

INE5645 — Programação Paralela e Distribuída

Trabalho I: Programação Paralela

Pedro Santi Binotto (20200634)
Gabriel Lemos da Silva (18200628)



Introdução do tema

Modelo de sistema de cozinha de restaurante

Escopo

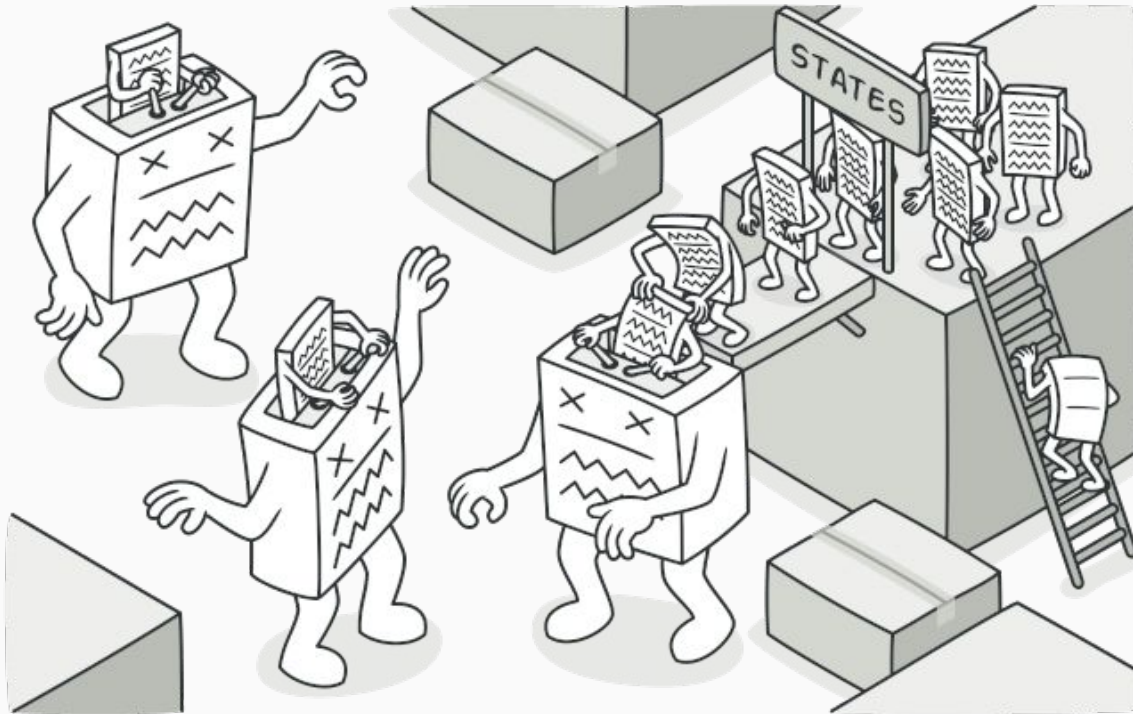
O sistema desenvolvido é uma representação de uma cozinha de um restaurante, implementada utilizando *threads* concorrentes, que representam os diferentes componentes abrangidos pela modelagem:

- Tempo de trabalho dos cozinheiros;
- Chegada de clientes e tempo de aguardo pelo pedido;
- Tempo de operação e fechamento do restaurante;
- Priorização e paralelização do preparo de pedidos;
- Acesso exclusivo à recursos limitados (equipamentos de cozinha; fogões, fornos, etc.).

Especificação técnica

Para simular os processos concorrentes que acontecem em uma cozinha comercial em um restaurante, as seguintes estruturas e padrões foram empregados:

- *Threads;*
- *Publishers/Subscriber (Observer Pattern);*
- *Barrier Pattern;*
- *Synchronized-Access Queues;*
- *Counting Semaphores.*



Detalhamento sobre implementação

Cada um dos seguintes elementos é representado por uma *thread* diferente:

- Cozinheiros (uma *thread* por cozinheiro);
- Clientes (uma *thread* por cliente);
- *Timer* do restaurante (*thread* paralela em background controla o tempo que o restaurante permanece aberto para novos pedidos)

Detalhamento sobre implementação

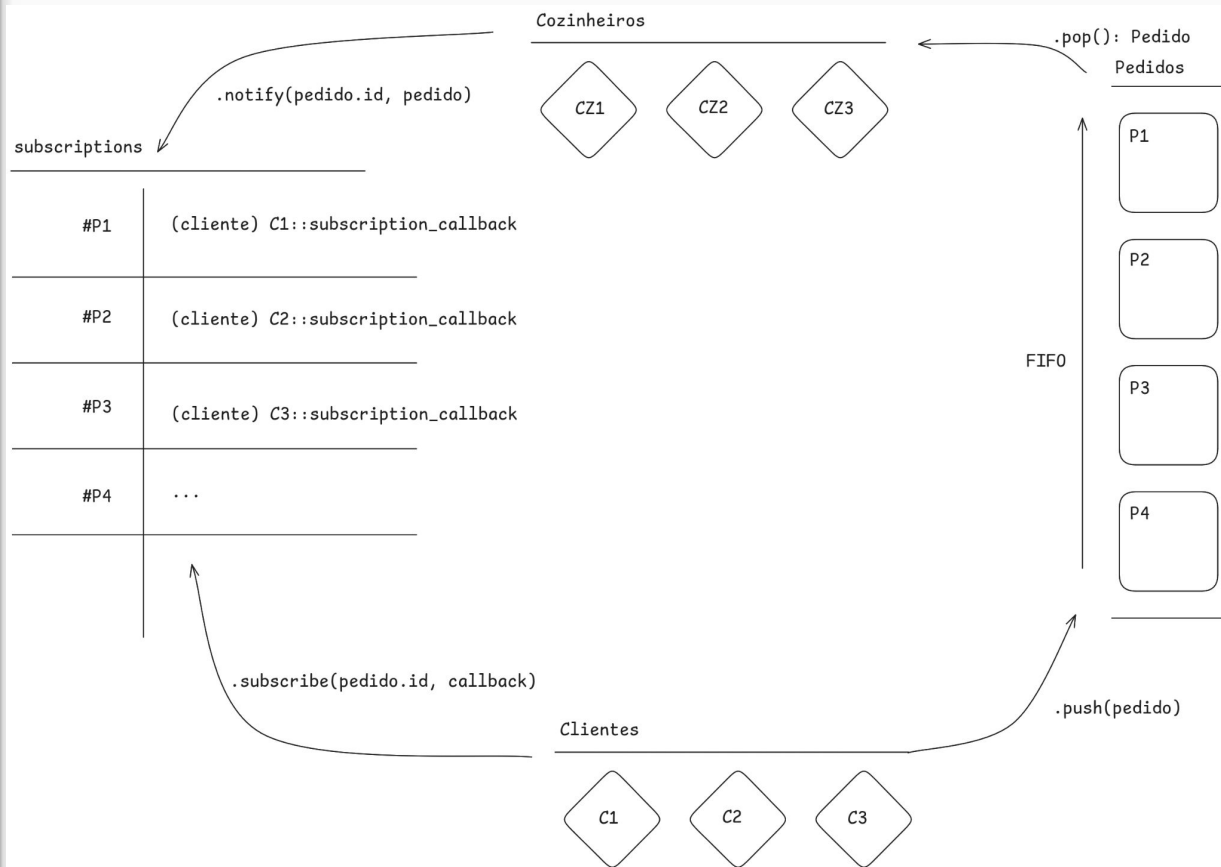
Estas *threads* são mantidas em sincronia utilizando os seguintes mecanismos:

- Os pedidos realizados por cada *thread* cliente são mantidos em uma implementação *thread-safe* de uma fila FIFO;
- Cada *thread* cozinheiro lê o pedido no topo da fila e inicia o preparo;
- O cozinheiro pode acessar os recursos limitados (fogão, forno) exclusivamente, mediados por instâncias de *counting semaphores*;
- Quando o preparo do pedido está finalizado, o cliente é notificado pelo cozinheiro através de um mecanismo de gerenciamento de observadores;
- Uma vez que a cozinha está *fechada*, apenas os pedidos que já foram realizados serão processados.

Fluxo da aplicação

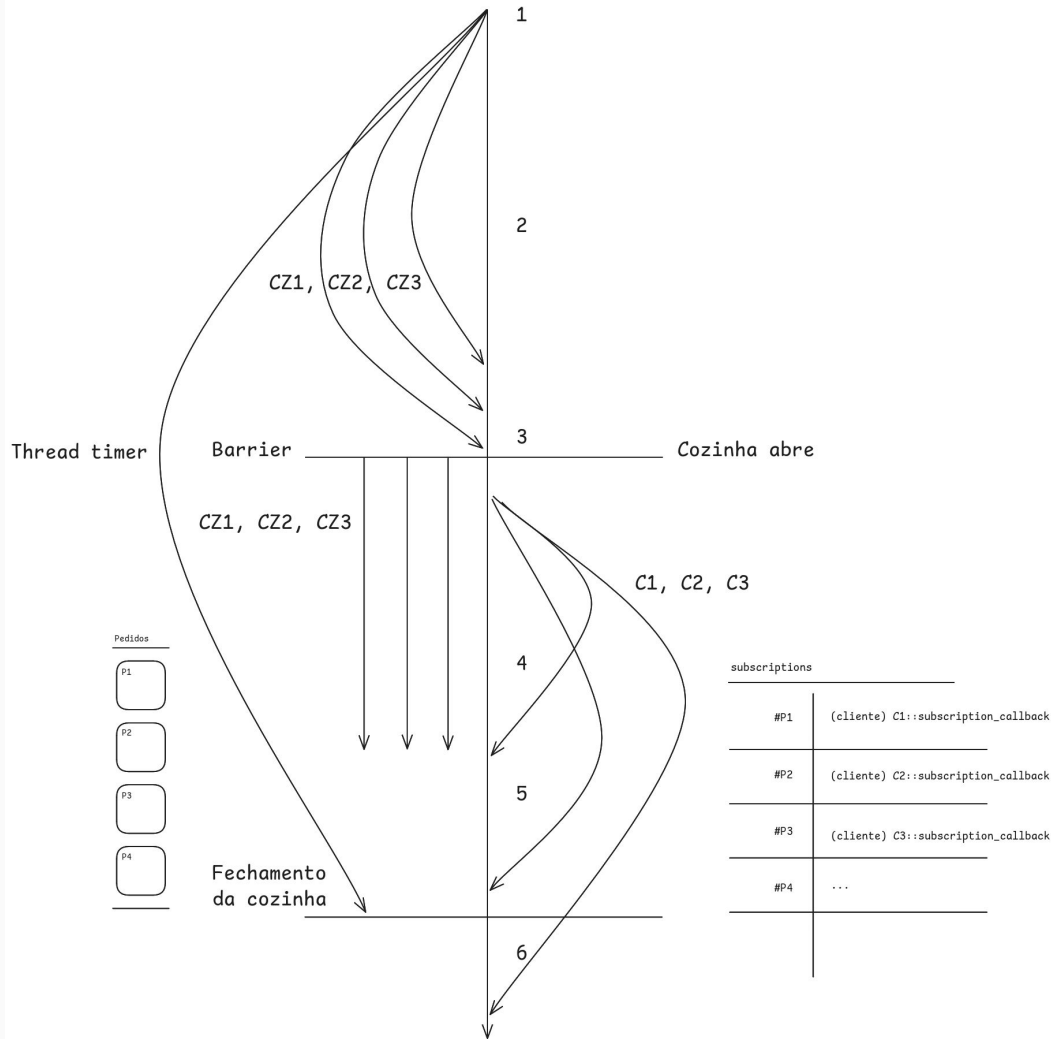
Evidenciado:

- Mecanismo de *Observers* ("subscriptions");
- *Threads* cozinheiros (CZ1, CZ2...), clientes (C1, C2...);
- Fila de pedidos, com operações push e pop;



Fluxo da aplicação

- Início da aplicação, cozinha fechada (1);
- Cozinheiros chegam para trabalhar, um por vez (2);
- Uma vez que todos os cozinheiros estão na cozinha, restaurante abre para pedidos (3);
- Clientes, um por vez, chegam no restaurante (4):
 - Realizam e aguardam pedido, caso o restaurante está aberto para pedidos;
 - Desistem e vão embora, caso contrário;
- À medida que pedidos ficam prontos, clientes recebem e vão embora (5);
- Restaurante fecha para pedidos e, uma vez que todos na fila forem entregues, cozinheiros finalizam turno (6).



Implementação dos mecanismos

> barrier

- Implementação que permite que a instância seja reutilizada após notificação (via variável `generation`);
- Na chamada de uma *thread* sobre `wait()`, contador (`count`, inicializado no valor máximo) é decrementado e, ao atingir 0, que significa que as *threads* que estão esperando podem ser liberadas, via condição de variável `cv`;

```
struct barrier {  
private:  
    std::mutex mutex;  
    std::condition_variable cv;  
    const size_t threshold;  
    size_t count;  
    size_t generation = 0;  
  
public:  
    void wait() {  
        std::unique_lock<std::mutex> lock(mutex);  
        size_t current_gen = generation;  
  
        if (--count == 0) {  
            generation++;  
            count = threshold;  
            cv.notify_all();  
        } else {  
            cv.wait(  
                lock,  
                [this, current_gen] {  
                    return current_gen != generation;  
                });  
        }  
    }  
};
```

Implementação dos mecanismos

> observer_subject

- Implementação genérica permite que seja utilizado para quaisquer tipos de dados (T utilizado como “chave”, enquanto U representa o dado transitado);
- Mantém um mapa de todas as chaves (eventos) e, para cada, um vetor de seus observadores;
- Operações feitas *thread-safe* através do uso de proteção por escopo (`std::lock_guard`, da STL);

```
template <typename T, typename U> struct observer_subject {
public:
    void subscribe(T event_id, std::function<void(const U &)>
callback) {
        std::lock_guard<std::mutex> lock(m);
        subscriptions[event_id].push_back(std::move(callback));
    }

    void notify(T event_id, const U &event) {
        std::lock_guard<std::mutex> lock(m);
        std::vector<std::function<void(const U &)>> fs =
subscriptions[event_id];
        for (std::function<void(const U &)> callback : fs) {
            callback(event);
        }
    }

private:
    std::unordered_map<T, std::vector<std::function<void(const
U &)>>> subscriptions;
    std::mutex m;
};
```

Implementação dos mecanismos

> counting_semaphore

- Sincronização de acesso protegido por escopo (`std::unique_lock`) e mecânica de “wait” implementada utilizando *condition variables*;

```
struct counting_semaphore {  
private:  
    std::mutex mutex;  
    std::condition_variable cv;  
    int count;  
  
public:  
    void acquire() {  
        std::unique_lock<std::mutex> lock(mutex);  
        cv.wait(lock, [this] { return count > 0; });  
        count--;  
    }  
    void release() {  
        {  
            std::lock_guard<std::mutex> lock(mutex);  
            count++;  
        }  
        cv.notify_one();  
    }  
};
```

Implementação dos mecanismos

> synchronizing_queue

- Implementação *thread-safe* de uma fila FIFO;
- Utiliza a implementação padrão de fila (`std::queue`) da STL, apenas oferecendo uma camada de sincronização utilizando de proteção por escopo (`std::lock_guard`) e *condition variables*;

OBS.: Alguns membros da classe foram omitidos do slide para facilitar legibilidade: `size()`, `empty()`, `peek()`

```
template <typename T> struct synchronizing_queue {
private:
    std::queue<T> elements;
    std::mutex mutex;
    std::condition_variable cv;

public:
    void push(const T &element) {
        {
            std::lock_guard<std::mutex> lock(mutex);
            elements.push(element);
        }
        cv.notify_one();
    }

    std::optional<T> pop() {
        std::unique_lock<std::mutex> lock(mutex);
        cv.wait(lock, [this] { return !elements.empty(); });

        if (elements.empty())
            return std::nullopt;

        T element = elements.front();
        elements.pop();
        return element;
    }
};
```

Execução e parametrização

```
pedro@bridge ~$ project (U) main 1
total 116
drwxrwxr-x 3 pedro pedro 4096 May 18 20:29 .
drwxrwxr-x 4 pedro pedro 4096 May 7 21:07 ..
-rw-r--r-- 1 pedro pedro 6031 May 7 20:25 .clang-format
-rw-r--r-- 1 pedro pedro 304 May 7 20:25 .gitignore
-rw-r--r-- 1 pedro pedro 83100 May 7 20:21 LINES645_trabalho1.pdf
-rw-r--r-- 1 pedro pedro 571 May 15 13:33 Makefile
-rw-r--r-- 1 pedro pedro 3635 May 18 20:28 README.md
drwxrwxr-x 2 pedro pedro 4096 May 18 20:03 src
pedro@bridge ~$ project (U) main make

Choose a make command to run

build      compile project to binary
clean      clean up object and binary files
run        build and run project;
run $(ARGS) run with command line args via 'make run ARGS="<arg1, arg2 ...>"'

pedro@bridge ~$ project (U) main make run
g++ -std=c++17 -Wall -Wextra -g -pedantic -lncurses -pthread -o src/main.cpp -o obj/main.o
src/main.cpp: In lambda function:
src/main.cpp:122:63: warning: unused parameter ' ' [-Wunused-parameter]
   122 |     subscriptions.subscribe(client_order.id, [ &](const order &o) {
       |                                                    ~~~~~
g++ -std=c++17 -Wall -Wextra -g -pedantic -lncurses -pthread obj/main.o -o bin/restaurant -lncurses -pthread
./bin/restaurant
[NUM_COZINHEIROS, NUM_CLIENTES, NUM_FOGÕES, NUM_FORNOS, TEMPO_FECHAMENTO_RESTAURANTE_SEGUNDOS] não informados; usando parâmetros padrão:
[5, 30, 2, 2, 15]

*****

Bienvenue, monsieur ou madame, seja calorosamente bem-vindo ao
Chez L'Exagéré, o restaurante francês onde a elegância é obrigatória e o
cardápio, praticamente ilegal.

Aqui, cada prato é uma obra de arte - e, convenhamos, o preço também.

Nosso menu é elaborado diariamente - ou, como preferimos dizer,
emocionalmente - pelo renomado chef Nathalie Fontaine, que acredita firmemente
que manteiga é um estado de espírito.

*****

Iniciando simulação.
[49363282] Restaurante abrirá em breve
[49367890] Cozinheiro Juliana Souza (id: 2) chegou para trabalhar
[50187802] Cozinheiro Gabriel Binotto (id: 5) chegou para trabalhar
20:28:00 *** [Makefile:17: run] Intermittent
```

Argumentos de linha de comando

O programa disponibiliza opções de parametrização para as seguintes variáveis:

> `TEMPO_FECHAMENTO_RESTAURANTE_SEGUNDOS`

Permite configurar o tempo de operação no qual a cozinha receberá pedidos;

> `NUM_COZINHEIROS`

Permite configurar o número de threads que representarão cozinheiros;

> `NUM_CLIENTES`

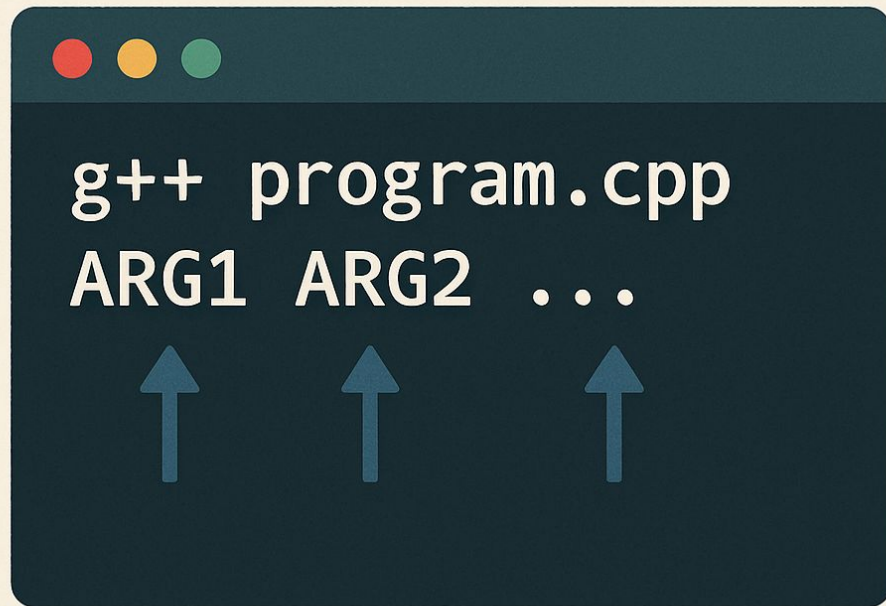
Permite configurar o número de threads que representarão clientes

> `NUM_FOGÕES`

Número de fogões disponibilizados na cozinha;

> `NUM_FORNOS`

Número de fornos disponibilizados na cozinha;



Command Line Arguments **C++**

Relatório de dados

Ao final da execução, o programa exibirá um relatório de dados da execução da simulação.

Estatísticas finais:

Tempo de operação da cozinha (segundos): 15

Total de pedidos realizados por clientes: 18

Total de pedidos negados (tentativas após fechamento): 12

Total de pedidos preparados por cada cozinheiro:

Cozinheiro Mateus Silva (id: 5):

Total de pedidos realizados: 4

ID Pedido: 4

Tempo de preparo (segundos): 4

ID Pedido: 10

...

Simulação finalizada

Obrigado!



Por: Gabriel Lemos da Silva [18200628] e Pedro Santi Binotto [20200634]