    // criar threads produtoras
    // criar thread consumidora
    // esperar threads terminarem
}
```

(f) Abaixo está um exemplo de saída do programa. É importante que as *threads* imprimam uma mensagem informando que finalizaram:

```
// ...
// (P) TID: 1000 | VALOR: R$ 210 | ITERACAO: 25
// (P) TID: 1001 | VALOR: R$  32 | ITERACAO: 20
// (P) TID: 1001 finalizou
// (P) TID: 1002 | VALOR: R$   5 | ITERACAO: 12
// (C) TID: 1003 | MEDIA: R$ 620 | ITERACAO: 30
// ...
```

(g) Repita a questão com 6 *threads* produtoras e 2 consumidoras. O projeto ficou mais complicado? Se sim, quais desafios você enfrentou? Forneça uma justificativa.

2. (4 pontos) De acordo com Gottfried Wilhelm Leibniz, o valor de π pode ser aproximado por uma série:

$$\pi = 4 \cdot \sum_{k=0}^{\infty} \frac{(-1)^k}{2k+1} = 4 \cdot \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \frac{1}{9} - \cdots\right)$$

Essa série é conhecida como **Fórmula de Leibniz**. Calcular esse valor sequencialmente pode levar bastante tempo (considerando uma quantidade considerável de termos), sendo mais interessante dividir o processamento em várias *threads*. Baseado nisso, faça uma análise do uso de *multithreading* para o processamento da soma, comparando o tempo empregado no processamento sequencial e paralelo.

(a) Primeiramente, faça um programa (q3_1.c) que implemente a série sequencialmente. Considere que a série possui 2.000.000.000 termos (ao invés de infinito). Ao final do programa, o valor calculado e o tempo empregado deve ser mostrado.

(b) Após isso, faça um outro programa (q3_2.c) que implemente a série de forma paralela, usando várias *threads*. A *thread* principal deve criar 16 *threads*, que vão calcular uma parte da série, isto é, somar n/16 termos, e, ao final, somar esse resultado em uma variável que deve conter todos os termos. A *thread* principal deve esperar que as outras terminem de forma calcular o valor final da série (multiplicar o somatório por 4), mostrar o tempo empregado por cada *thread*, o tempo empregado pelo processo e a soma dos tempos das *threads*.

(c) Mantenha no escopo global uma variável que irá manter a soma de todos os termos. Após cada *thread* realiza sua soma parcial, acumule nessa variável. Não esqueça de protegê-la de acesso simultâneo. Ao final, mostre o valor obtido com 9 casas decimais de precisão.

(d) Abaixo está um exemplo da estrutura geral do código usando *threads*:

```
1  #define NUM_TERMS 2000000000
2  #define NUM_THREADS 16
3  #define PARTIAL_NUM_TERMS ((NUM_TERMS)/(NUM_THREADS))
4
5  double partialFormula(int first_therm) {
6      // A funcao ira processar PARTIAL_NUM_TERMS termos
7      const int num_terms = first_therm + PARTIAL_NUM_TERMS;
8
9      // Aproxima o pi, de first_therm ate num_terms - 1
10     double pi_approximation = 0;
11     double signal = 1.0;
```

```
12      for (int k = first_therm; k < num_terms; k++) {
13          pi_approximation += signal/(2*k + 1);
14          signal *= -1.0;
15      }
16
17      return pi_approximation;
18  }
19
20  void *partialProcessing(void *args) {
21      int first_therm = *((int *) args);
22      // obter tempo de inicio
23      int sum = partialFormula(first_therm);
24      // acessar buffer compartilhado
25      // obter tempo de fim
26      // mostrar TID e tempo empregado
27  }
28
29  int main(void) {
30      // obter tempo de inicio
31      for(int i = 0; i < NUM_THREADS; i++) {
32          // criar threads parciais
33      }
34      // esperar threads terminarem
35      // obter tempo de fim
36      // mostrar resultado e tempo emprego
37  }
```

(e) Abaixo está um exemplo de saída do programa. É importante que as *threads* imprimam uma mensagem ao final informando seu TID e o tempo empregado:

```
1  // Total Processo (Sequencial): 23.20s
2  // ...
3  // TID: 1000: 2.33s
4  // TID: 1001: 2.17s
5  // ...
6  // TID: 1014: 1.89s
7  // TID: 1015: 3.01s
8  // Total Processo (Paralelo): 3.19s
9  // Total Threads: 43.21s
```

(f) Repita a análise com um número diferente de *threads* parciais (2, 4, 8, 16, 32, 64, etc) e veja os resultados. Até onde o aumento no número de *threads* impacta suficientemente na execução?

Boa sorte!