UNIVERSIDADE FEDERAL DE SANTA CATARINA
CAMPUS FLORIANÓPOLIS
INE - DEPARTAMENTO DE INFORMÁTICA E ESTATÍSTICA
PROGRAMAÇÃO CONCORRENTE

RELATÓRIO TRABALHO I DE PROGRAMAÇÃO CONCORRENTE

Rodadas Grátis em um Bar

Erik Orsolin de Paula (22102195)

Rafael Bitencourt (22203673)

Pedro Augusto da Fontoura (22215098)

FLORIANÓPOLIS
2023

1 - Introdução

Neste trabalho, propomos uma simulação de um bar que oferece um número específico de rodadas grátis para seus clientes. A dinâmica de atendimento é conduzida por um conjunto de garçons, onde cada um tem a capacidade de atender um número limitado de clientes por vez. Esta capacidade é homogênea entre todos os garçons. O cenário se torna intrigante pois cada cliente, após ser atendido, engaja em conversas com amigos por um tempo aleatório antes de fazer um novo pedido e, depois do pedido ser feito, gasta um tempo aleatório para consumi-lo. Além disso, há uma regra adicional: um garçom só se desloca para buscar os pedidos na copa quando todos os clientes que ele pode atender já fizeram seus pedidos, ou seja, a capacidade máxima Gn do garçom foi atingida.

O desafio apresentado por este problema envolve conceitos de programação concorrente como threads, semáforos, mutexes e sincronização. A simulação busca entender como diferentes variáveis - como o número de garçons, a capacidade de atendimento e os tempos aleatórios de espera dos clientes - influenciam o processo de atendimento durante as rodadas grátis. O objetivo é entender e otimizar o fluxo de atendimento, garantindo que o maior número de clientes seja atendido de forma eficiente ao longo das rodadas estipuladas.

Ao longo deste relatório, detalharemos as abordagens adotadas, as técnicas de programação concorrente utilizadas e os resultados obtidos.

2 - Código desenvolvido

main.cpp - inicialização de variáveis globais e estruturas

```c
// N -> número de clientes
// G -> número de garçons
// Gn -> quantidade de clientes que cada Garçom  atende
// R -> número de rodadas
// max_conversa -> tempo máximo de conversa
// max_consumo -> tempo máximo de consumo


int N, G, Gn, R, max_conversa, max_consumo, rodada = 0, index_clientes = 0, index_garcons = 0, pedidos_entregues = 0;
sem_t sem_pedidos, sem_controle, mutex_clientes, mutex_garcons, mutex_rodada, garcons_prontos;

typedef struct {
    int id_cliente;
    int* lista_id_clientes;
    sem_t** lista_sem_clientes;
} info_clientes;

typedef struct {
    int id_garcom;
    int* lista_id_clientes;
    sem_t** lista_sem_clientes;
} info_garcons;
```

Nessa parte inicial do código são declaradas as variáveis globais, semáforos e estruturas que serão utilizadas no código para o controle de rodadas, atendimentos e lógica de sincronização entre clientes, garçons e troca de rodadas.

main.cpp - thread cliente

```c
// função das threads cliente
void* cliente(void* info_cliente) {

    info_clientes* cliente = (info_clientes*) info_cliente; // cast para o tipo info_clientes

    sem_t sem_cliente;  // semáforo usado para o cliente receber seu pedido
    sem_init(&sem_cliente, 0, 0);

    // loop de comportamento do cliente
    while (rodada < R) {

        // conversando
        if (max_conversa_seg) sleep(rand() % max_conversa_seg);
        if (max_conversa_micro) usleep((rand() % 1000) * (max_conversa_micro / 1000));

        // fazendo pedido
        sem_wait(&sem_controle); // semáforo que garante que G*Gn clientes façam seus pedidos
        if (rodada == R) break;

        sem_wait(&mutex_clientes); // mutex para proteger o uso da variável index_clientes

        printf("CLIENTE %02d PEDIU\n", cliente->id_cliente);
        cliente->lista_id_clientes[(index_clientes)%N] = cliente->id_cliente; // adicionando o id do cliente na lista de ids
        cliente->lista_sem_clientes[(index_clientes++)%N] = &sem_cliente;     // adicionando o semáforo do cliente na lista de semáforos
        sem_post(&sem_pedidos); // sinalizando para os garçons que um pedido foi feito

        sem_post(&mutex_clientes);

        // esperando pedido
        sem_wait(&sem_cliente);

        // comendo
        if (max_consumo_seg) sleep(rand() % max_consumo_seg);
        if (max_consumo_micro) usleep((rand() % 1000) * (max_consumo_micro / 1000));
    }

    pthread_exit(NULL);
}
```

Função `cliente`

A função `cliente` representa o comportamento de um cliente individual em um bar. Esse cliente pode realizar algumas ações como conversar, fazer um pedido e consumir o que foi pedido. A função usa conceitos de programação concorrente com o uso de threads e semáforos para garantir que os clientes se comportem de forma sincronizada e sem conflitos.

1. **Inicialização**:
   Primeiramente, a função recebe um argumento, que é um ponteiro para uma estrutura chamada `info_clientes`. Essa estrutura contém informações como o ID do cliente, uma lista de IDs de clientes que fizeram pedidos e uma lista de semáforos associados a esses clientes. O argumento é então convertido de volta para o tipo `info_clientes` para que possamos acessar suas propriedades.
   Em seguida, um semáforo chamado `sem_cliente` é inicializado. Este semáforo é utilizado pelo cliente para aguardar a entrega de seu pedido.
2. **Comportamento do Cliente**:
   O cliente tem um loop de comportamento que se repete enquanto o número de rodadas for menor que `R` (o total de rodadas definido pelo programa).
   Dentro desse loop:
   - **Conversando**:
     O cliente conversa por um tempo aleatório. O tempo máximo de conversa pode incluir segundos (com a função `sleep()`) e microssegundos (com a função `usleep()`), ambos determinados pelas variáveis `max_conversa_seg` e `max_conversa_micro`.
   - **Fazendo Pedido**:
     Antes de fazer um pedido, o cliente aguarda o semáforo `sem_controle`, que garante que apenas um número específico de clientes (definido pelo produto de garçons e a quantidade de clientes que cada garçom atende) faça seus pedidos simultaneamente.
     Se o número de rodadas atingir `R`, o cliente sai do loop. Caso contrário, ele bloqueia o acesso à variável `index_clientes` usando o semáforo `mutex_clientes`, garantindo que apenas um cliente acesse essa variável por vez.
     O cliente então registra seu ID e seu semáforo pessoal nas listas correspondentes. Depois de fazer isso, ele sinaliza que fez um pedido usando o semáforo `sem_pedidos` e, em seguida, libera o semáforo `mutex_clientes`.
   - **Esperando Pedido**:
     O cliente aguarda seu pedido usando seu semáforo pessoal

`sem_cliente`. Quando seu pedido for entregue por um garçom, esse semáforo será liberado.

- **Comendo**:
  Assim como na fase de conversa, o cliente consome o pedido por um tempo aleatório definido pelas variáveis `max_consumo_seg` e `max_consumo_micro`.

3. **Finalização**:
   Uma vez que o cliente tenha concluído todas as rodadas, ele encerra sua thread e sai do bar.

main.cpp - thread garçom

```
// função das threads garçom
void* garcom(void* info_garcom) {

    info_garcons* garcom = (info_garcons*) info_garcom; // cast para o tipo info_garcons

    int ids_anotados[Gn]; // vetor para a fila interna de ids de cada garçom
    sem_t* sem_anotados[Gn]; // vetor para a fila interna de semáforos de cada garçom

    // loop de comportamento do garçom
    while (rodada < R) {

        sem_wait(&garcons_liberados); // semáforo para impedir o deadlock em que N < (Gn - 1) * G

        // enchendo a bandeja
        for (int i = 0; i < Gn; i++) {

            // esperando ser feito um pedido
            sem_wait(&sem_pedidos);

            // anotando pedido
            sem_wait(&mutex_garcons); // mutex para proteger o uso da variável index_garcons

            printf("GARÇOM  %02d ANOTOU   O PEDIDO DO CLIENTE %02d (%d/%d)\n", garcom->id_garcom, garcom->lista_id_clientes[(index_garcons)%N], i + 1, Gn);
            ids_anotados[i] = garcom->lista_id_clientes[(index_garcons)%N]; // adicionando o id do cliente na fila interna
            sem_anotados[i] = garcom->lista_sem_clientes[(index_garcons++)%N]; // adicionando o semáforo do cliente na fila interna

            sem_post(&mutex_garcons);

        }

        // entregando os pedidos
        for (int i = 0; i < Gn; i++) {
            printf("GARÇOM  %02d ENTREGOU O PEDIDO DO CLIENTE %02d\n", garcom->id_garcom, ids_anotados[i]);
            sem_post(sem_anotados[i]); // sinalizando para o cliente que seu pedido foi entregue
        }
```

Função `garcom`

A função `garcom` representa o comportamento de um garçom individual em um bar. Este garçom tem a responsabilidade de receber pedidos dos clientes e entregá-los de forma eficiente. Assim como a função `cliente`, esta função também faz uso intenso de semáforos para sincronizar as ações e evitar conflitos.

1. **Inicialização**:
   A função começa recebendo um argumento que representa as informações do garçom. Esta estrutura contém o ID do garçom, uma lista de IDs de clientes que fizeram pedidos, seus semáforos correspondentes e outras informações relevantes.
2. **Comportamento do Garçom**:
   O garçom entra em um loop contínuo de trabalho, esperando por pedidos e, em seguida, entregando-os. Para isso:

- **Aguardando Pedidos**:
  O garçom aguarda que um cliente faça um pedido através do semáforo `sem_pedidos`. Quando um cliente faz um pedido, ele libera este semáforo, permitindo que o garçom saiba que há um pedido pendente.
- **Recebendo o Pedido**:
  Uma vez que o semáforo `sem_pedidos` é liberado, o garçom bloqueia o acesso à variável `index_garcons` usando o semáforo `mutex_garcons`. Isso garante que apenas um garçom possa pegar um pedido de cada vez, evitando possíveis conflitos.
  O garçom então acessa as informações do cliente que fez o pedido. Ele obtém o ID do cliente e o semáforo associado a esse cliente da lista.
  Uma vez que o garçom tenha as informações do cliente, ele incrementa a variável `index_garcons` (que funciona como um índice para rastrear os pedidos pendentes) e, em seguida, libera o semáforo `mutex_garcons`, permitindo que outros garçons ou clientes acessem a variável.
- **Entregando o Pedido**:
  Uma vez que o pedido esteja pronto, o garçom usa o semáforo associado ao cliente (que foi obtido na etapa de "Recebendo o Pedido") para sinalizar que o pedido foi entregue. Isso é feito liberando o semáforo do cliente, permitindo que o cliente prossiga para consumir o pedido.

main.cpp - thread garçom (lógica de fim de rodada)

```
    // comportamento do garçom que já atendeu seus clientes
    sem_wait(&mutex_rodada); // mutex para proteger o uso da variável rodada

    sem_post(&garcons_liberados); // sinalizando que mais um garçom pode atender sem causar o deadlock em que N < (Gn - 1) * G
    pedidos_entregues += Gn; // incrementando o número de pedidos entregues


    // caso nem todos os garçons tenham terminado de atender
    if (!(pedidos_entregues == G * Gn)) {

        sem_post(&mutex_rodada); // liberando o mutex para que mais garçons verifiquem o fim de rodada
        sem_wait(&garcons_prontos); // esperando o ultimo garçom incrementar a rodada

    } else {

        printf("\nFim da rodada %d\n\n", ++rodada); // incrementando a rodada e imprimindo o fim da rodada

        pedidos_entregues = 0; // resetando o número de pedidos entregues
        for(int i = 0; i < G - 1; i++) sem_post(&garcons_prontos); // liberando os garçons para a próxima rodada

        if (rodada < R) for (int i = 0; i < G * Gn; i++) sem_post(&sem_controle); // liberando G * Gn clientes para fazerem seus pedidos
        else for (int i = 0; i < N; i++) sem_post(&sem_controle); // liberando todos os clientes para sairem do bar

        sem_post(&mutex_rodada);
    }
}
```

1. **Proteção da Variável de Controle da Rodada**: O `mutex_rodada;` é um semáforo do tipo mutex usado para proteger o acesso à variável `rodada`. Isso garante que apenas um garçom por vez possa verificar ou modificar essa variável.
2. **Liberação de Garçons e Contabilização de Pedidos**: O garçom sinaliza que concluiu sua rodada com `sem_post(&garcons_liberados);`, liberando espaço para outro garçom. O número de pedidos entregues é incrementado por

`pedidos_entregues += Gn;`, onde `Gn` representa o número de clientes que um garçom atende em uma rodada.

3. **Verificação da Conclusão da Rodada**: A condição `if (!(pedidos_entregues == G * Gn))` verifica se todos os garçons já entregaram seus pedidos. Se não for o final da rodada, o garçom libera o mutex com `sem_post(&mutex_rodada);` e aguarda o último garçom sinalizar o fim da rodada com `sem_wait(&garcons_prontos);`.
4. **Final da Rodada**: Se todos os garçons concluíram suas entregas (caso contrário do `if` anterior), o código incrementa a variável `rodada` e exibe uma mensagem indicando o fim dessa rodada. O contador `pedidos_entregues` é redefinido para zero. A lógica subsequente libera os garçons para a próxima rodada e, com base na rodada atual, decide quantos clientes podem fazer seus pedidos na rodada seguinte ou, se for a última rodada, quantos clientes podem deixar o bar.
5. **Liberação do Mutex**: Por fim, o mutex `mutex_rodada` é liberado com `sem_post(&mutex_rodada);`, permitindo que outros garçons verifiquem ou modifiquem a variável `rodada` conforme necessário.

main.cpp - main

```
int main(int argc, char* argv[]){

    // tratamento de exceções
    if (argc != 7 || atoi(argv[1]) <= 0 || atoi(argv[2]) <= 0 || atoi(argv[3]) <= 0 || atoi(argv[4]) <= 0 || atoi(argv[5]) <= 0 || atoi(argv[6]) <= 0) {
        printf("Uso: %s <clientes> <garcons> <clientes/garcon> <rodadas> <max.conversa> <max.consumo>\n", argv[0]);
        exit(1);
    } else if (atoi(argv[1]) < atoi(argv[3])) {
        printf("Erro: O número de clientes deve ser maior que a capacidade de cada garçom. Caso contrário ocorrerá um deadlock inevitável, quando seguidas as regras do enunciado.\n");
        exit(1);
    }

    // inicializando as variáveis com os argumentos
    N = atoi(argv[1]);
    G = atoi(argv[2]);
    Gn = atoi(argv[3]);
    R = atoi(argv[4]);
    max_conversa_mili = atoi(argv[5]);
    max_consumo_mili = atoi(argv[6]);

    // converte de milésimos para segundos e microsegundos
    max_conversa_seg = max_conversa_mili / 1000;
    max_conversa_micro = (max_conversa_mili % 1000) * 1000;
    max_consumo_seg = max_consumo_mili / 1000;
    max_consumo_micro = (max_consumo_mili % 1000) * 1000;

    // tratamento do caso em que N < (Gn - 1) * G
    Gl = G;
    while (N <= (Gn - 1) * Gl) Gl--;

    // declarando vetores
    int* lista_id_clientes = (int*) malloc(N * sizeof(int)); // alocando dinâmicamente a lista de ids dos clientes que pedirão
    sem_t** lista_sem_clientes = (sem_t**) malloc(N * sizeof(sem_t*)); // alocando dinâmicamente a lista de semáforos dos clientes que pedirão
    info_clientes info_cliente[N]; // vetor de structs para passar os parâmetros para as threads clientes
    info_garcons info_garcom[G]; // vetor de structs para passar os parâmetros para as threads garçons
```

1. **Tratamento de Exceções**:
    - `if (argc != 7 || ...)` verifica se o número de argumentos é diferente de 7 e se algum deles não é positivo. Caso verdadeiro, imprime uma mensagem de erro informando o usuário sobre o uso correto do programa e encerra a execução.
    - O segundo `if` garante que o número de clientes seja maior que a capacidade de atendimento de um garçom, caso contrário avisa sobre um deadlock inevitável se seguido as restrições do enunciado.
2. **Inicialização de Variáveis com Argumentos**:

- O programa lê os argumentos passados pela linha de comando e os atribui às variáveis correspondentes, utilizando a função `atoi()` para converter as strings para inteiros.

3. **Conversão de Milésimos para Segundos e Microsegundos**:
   - Convertendo o tempo máximo de conversa e consumo de milésimos para uma combinação de segundos e microsegundos. Isso é feito pelo fato da função sleep aceitar apenas segundos e função usleep aceitar apenas microsegundos.
4. **Tratamento Especial de Garçons**:
   - A lógica `while (N <= (Gn - 1) * Gl) Gl--;` É um tratamento para evitar situações em que o número de clientes é insuficiente para ser atendido pelo total de garçons com sua capacidade. A variável `Gl` é decrementada até que a condição seja satisfeita.
5. **Preparação da Memória para Clientes e Garçons**:
   - Alocação de memória para as listas de IDs e semáforos dos clientes. Isso permite rastrear os clientes individualmente.
   - Definição de dois vetores de estruturas (ou structs) – um para clientes e outro para garçons. Estas estruturas contêm informações e parâmetros que serão passados para as threads quando forem criadas.

main.cpp - main

```
// inicializando os semáforos
sem_init(&sem_controle, 0, (G*Gn));
sem_init(&garcons_liberados, 0, Gl);
sem_init(&garcons_prontos, 0, 0);
sem_init(&mutex_clientes, 0, 1);
sem_init(&mutex_garcons, 0, 1);
sem_init(&mutex_rodada, 0, 1);
sem_init(&sem_pedidos, 0, 0);

// declarando as threads
pthread_t clientes[N];
pthread_t garcons[G];

// abrindo o bar
printf("\nBAR ABERTO: %d RODADAS GRÁTIS!!!\n\n", R);

// inicializando as threads dos garçons
for (int i = 0; i < G; i++){

    // passando os parâmetros para as threads garçons
    info_garcom[i].id_garcom = i + 1;
    info_garcom[i].lista_id_clientes = lista_id_clientes;
    info_garcom[i].lista_sem_clientes = lista_sem_clientes;

    pthread_create(&garcons[i], NULL, garcom, (void*)&info_garcom[i]);
}

// inicializando as threads dos clientes
for (int i = 0; i < N; i++){

    // passando os parâmetros para as threads clientes
    info_cliente[i].id_cliente = i + 1;
    info_cliente[i].lista_id_clientes = lista_id_clientes;
    info_cliente[i].lista_sem_clientes = lista_sem_clientes;

    pthread_create(&clientes[i], NULL, cliente, (void*)&info_cliente[i]);
}
```

```c
    // esperando as threads garçons terminarem
    for (int i = 0; i < G; i++) pthread_join(garcons[i], NULL);

    // esperando as threads clientes terminarem
    for (int i = 0; i < N; i++) pthread_join(clientes[i], NULL);

    // fechando o bar
    printf("BAR FECHADO: ATÉ A PRÓXIMA!!!\n\n");

    // liberando a memória alocada
    free(lista_id_clientes);
    free(lista_sem_clientes);

    // destruindo os semáforos
    sem_destroy(&sem_pedidos);
    sem_destroy(&sem_controle);
    sem_destroy(&mutex_rodada);
    sem_destroy(&mutex_garcons);
    sem_destroy(&mutex_clientes);
    sem_destroy(&garcons_prontos);
    sem_destroy(&garcons_liberados);

    return 0;
}
```

1. **inicialização dos Semáforos**:
    - `sem_init()` é usado para inicializar semáforos. O primeiro argumento é o semáforo, o segundo não é utilizado e o terceiro é o valor inicial.
2. **Declarando as Threads**:
    - Arrays `clientes` e `garcons` são declarados para armazenar as threads dos clientes e dos garçons, respectivamente.
3. **Mensagem Inicial**:
    - A mensagem "BAR ABERTO" é impressa, indicando o início das operações e a quantidade de rodadas grátis.
4. **Inicializando as Threads dos Garçons**:
    - Um loop é usado para inicializar cada thread de garçom. Para cada garçom, os parâmetros necessários são armazenados em uma struct `info_garcom`, e a thread é criada usando `pthread_create()`.
5. **Inicializando as Threads dos Clientes**:
    - Similarmente, outro loop é usado para inicializar as threads dos clientes.
6. **Esperando as Threads Terminarem**:
    - `pthread_join()` é usado para esperar até que uma thread termine sua execução. Dois loops são usados, um para esperar todos os garçons e outro para todos os clientes. Isso garante que o programa principal não prossiga até que todas as threads (clientes e garçons) tenham concluído suas operações.
7. **Mensagem de Encerramento**:
    - Uma mensagem informando que o bar está fechado é impressa.
8. **Liberando memória alocada:**
    - free() é usado para liberar memória alocada dinamicamente para lista_id_clientes e lista_sem_clientes, evitando, assim, o vazamento de memória.
9. **Destruição dos Semáforos:**
    - Todos os semáforos inicializados anteriormente são destruídos usando `sem_destroy()`.