Universidade de Brasília

Ciência da Computação

Programação Concorrente

João Victor Pereira Vieira - 211036114

Restaurant_With_Threads

Brasília-DF (Campus Darcy Ribeiro)

2025.2

Introdução

Este relatório descreve a implementação de um problema clássico de programação concorrente: a simulação de um sistema de restaurante. Em ambientes computacionais modernos, a execução concorrente de tarefas é fundamental para a otimização de recursos e a responsividade de aplicações. Contudo, a concorrência introduz desafios inerentes, como condições de corrida, deadlocks e starvation, que podem comprometer a integridade dos dados e a estabilidade do sistema.

O objetivo deste trabalho é demonstrar a aplicação prática e a solução de tais desafios utilizando os mecanismos de sincronização fornecidos pela biblioteca POSIX Pthreads em linguagem C. O cenário do "RESTAURANTE-WITH-THREADS" serve como um modelo ilustrativo onde múltiplas entidades (clientes, garçons, cozinheiros, etc.) operam e interagem de forma independente, mas precisam coordenar suas ações para garantir o funcionamento harmonioso e correto do sistema.

Ao longo do desenvolvimento, foram exploradas diversas ferramentas de sincronização, incluindo mutexes para proteção de acesso a recursos compartilhados, semáforos para controle de acesso a pools de recursos e sinalização de eventos, e variáveis condicionais para gerenciar a espera e notificação entre threads. A presente formalização do problema, descrição do algoritmo, e análise dos resultados buscarão validar a eficácia dessas técnicas na construção de um sistema concorrente robusto e funcional.

1. Formalização do problema

O problema proposto envolve a simulação de um restaurante, um ambiente inerentemente concorrente, onde múltiplas entidades (funcionários e clientes) interagem simultaneamente para alcançar objetivos individuais e coletivos. O objetivo principal é modelar a comunicação e sincronização entre essas entidades utilizando mecanismos de concorrência, como threads, mutexes, semáforos e variáveis condicionais, a fim de evitar condições de corrida, deadlocks e starvation, garantindo o funcionamento lógico e eficiente da simulação.

Entidades Concorrentes:

A simulação é composta por um total de 8 tipos de entidades, cada uma representada por uma ou mais threads, desempenhando papéis distintos:

1. **Gerente do Dia:** Thread principal responsável por coordenar a operação diária. Ele inicia o ambiente, “contrata” e supervisiona as outras threads, monitora o tempo de funcionamento do restaurante, gerencia o fluxo de clientes (até um limite máximo ou fim do tempo) e consolida os resultados financeiros ao final do dia.

2. **Timer do Restaurante:** Uma thread dedicada que funciona como o "relógio" do dia. Ela é iniciada pelo Gerente e corre em paralelo. Seu único trabalho é dormir pelo tempo de funcionamento predeterminado e, ao acordar, sinalizar o fechamento do restaurante, impedindo a entrada de novos clientes.

3. **Cliente:** Múltiplas threads que chegam ao restaurante com o objetivo de serem atendidas. Cada cliente busca uma mesa, faz um pedido, espera o prato, come e, finalmente, paga e sai, podendo deixar a mesa limpa ou suja. Clientes podem desistir se a espera for muito longa ou se o restaurante fechar.

4. **Gestor de Mesas:** Uma thread dedicada a gerenciar a disponibilidade de mesas. Ele é ativado quando o restaurante está cheio e há clientes esperando, adicionando novas mesas até o limite máximo para acomodar a demanda. Se não tiver mesas para alocar, ele não irá fazer milagre, e o cliente terá que esperar até que tenha, ou ir embora
.
5. **Cozinheiro:** Múltiplas threads que preparam os pratos. Eles verificam o estoque de ingredientes e preparam o prato solicitado, notificando o garçom quando o pedido está pronto. Podem precisar acionar o Estoquista se os ingredientes acabarem.

6. **Estoquista:** Uma thread que repõe o estoque de ingredientes da cozinha. Ele é acionado pelos Cozinheiros quando há falta de itens e, após a reposição, notifica os Cozinheiros para que retomem suas tarefas.

7. **Responsável pela Limpeza:** Uma thread que mantém o ambiente do restaurante higienizado. Ele é acionado quando clientes deixam mesas sujas e, após a limpeza, libera a mesa para novos clientes.

Recursos Compartilhados e Desafios de Sincronização:

Diversos recursos são acessados e modificados por múltiplas threads, criando a necessidade crítica de mecanismos de sincronização:

- **Mesas do Restaurante:** Representam os espaços onde os clientes podem comer. São limitadas e seu status (ocupada/livre, limpa/suja) é compartilhado e modificado por Clientes, Gestor de Mesas e Responsável pela Limpeza.

- **Filas de Pedidos:** Listas onde os Clientes depositam seus pedidos e os Garçons os retiram para levar à cozinha, e onde os Cozinheiros entregam os pratos prontos para os Garçons.

- **Estoque de Pratos:** Array que representa a quantidade de ingredientes disponíveis para cada tipo de prato. É acessado por Cozinheiros (para consumir) e pelo Estoquista (para repor).

- **Função de Geração de Números Aleatórios (**rand_safe**):** Como rand() não é intrinsecamente thread-safe em alguns sistemas e gera números repetitivos em ambientes concorrentes sem proteção, seu acesso deve ser sincronizado para garantir aleatoriedade genuína.

Os principais desafios de sincronização incluem evitar que vários clientes tentem ocupar a mesma mesa simultaneamente, que vários garçons peguem o mesmo pedido, que cozinheiros acessem o estoque de forma inconsistente, e que o gerente encerre o dia antes que todos os processos pendentes sejam concluídos.

2. Descrição do Algoritmo

Para solucionar os desafios de sincronização delineados na seção anterior, a arquitetura da solução foi baseada em três mecanismos principais da biblioteca Pthreads: **Mutexes**, **Semáforos** e **Variáveis Condicionais**. Cada um foi escolhido para resolver um tipo específico de problema de concorrência.

OBS: O código ficou muito extenso, então não vou abordar muitas imagens dele aqui, logo essa parte seguirá mais da parte lógica. Segue o vídeo mostrando o programa em si com detalhes e sua devida apresentação: https://youtu.be/VGL7m8LxYHE

2.1 Mutexes para Exclusão Mútua

O mecanismo mais fundamental utilizado foi o pthread_mutex_t (lock) para garantir a exclusão mútua e prevenir condições de corrida em "seções críticas". Mutexes foram aplicados para proteger o acesso a dados simples e recursos compartilhados que não podem ser modificados por múltiplas threads ao mesmo tempo.

- **Estado do Restaurante:** O mutex_restaurante protege variáveis de estado globais como mesas_ocupadas, clientes_esperando e a flag restaurante_fechado.
- **Recursos Específicos:** Mutexes dedicados, como mutex_estoque, mutex_lucro e as travas das filas de pedidos (mutex_fila_...), foram usados para granularizar a proteção e reduzir a contenção, permitindo que um Cozinheiro acesse o estoque sem travar um Garçom que acessa uma fila.
- **Função Não-Thread-Safe:** Um mutex_rand_seed foi usado para serializar o acesso à função rand_safe, protegendo sua semente interna e garantindo a aleatoriedade em um ambiente concorrente.

2.2 Semáforos como Contadores (Padrão Produtor-Consumidor)

O padrão Produtor-Consumidor foi implementado extensivamente usando sem_t (semáforos) para gerenciar filas de "trabalho" e desacoplar as threads. O semáforo atua como um contador atômico de tarefas pendentes, permitindo que as threads "consumidoras" durmam (sem_wait) até que uma "produtora" sinalize que há trabalho (sem_post).

- Pedidos: O Cliente (produtor) sinaliza sem_clientes_chamando, e o Garçom (consumidor) espera. O Garçom (produtor) sinaliza sem_pedidos_pendentes, e o Cozinheiro (consumidor) espera.
- Sinalização de Eventos: O Cliente (produtor) sinaliza sem_limpeza_necessaria para acordar a thread de Limpeza (consumidor).

2.3 Variáveis Condicionais para Sincronização Complexa

Enquanto semáforos resolvem contagens simples, as pthread_cond_t (variáveis condicionais) foram usadas para lógicas de espera mais complexas, onde uma thread deve aguardar por uma *condição de estado* específica antes de prosseguir. Elas sempre são usadas em conjunto com um mutex.

- **Espera por Recursos (Cozinheiro/Estoquista):** O Cozinheiro não pode simplesmente "esperar por um sinal"; ele deve verificar se o estoque é zero. Ele entra em um loop while (estoque[prato] == 0) e chama pthread_cond_wait(&cond_estoque_reposto, &mutex_estoque). Isso atomicamente libera o mutex e o coloca para dormir. O Estoquista, ao repor, chama pthread_cond_broadcast(&cond_estoque_reposto), acordando todos os cozinheiros, que então re-verificam o while para ver se podem prosseguir.
- **Lógica de Fim de Dia (Gerente):** A sincronização mais complexa é a do gerente_do_dia_func. Ele espera em pthread_cond_wait(&cond_todos_clientes_sairam) dentro de um loop while(1). Ele é

acordado por *qualquer* evento de saída (Cliente saindo, Limpeza terminando, ou o Timer estourando). Ao acordar, ele reavalia as duas condições de término do dia: (1) se todos os clientes criados já saíram ou (2) se o timer acabou e todas as mesas estão vazias. Se nenhuma for verdadeira, ele volta a dormir.

- **Espera com Timeout (Cliente):** O Cliente usa pthread_cond_timedwait(&cond_mesa_disponivel, ...) para esperar por uma mesa, mas com um limite de tempo, implementando a lógica de "desistir".

Esta abordagem em camadas garante que dados simples sejam protegidos (mutexes), que o fluxo de trabalho seja eficiente (semáforos) e que lógicas de estado complexas sejam tratadas corretamente (variáveis condicionais).

3. Resultados e Discussão

A execução do algoritmo de simulação do restaurante confirmou o funcionamento correto dos mecanismos de sincronização implementados. Através dos logs de saída, foi possível observar que as 8 entidades concorrentes (Gerente, Timer, Clientes, Garçons, Cozinheiros, Gestor de Mesas, Estoquista e Limpeza) interagiram conforme o esperado, sem a ocorrência de deadlocks ou corrupção de dados nas variáveis compartilhadas, como o lucro_dia ou o estoque.

Os principais resultados observados incluem:

- **Tratamento de Condições de Borda:** O sistema lidou com sucesso com cenários de alta contenção, como a falta de estoque (onde os Cozinheiros corretamente esperaram pelo Estoquista) e a falta de mesas (onde os Clientes desistiram com `ETIMEDOUT` ou aguardaram a Limpeza).
- **Aleatoriedade Garantida:** A implementação da função rand_safe com um mutex dedicado (mutex_rand_seed) se provou crucial, gerando pedidos de pratos e preços variados, o que resultou em simulações dinâmicas e lucros diários distintos.
- **Correção da Lógica de Sincronização:** A lógica de fim de dia do Gerente, baseada em duas condições (clientes_que_sairam_total ou o timer_restaurante_fechado), funcionou robustamente . Isso garantiu que o dia só encerrasse após todos os clientes (incluindo aqueles que desistiram) serem contabilizados e todas as mesas liberadas, evitando o encerramento prematuro que foi identificado durante o desenvolvimento.

A discussão central dos resultados é que a escolha de diferentes ferramentas de sincronização (mutexes, semáforos e CVs) para diferentes problemas foi a chave para o sucesso do projeto. O uso de semáforos para filas de trabalho (como sem_limpeza_necessaria) e variáveis condicionais para esperas complexas (como a do Gerente em cond_todos_clientes_sairam) permitiu um design mais limpo e eficiente do que tentar resolver todos os problemas apenas com mutexes.

Conclusão

Este projeto demonstrou com sucesso a implementação de um sistema concorrente complexo para simular o funcionamento de um restaurante. O objetivo de modelar a comunicação e resolver condições de corrida entre múltiplos processos (threads) através de memória compartilhada foi atingido utilizando a biblioteca POSIX Pthreads.

O desenvolvimento do trabalho reforçou a importância crítica de selecionar o mecanismo de sincronização adequado para cada tarefa. Mutexes foram essenciais para garantir a exclusão mútua em dados simples (como o estoque e o lucro); semáforos se mostraram ideais para o padrão produtor-consumidor (como nas filas de pedidos); e as variáveis condicionais foram indispensáveis para gerenciar esperas baseadas em estados complexos, como o término do expediente.

Ao final, o projeto não apenas valida a aplicação prática dos conceitos teóricos de programação concorrente, mas também serve como um estudo de caso sobre como a decomposição de um problema complexo em entidades concorrentes menores e bem definidas permite a construção de um sistema robusto, eficiente e correto.

Bibliografia

ALCHIERI, Eduardo Adílio Penlison. **Programação Concorrente**: aulas, slides e códigos. Brasília: Universidade de Brasília, 2025. Material da disciplina . Plataforma Aprender3. Acesso restrito a alunos da disciplina.

INTERNATIONAL ORGANIZATION FOR STANDARDIZATION. **ISO/IEC 9899:2011**: Information technology — Programming languages — C. Genebra: ISO, 2011.

SILBERSCHATZ, Abraham; GALVIN, Peter B.; GAGNE, Greg. **Operating System Concepts**. 10. ed. Hoboken: John Wiley & Sons, 2018.

THE OPEN GROUP. **IEEE Std 1003.1-2017 (POSIX.1-2017)**. New York: IEEE, 2018. Disponível em: https://pubs.opengroup.org/onlinepubs/9699919799/. Acesso em: 01 nov. 2025.