Projeto final - Processamento de Pedidos em uma Cozinha de Restaurante

Vitor Guedes Frade/221017130 *

Universidade de Brasília, 30 de agosto de 2024

Figura 1

1 Introdução

No contexto atual da indústria gastronômica, a eficiência operacional dentro de uma cozinha de restaurante é essencial para garantir a qualidade do serviço e a satisfação do cliente. Com o aumento da demanda e a complexidade dos cardápios, a coordenação e o gerenciamento dos recursos da cozinha se tornaram desafios significativos. A implementação de soluções que possam otimizar o fluxo de trabalho, reduzir o tempo de preparo dos pratos e evitar gargalos operacionais é, portanto, uma prioridade para estabelecimentos que buscam excelência em seus serviços.

Este relatório apresenta o desenvolvimento de um programa de simulação para o Processamento de Pedidos em uma Cozinha de Restaurante, utilizando programação concorrente. O projeto foi concebido com o objetivo de modelar a dinâmica de trabalho em uma cozinha, onde múltiplos cozinheiros precisam acessar simultaneamente recursos limitados, como fogões, panelas, fornos e bancadas de preparo, para preparar diferentes pratos do cardápio.

A implementação foi realizada em linguagem C, utilizando a biblioteca POSIX para o gerenciamento de threads, mutexes e semáforos, permitindo a simulação de um ambiente de cozinha com concorrência realista entre os cozinheiros. Ao longo do desenvolvimento, foram abordados conceitos fundamentais de programação concorrente, como sincronização de threads, prevenção de deadlocks e starvation,e gerenciamento eficiente de recursos compartilhados.

O relatório detalha os desafios encontrados durante o desenvolvimento, as estratégias adotadas e os resultados obtidos. Acredita-se que o programa desenvolvido seja uma ferramenta útil tanto para a análise e otimização de processos em cozinhas de restaurantes quanto para o aprendizado e compreensão de conceitos de programação concorrente.

2 Formalização do Problema Proposto

O problema proposto consiste em modelar e simular o processamento de pedidos em uma cozinha de restaurante, onde diversos cozinheiros compartilham recursos limitados para preparar diferentes pratos de um cardápio. Este problema é caracterizado pela necessidade de sincronização e coordenação entre as tarefas dos cozinheiros, de modo a evitar conflitos e otimizar o uso dos recursos disponíveis.

*221017130@aluno.unb.br

2.1 Descrição dos Elementos do problema

- Cozinheiros: Representados como threads, os cozinheiros são responsáveis por preparar os pedidos. Cada cozinheiro deve realizar uma sequência de operações pré definidas pela receita para completar um prato, onde um conheiro é capaz de realizar qualquer receita desde que ele inicie essa tarefa. O número de cozinheiros presentes no restaurante é determinado e alteravel antes da simulação, onde a realização de um pedido sera iniciada somente se tiverem cozinheiros desocupados.

- Pedidos: Um pedido consiste em uma sequência de operações que devem ser realizadas por um cozinheiro. Cada pedido está associado a uma receita específica que determina os recursos necessários e a ordem de utilização desses recursos. Os pedidos são determinados por uma estrutura de dados que o cozinheiro recebe especificando a receita que deve ser ralizada, assim ocorrendo concorrencia pelo trabalho dos cozinheiros.

- Recursos da Cozinha: Os recursos da cozinha são limitados e destinados ao uso exclusivo dos cozinheiros. Eles são representados por estruturas de dados que gerenciam as respectivas concorrências entre os cozinheiros. Cada recurso pode ser acessado por um único cozinheiro durante um período específico, enquanto uma receita solicitada em um pedido está sendo preparada.

  - Fogões
  - Forno
  - Panelas
  - Bancadas
  - Grill

2.2 Objetivo do programa

O objetivo do sistema é garantir que todos os pedidos sejam processados de forma eficiente, minimizando o tempo de espera para cada prato e evitando conflitos pelo uso dos recursos da cozinha. Para atingir este objetivo, o programa deve ser capaz de:

- Sincronizar o acesso aos recursos compartilhados, garantindo que apenas um cozinheiro utilize um recurso específico a qualquer momento, e todas receitas solicitadas sejam finalizadas.

- Coordenar a execução das operações de forma que a ordem de preparo dos ingredientes e o uso dos recursos respeitem a sequência necessária de cada receita.

- Prevenir situações de deadlock, onde múltiplos cozinheiros possam ficar bloqueados esperando por recursos que nunca serão liberados, ou quando um processo espera indefinidamente por um recurso e não consegue concluir a sua função.

3 Descrição do Algoritmo Desenvolvido para Solução do Problema Proposto

O programa de simulação do problema proposto possui a seguinte estrutura base de implementação:

3.1 Definições

Listing 1: *Definição da quantidade de recursos*

```
#define NUM_FOGAO 2
#define NUM_PANELA 3
#define NUM_FORNO 1
#define NUM_BANCADA 2
#define NUM_GRILL 1
#define NUM_RECEITAS 6
#define NUM_COZINHEIROS 4
```

O trecho acima define os limites de recursos disponíveis para uso na simulação em questão, a fim de verificar diferentes desempenhos no contexto empregado.

Listing 2: *Definição dos recursos com suas respectivas ferramentas necessárias para implementação*

```
// Sem foros para os recursos
sem_t sem_fogao;
sem_t sem_panela;
sem_t sem_forno;
sem_t sem_bancada;
sem_t sem_grill;
sem_t sem_cozinheiros;


pthread_mutex_t print_lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t recurso_lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t cozinheiros_lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t cozinheiros_disponiveis_lock =
    PTHREAD_MUTEX_INITIALIZER;
pthread_mutex_t queue_lock =
    PTHREAD_MUTEX_INITIALIZER;

pthread_cond_t cozinheiro_cond =
    PTHREAD_COND_INITIALIZER;
pthread_cond_t queue_cond =
    PTHREAD_COND_INITIALIZER;

pthread_t cozinheiro_tid[NUM_COZINHEIROS];

// Vetor para rastrear IDs de cozinheiros
int id_disponiveis[NUM_COZINHEIROS];
```

Os recursos foram definidos com semáforos para sinalizar suas respectivas diponibilidades para uso. Para auxilio e controle de concorrencia foram definidas e inicializadas chaves de acesso a região de execlusão mutua (locks) e variaveis condicionais, onde serão empregadas nas funções que ocorrem as concorrencias.

3.2 Estruturas de dados

Listing 3: *Estruturas de dados*

```
typedef struct {
    sem_t** semaphores;  // Array de ponteiros
        para sem foros
    char** resource_names;  // Array de nomes
        dos recursos
    int size;            // Tamanho do array de
         sem foros e nomes
} SemaphoreArray;

typedef struct {
    char* key;               // Chave (por
        exemplo, string)
    SemaphoreArray value;    // Valor (array de
         sem foros)
} DictionaryEntry;

typedef struct {
    DictionaryEntry* entries; // Array de
        entradas do dicion rio
    int size;                 // N mero de
        entradas no dicion rio
} Dictionary;

typedef struct {
    int id;              // ID do cozinheiro
    char* receita;       // Receita que o
        cozinheiro vai preparar
} CozinheiroInfo;

typedef struct Node {
    int id;
    struct Node* next;
} Node;

Dictionary receitas;
Node* head = NULL;
Node* tail = NULL;
```

Foram implementadas as estruturas acima que simulam um dicionário, onde o cardapio do restaurante será definido de maneira que as receitas são as chaves, e os recursos necessarios estão em um array nas suas respectivas ordem de uso. Cada cozinheiro thread de cozinheiro recebera uma estrutura que especifica o id do cozinheiro em questão, e tambem informa o nome da receita que deve ser realizada no momento. Para ordenar os cozinheiros quanto a sua disponibilidade, foi utilizada uma fila, então foi definida uma estrutura de nós para indentificar cada um.

3.3 Funções auxiliares

Listing 4: *Incialização das estruturas do tipo dicionário*

```

void initDictionary(Dictionary* dict, int
    initialSize) {
    dict->entries = (DictionaryEntry*)malloc(
        initialSize * sizeof(DictionaryEntry));
    dict->size = 0;
}
```

A função initDictionary inicializa dicionários alocando espaço de memória com o tamanho passado como parametro.

Listing 5: *Contador de semáforos em um array*

```

int countSemaphores(sem_t** semaphoreArray) {
    int count = 0;
    while (semaphoreArray[count] != NULL) {
```

```
        count++;
    }
    return count;
}
```

Listing 6: *Definição de recursos da receita*

```

void addSemaphoreArray(Dictionary* dict, const
    char* key, sem_t** semaphoreArray, const
    char** resourceNames) {
    dict->entries[dict->size].key = strdup(key)
        ;
    dict->entries[dict->size].value.semaphores
        = semaphoreArray;
    dict->entries[dict->size].value.size =
        countSemaphores(semaphoreArray);

    // Alocar e copiar os nomes dos recursos
    dict->entries[dict->size].value.
        resource_names = malloc(dict->entries[
        dict->size].value.size * sizeof(char*))
        ;
    for (int i = 0; i < dict->entries[dict->
        size].value.size; i++) {
        dict->entries[dict->size].value.
            resource_names[i] = strdup(
            resourceNames[i]);
    }

    dict->size++;
}
```

Listing 7: *Obter recursos da receita na sua respectiva ordem*

```

SemaphoreArray* getSemaphoreArray(Dictionary*
    dict, const char* key) {
    for (int i = 0; i < dict->size; i++) {
        if (strcmp(dict->entries[i].key, key)
            == 0) {
            return &dict->entries[i].value;
        }
    }
    return NULL; // Se a chave n o for
        encontrada
}
```

Listing 8: *Limpa a memória do dicionário criado anteriormente*

```

void freeDictionary(Dictionary* dict) {
    for (int i = 0; i < dict->size; i++) {
        free(dict->entries[i].key);
        free(dict->entries[i].value.semaphores)
            ;
    }
    free(dict->entries);
}
```

Listing 9: *Adiciona um novo elemento na fila*

```
void enqueue(int id) {
    Node* newNode = (Node*)malloc(sizeof(Node))
        ;
    newNode->id = id;
    newNode->next = NULL;

    pthread_mutex_lock(&queue_lock);
        if (tail == NULL) {
            head = tail = newNode;
        } else {
            tail->next = newNode;
            tail = newNode;
        }
        pthread_cond_signal(&queue_cond);
    pthread_mutex_unlock(&queue_lock);
}
```

Listing 10: *Remove o próximo elemento a sair da fila*

```
int dequeue() {
    pthread_mutex_lock(&queue_lock);
        while (head == NULL) {
            pthread_cond_wait(&queue_cond, &
                queue_lock);
        }
        Node* temp = head;
        int id = temp->id;
        head = head->next;
        if (head == NULL) {
            tail = NULL;
        }
        free(temp);
    pthread_mutex_unlock(&queue_lock);
    return id;
}
```

3.4 FUNÇÕES PRINCIPAIS

As funções a seguir são responsáveis pelos processos e threads, onde ocorrem as instâncias das estruturas de dados e recursos, assim como a simulação do problema proposto.

3.4.1 FUNÇÃO `main`

A função `main` inicializa os recursos e constrói as respectivas estruturas de dados. Posteriormente, faz os pedidos de maneira randômica e cria threads para realizar a receita de acordo com a disponibilidade dos cozinheiros.

Listing 11: *Função `main` completa*

```

int main() {
    initDictionary(&receitas, NUM_RECEITAS);

    // Inicializando os recursos das receitas
        em forma de sem foro
    sem_init(&sem_fogao, 0, NUM_FOGAO);
    sem_init(&sem_panela, 0, NUM_PANELA);
    sem_init(&sem_forno, 0, NUM_FORNO);
    sem_init(&sem_bancada, 0, NUM_BANCADA);
    sem_init(&sem_grill, 0, NUM_GRILL);
    sem_init(&sem_cozinheiros, 0,
        NUM_COZINHEIROS);

    const char* nomes_recursos_file[] = { "
        Fog o", "Panela", "Bancada" };
    const char* nomes_recursos_lasanha[] = { "
        Fog o", "Panela", "Forno", "Bancada"
        };
    const char* nomes_recursos_risoto[] = { "
        Fog o", "Panela", "Bancada" };
    const char* nomes_recursos_salmao[] = { "
        Grill", "Bancada" };
    const char* nomes_recursos_pizza[] = { "
        Forno", "Bancada" };
    const char* nomes_recursos_costela[] = { "
        Bancada", "Panela","Forno","Bancada" };

    // Criando arrays de ponteiros para os
        sem foros e terminando com NULL
    sem_t* recursos_file[] = { &sem_fogao, &
        sem_panela, &sem_bancada, NULL };
```

```
    sem_t* recursos_lasanha[] = { &sem_fogao, &sem_panela, &sem_forno, &sem_bancada, NULL };
    sem_t* recursos_risoto[] = { &sem_fogao, &sem_panela, &sem_bancada, NULL };
    sem_t* recursos_salmao[] = { &sem_grill, &sem_bancada, NULL };
    sem_t* recursos_pizza[] = { &sem_forno, &sem_bancada, NULL };
    sem_t* recursos_costela[] = { &sem_bancada, &sem_panela, &sem_forno, &sem_bancada, NULL };

    addSemaphoreArray(&receitas, "file", recursos_file, nomes_recursos_file);
    addSemaphoreArray(&receitas, "lasanha", recursos_lasanha, nomes_recursos_lasanha);
    addSemaphoreArray(&receitas, "risoto", recursos_risoto, nomes_recursos_risoto);
    addSemaphoreArray(&receitas, "salmao", recursos_salmao, nomes_recursos_salmao);
    addSemaphoreArray(&receitas, "pizza", recursos_pizza, nomes_recursos_pizza);
    addSemaphoreArray(&receitas, "costela", recursos_costela, nomes_recursos_costela);

    char* menu[NUM_RECEITAS] = {"file", "lasanha", "risoto", "salmao", "pizza", "costela"};

    // Inicializar todos os IDs como dispon veis
    for (int i = 0; i < NUM_COZINHEIROS; i++) {
        id_disponiveis[i] = 1;  // 1 significa dispon vel
    }

    printf("Restaurante abriu!\n");

    int quant_loop = 0;
    while (quant_loop < 10) {
        pthread_mutex_lock(&cozinheiros_lock);
            int quant_cozinheiros;
            sem_getvalue(&sem_cozinheiros,&quant_cozinheiros);

            printf("Quantidade de cozinheiros disponiveis: %d\n", quant_cozinheiros);

            while (quant_cozinheiros == 0) {
                pthread_cond_wait(&cozinheiro_cond, &cozinheiros_lock);
                sem_getvalue(&sem_cozinheiros,&quant_cozinheiros);
            }

            // Procurar um ID dispon vel
            int id_disponivel = -1;
            for (int i = 0; i < NUM_COZINHEIROS; i++) {
                if (id_disponiveis[i] == 1) {
                    id_disponivel = i;
                    enqueue(i + 1);  // Adiciona o ID    fila FIFO
                    id_disponiveis[i] = 0;  // Marcar o ID como em uso
                    break;
                }
            }

            //mostrarCozinheirosDisponiveis();

            // Esperar at  que seja a vez do pr ximo cozinheiro
            if (id_disponivel != -1) {
                id_disponivel = dequeue();  // Remove da fila FIFO
                sem_wait(&sem_cozinheiros);
            }
        pthread_mutex_unlock(&cozinheiros_lock);

        if (id_disponivel != -1) {
            int randomIndex = rand() % NUM_RECEITAS;
            char* randomReceita = menu[randomIndex];

            // Alocar mem ria para a estrutura CozinheiroInfo e atribuir valores
            CozinheiroInfo* info = (CozinheiroInfo*)malloc(sizeof(CozinheiroInfo));
            info->id = id_disponivel;  // Atribuir ID dispon vel
            info->receita = randomReceita;

            pthread_create(&cozinheiro_tid[id_disponivel - 1], NULL, cozinheiro, (void*)info);
        }

        quant_loop++;
    }

    // Esperar todas as threads terminarem
    for (int i = 0; i < NUM_COZINHEIROS; i++) {
        pthread_join(cozinheiro_tid[i], NULL);
    }

    freeDictionary(&receitas);
    sem_destroy(&sem_fogao);
    sem_destroy(&sem_panela);
    sem_destroy(&sem_forno);
    sem_destroy(&sem_bancada);
    sem_destroy(&sem_grill);
    sem_destroy(&sem_cozinheiros);
    pthread_mutex_destroy(&print_lock);
    pthread_mutex_destroy(&recurso_lock);
    pthread_mutex_destroy(&cozinheiros_lock);
    pthread_cond_destroy(&cozinheiro_cond);

    return 0;
}
```

Inicialização do Dicionário e Semáforos

A função começa inicializando o dicionário que armazenará as receitas e os semáforos associados aos recursos necessários para cada uma delas. Em seguida, todos os semáforos que representam os recursos da cozinha são inicializados, como o fogão, a panela, o forno, a bancada e o grill. Esses semáforos são configurados para permitir apenas um número limitado de acessos simultâneos, de acordo com a quantidade disponível de cada recurso.

Definição das Receitas e Seus Recursos

O próximo passo consiste em definir os recursos necessários para cada receita. Para isso, são criados arrays de semáforos e strings que representam os recursos necessários para cada prato (como `"Fogão"`, `"Panela"`, `"Forno"`, etc.). Esses arrays são, então, adicionados ao dicionário de receitas utilizando a função `addSemaphoreArray`. Cada receita é mapeada a um conjunto específico de semáforos e nomes de recursos.

Disponibilidade Inicial dos Cozinheiros

O array `id_disponiveis` é inicializado, marcando todos os IDs de cozinheiros como disponíveis. Essa estrutura é utilizada para rastrear quais cozinheiros estão livres para iniciar novas tarefas.

Loop Principal de Operação

A função entra em um loop que simula a operação contínua do restaurante. O loop é configurado para iterar 10 vezes, simulando a preparação de 10 pratos. Durante cada iteração, o programa verifica quantos cozinheiros estão disponíveis utilizando o semáforo `sem_cozinheiros`. Se não houver cozinheiros disponíveis, o programa espera até que um cozinheiro fique livre.

Uma vez que um cozinheiro está disponível, ele é marcado como ocupado, e seu ID é enfileirado para garantir que a ordem de atendimento seja mantida. Em seguida, o ID é retirado da fila para iniciar a preparação da receita.

Atribuição de Receitas aos Cozinheiros

Dentro do loop principal, uma receita é selecionada aleatoriamente a partir do menu disponível. Uma estrutura `CozinheiroInfo` é criada e preenchida com o ID do cozinheiro e a receita atribuída. Uma nova thread é criada para cada cozinheiro, onde a função `cozinheiro` é chamada para gerenciar a preparação da receita atribuída.

Finalização e Limpeza

Após o loop principal, o programa aguarda a finalização de todas as threads dos cozinheiros utilizando `pthread_join`. Isso garante que todos os cozinheiros concluam suas tarefas antes de o programa terminar. Finalmente, os recursos alocados são liberados, os semáforos são destruídos, e os mutexes e variáveis de condição são finalizados para garantir que todos os recursos do sistema sejam corretamente desalocados.

3.4.2 Função `cozinheiros`

A função `cozinheiros` é uma função de thread que simula o trabalho de um cozinheiro preparando uma receita específica em um ambiente de cozinha. O algoritmo gerencia a alocação e liberação de recursos (representados por semáforos) necessários para a preparação das receitas, garantindo que o acesso aos recursos seja coordenado entre múltiplos cozinheiros. Assim gerenciado a concorrencia e dando feedbacks do estado atual do processo na simulação.

Inicialização da Estrutura CozinheiroInfo

A função começa recebendo um ponteiro `arg` que é convertido para uma estrutura `CozinheiroInfo`. Essa estrutura contém o ID do cozinheiro e a receita que ele irá preparar. Com base na receita, a função obtém o conjunto de semáforos associados aos recursos necessários para a preparação da receita usando a função `getSemaphoreArray`.

Anúncio do Início da Preparação

O cozinheiro imprime uma mensagem indicando que ele está começando a preparar a receita. Em seguida, se o conjunto de semáforos (`SemaphoreArray`) não for nulo, o cozinheiro entra em um loop para tentar adquirir cada um dos recursos necessários.

Aquisição dos Recursos

Para cada recurso necessário:

- **Checagem de Disponibilidade:** O cozinheiro verifica se o recurso está disponível, usando `sem_getvalue` para obter o valor atual do semáforo. Se o recurso estiver indisponível (valor do semáforo é zero), uma mensagem é impressa e o cozinheiro espera por um tempo aleatório antes de tentar novamente.

- **Espera e Uso do Recurso:** Se o recurso estiver disponível, o cozinheiro faz uma chamada a `sem_wait` para adquirir o recurso, imprimindo uma mensagem indicando que ele está usando o recurso. O cozinheiro então simula o tempo necessário para usar o recurso, que é definido por um tempo aleatório entre 5 e 15 segundos (usando a função `sleep`).

- **Liberação do Recurso:** Após o uso do recurso, o cozinheiro libera o recurso chamando `sem_post`, permitindo que outros cozinheiros possam utilizá-lo. Uma mensagem é impressa indicando que o recurso foi liberado.

Conclusão da Receita

Após adquirir e utilizar todos os recursos necessários, o cozinheiro imprime uma mensagem informando que a receita foi concluída com sucesso.

Atualização do Status do Cozinheiro

Depois de finalizar a receita, o cozinheiro atualiza seu status para indicar que está disponível novamente:

- **Liberação do Cozinheiro:** O semáforo `sem_cozinheiros` é incrementado, sinalizando que um cozinheiro está livre.

- **Atualização do ID Disponível:** O ID do cozinheiro é marcado como disponível no array `id_disponiveis`.

- **Sinalização para Outras Threads:** Uma condição (`pthread_cond_signal`) é sinalizada para alertar outras threads de que um cozinheiro está disponível.

Liberação de Recursos e Saída

Finalmente, a memória alocada para a estrutura `CozinheiroInfo` é liberada, e a thread do cozinheiro é finalizada com `pthread_exit`.

4 CONCLUSÃO

O desenvolvimento do sistema de gerenciamento de recursos para cozinheiros em um ambiente de cozinha, utilizando programação concorrente, demonstrou a eficiência e a importância de técnicas como threads, semáforos, e variáveis de condição na coordenação de tarefas simultâneas. O algoritmo foi projetado para garantir que múltiplos cozinheiros pudessem trabalhar de maneira coordenada e eficiente, sem causar contenção ou deadlocks nos recursos compartilhados.

A abordagem adotada para gerenciar os recursos da cozinha, através do uso de semáforos, permitiu controlar o acesso concorrente a equipamentos como fogões, panelas, fornos, bancadas e grills. Cada recurso foi tratado como um semáforo com um valor inicial que representa sua quantidade disponível, garantindo assim que os cozinheiros possam adquirir e liberar recursos de forma ordenada e controlada.

A estruturação do código em torno de funções que gerenciam tanto a aquisição quanto a liberação de recursos, aliada à criação de threads individuais para cada cozinheiro, demonstrou uma solução escalável e flexível para o problema. O uso de `pthread_mutex_t` e `pthread_cond_t` garantiu que o sistema fosse capaz de lidar com a disponibilidade de cozinheiros, sem provocar ineficiências ou condições de corrida.

Além disso, a implementação de um sistema de filas para gerenciar os IDs dos cozinheiros assegurou a manutenção da ordem de atendimento, prevenindo que um cozinheiro ficasse sobrecarregado ou que as threads fossem executadas de maneira não organizada.

Ao longo do desenvolvimento e da execução dos testes, verificou-se que o sistema é robusto e capaz de lidar com as complexidades inerentes ao ambiente de cozinha simulado. A capacidade de simular múltiplas iterações de preparo de pratos, com diferentes requisitos de recursos, validou a eficácia do algoritmo proposto.

Em suma, o trabalho realizado neste projeto não só atendeu aos requisitos estabelecidos, como também proporcionou uma visão prática e aplicada das técnicas de programação concorrente, evidenciando o valor dessas ferramentas no desenvolvimento de sistemas complexos e altamente paralelizados.

REFERÊNCIAS

[1] Pthreads Programming. *Carnegie Mellon University, School of Computer Science*. Disponível em: https://www.cs.cmu.edu/afs/cs/academic/class/15492-f07/www/pthreads.html. Acesso em: 14 ago. 2024.