Produto-Consumidor

1.0.0

Gerado por Doxygen 1.13.2

Capítulo 1

Índice das Estruturas de Dados

1.1 Estruturas de Dados

Aqui estão as estruturas de dados, uniões e suas respectivas descrições:

Capítulo 2

Índice dos Arquivos

2.1 Lista de Arquivos

Esta é a lista de todos os arquivos e suas respectivas descrições:

Capítulo 3

Estruturas

3.1 Referência da Estrutura consumer_args

Estrutura para encapsular os argumentos a serem passados para cada thread consumidora.

Campos de Dados

- int thread_id

3.1.1 Descrição detalhada

Estrutura para encapsular os argumentos a serem passados para cada thread consumidora.

Definição na linha 62 do arquivo q1_2.c.

3.1.2 Campos

3.1.2.1 thread_id

```
int thread_id
```

Definição na linha 64 do arquivo q1_2.c.

A documentação para essa estrutura foi gerada a partir do seguinte arquivo:

- q1_2.c

3.2 Referência da Estrutura producer_args

Estrutura para encapsular os argumentos a serem passados para cada thread produtora.

Campos de Dados

- int thread_id
- int num_sales

3.2.1 Descrição detalhada

Estrutura para encapsular os argumentos a serem passados para cada thread produtora.

Definição na linha 57 do arquivo q1_1.c.

3.2.2 Campos

3.2.2.1 num_sales

```
int num_sales
```

O número total de vendas que esta thread de caixa deve produzir.

Definição na linha 60 do arquivo q1_1.c.

3.2.2.2 thread_id

```
int thread_id
```

Um identificador único para a thread do caixa.

Definição na linha 59 do arquivo q1_1.c.

A documentação para essa estrutura foi gerada a partir dos seguintes arquivos:

- q1_1.c
- q1_2.c

Capítulo 4

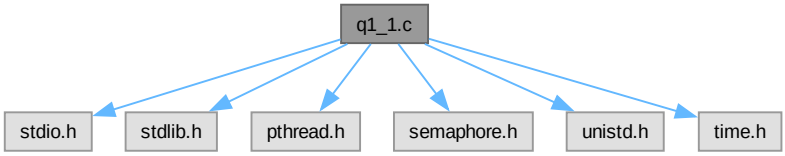
Arquivos

4.1 Referência do Arquivo q1_1.c

Simulação do problema Produtor-Consumidor usando pthreads, semáforos e variáveis de condição.

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <time.h>
```

Gráfico de dependência de inclusões para q1_1.c:



Estruturas de Dados

- struct producer_args

  Estrutura para encapsular os argumentos a serem passados para cada thread produtora.

Definições e Macros

- #define BUFFER_SIZE 5

  Define a capacidade máxima do buffer compartilhado.
- #define NUM_PRODUCERS 3

  Define o número de threads produtoras (caixas) a serem criadas.
- #define NUM_CONSUMERS 1

  Define o número de threads consumidoras (gerentes) a serem criadas.

Funções

- void * producer (void *args)

  Função executada pelas threads produtoras (caixas).
- void * consumer (void *args)

  Função executada pela thread consumidora (gerente).
- int main ()

  Ponto de entrada principal do programa.

Variáveis

- double buffer [BUFFER_SIZE]

  Array de doubles que funciona como o buffer circular compartilhado para armazenar os valores das vendas.
- int count = 0

  Contador que armazena o número atual de itens no buffer.
- int in_idx = 0

  Índice onde o próximo produtor irá inserir um item no buffer.
- int out_idx = 0

  Índice de onde o próximo consumidor irá remover um item do buffer.
- pthread_mutex_t mutex

  Mutex para garantir o acesso atômico às variáveis compartilhadas e ao buffer.
- sem_t empty_slots

  Semáforo que conta o número de posições vazias no buffer. Inicializado com BUFFER_SIZE.
- sem_t full_slots

  Semáforo que conta o número de posições preenchidas no buffer. Inicializado com 0.
- pthread_cond_t buffer_full_cond

  Variável de condição usada para sinalizar ao consumidor que o buffer está cheio.
- int active_producers = NUM_PRODUCERS

  Contador para rastrear o número de threads produtoras que ainda estão em execução.

4.1.1 Descrição detalhada

Simulação do problema Produtor-Consumidor usando pthreads, semáforos e variáveis de condição.

Este programa implementa uma solução para o problema clássico do produtor-consumidor. Ele simula um cenário com múltiplos "caixas" (produtores) que geram vendas (valores de ponto flutuante) e as colocam em um buffer circular compartilhado. Um único "gerente" (consumidor) aguarda até que o buffer esteja completamente cheio para então processar todas as vendas de uma vez, calculando o valor médio.

A sincronização entre as threads é gerenciada da seguinte forma:

- **Mutex (`mutex`):** Garante o acesso exclusivo às seções críticas, protegendo o buffer e as variáveis compartilhadas (`count`, `in_idx`, `out_idx`, `active_producers`) contra condições de corrida.
- **Semáforos (`empty_slots`, `full_slots`):** `empty_slots` controla o número de posições vazias no buffer, fazendo com que os produtores esperem se o buffer estiver cheio. `full_slots` foi mantido para ilustrar a solução clássica, embora o consumidor neste exemplo específico não espere por um único item.
- **Variável de Condição (`buffer_full_cond`):** Permite que o consumidor (gerente) espere de forma eficiente sem consumir CPU (`pthread_cond_wait`) até que o buffer esteja cheio ou que todos os produtores tenham terminado seu trabalho. Os produtores sinalizam (`pthread_cond_signal`) quando o buffer enche, e um `broadcast` é usado no final para garantir que o consumidor acorde e termine.

Definição no arquivo q1_1.c.

4.1.2 Definições e macros

4.1.2.1 BUFFER_SIZE

```
#define BUFFER_SIZE 5
```

Define a capacidade máxima do buffer compartilhado.

Definição na linha 34 do arquivo q1_1.c.

4.1.2.2 NUM_CONSUMERS

```
#define NUM_CONSUMERS 1
```

Define o número de threads consumidoras (gerentes) a serem criadas.

Definição na linha 46 do arquivo q1_1.c.

4.1.2.3 NUM_PRODUCERS

```
#define NUM_PRODUCERS 3
```

Define o número de threads produtoras (caixas) a serem criadas.

Definição na linha 40 do arquivo q1_1.c.

4.1.3 Funções

4.1.3.1 consumer()

```
void * consumer (
            void * args)
```

Função executada pela thread consumidora (gerente).

Função executada pelas threads consumidoras.

O consumidor entra em um loop infinito para processar as vendas. Ele bloqueia o mutex e aguarda na variável de condição (`pthread_cond_wait`) até que o buffer esteja cheio (`count == BUFFER_SIZE`) ou não haja mais produtores ativos (`active_producers == 0`). Quando acordado e a condição é satisfeita, ele processa *todos* os itens presentes no buffer, calculando a soma e a média. Em seguida, ele zera o contador de itens e libera os slots correspondentes no semáforo `empty_slots`. O loop termina quando não há mais produtores ativos e o buffer está vazio.

Parâmetros

args	Não utilizado (NULL).

Retorna

NULL.

Cada consumidor opera em um loop infinito, tentando processar vendas. Ele aguarda até que um item esteja disponível no buffer (`sem_wait(&full_slots)`). Após ser acordado, ele verifica a condição de término: se não há mais produtores ativos e o buffer está vazio. Se a condição for verdadeira, ele encerra. Caso contrário, ele adquire o bloqueio do mutex, consome um item do buffer, atualiza os contadores, libera o mutex e sinaliza que um espaço no buffer foi liberado (`sem_post(&empty_slots)`).

Parâmetros

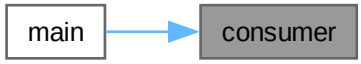
args	Ponteiro para uma estrutura `consumer_args` contendo o ID da thread.

Retorna

NULL.

Definição na linha 197 do arquivo q1_1.c.

Esse é o diagrama das funções que utilizam essa função:



4.1.3.2 main()

```
int main ()
```

Ponto de entrada principal do programa.

Inicializa o gerador de números aleatórios, o mutex, a variável de condição e os semáforos. Cria o número especificado de threads produtoras e consumidoras, passando os argumentos necessários. Aguarda a conclusão de todas as threads produtoras e consumidoras usando `pthread_join`. Por fim, destrói os primitivos de sincronização (mutex, cond, semáforos) e exibe uma mensagem de conclusão.

Retorna

0 em caso de sucesso.

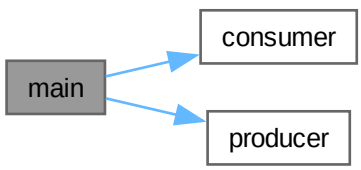
Inicializa os primitivos de sincronização (mutex e semáforos), cria as threads produtoras e consumidoras, e aguarda a conclusão de todas elas usando `pthread_join`. Após o término das threads, destrói os primitivos de sincronização e finaliza o programa.

Retorna

0 em caso de sucesso, ou um código de erro em caso de falha.

Definição na linha 266 do arquivo q1_1.c.

Este é o diagrama das funções utilizadas por essa função:



4.1.3.3 producer()

```
void * producer (
            void * args)
```

Função executada pelas threads produtoras (caixas).

Função executada pelas threads produtoras.

Cada produtor gera um número pré-definido de vendas com valores aleatórios. Para cada venda, ele aguarda por um slot vazio no buffer (`sem_wait`), bloqueia o mutex, adiciona o valor da venda ao buffer, atualiza os contadores e o índice de entrada. Se o buffer ficar cheio após a inserção, ele sinaliza a variável de condição `buffer_full_cond` para acordar o gerente. Após produzir todas as suas vendas, decrementa o contador `active_producers` e, se for o último produtor a terminar, envia um `broadcast` na variável de condição para garantir que o consumidor processe os itens restantes e termine.

Parâmetros

args	Um ponteiro para uma estrutura `producer_args` contendo o ID da thread e o número de vendas a produzir.

Retorna

NULL.

Cada produtor gera um número pré-definido de vendas (itens). Para cada venda, ele aguarda um espaço livre no buffer (`sem_wait(&empty_slots)`), adquire o bloqueio do mutex, insere o item, atualiza os contadores e libera o mutex. Por fim, sinaliza que um novo item está disponível para consumo (`sem_post(&full_slots)`). Ao final de sua produção, decrementa o contador `active_producers` e, se for o último produtor, acorda as threads consumidoras para que possam encerrar.

Parâmetros

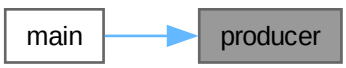
args	Ponteiro para uma estrutura `producer_args` contendo o ID da thread e o número de vendas a produzir.

Retorna

NULL.

Definição na linha 132 do arquivo q1_1.c.

Esse é o diagrama das funções que utilizam essa função:



4.1.4 Variáveis

4.1.4.1 active_producers

```
int active_producers = NUM_PRODUCERS
```

Contador para rastrear o número de threads produtoras que ainda estão em execução.

Definição na linha 115 do arquivo q1_1.c.

4.1.4.2 buffer

```
double buffer[BUFFER_SIZE]
```

Array de doubles que funciona como o buffer circular compartilhado para armazenar os valores das vendas.

Definição na linha 67 do arquivo q1_1.c.

4.1.4.3 buffer_full_cond

```
pthread_cond_t buffer_full_cond
```

Variável de condição usada para sinalizar ao consumidor que o buffer está cheio.

Definição na linha 109 do arquivo q1_1.c.

4.1.4.4 count

```
int count = 0
```

Contador que armazena o número atual de itens no buffer.

Definição na linha 73 do arquivo q1_1.c.

4.1.4.5 empty_slots

```
sem_t empty_slots
```

Semáforo que conta o número de posições vazias no buffer. Inicializado com BUFFER_SIZE.

Definição na linha 97 do arquivo q1_1.c.

4.1.4.6 full_slots

```
sem_t full_slots
```

Semáforo que conta o número de posições preenchidas no buffer. Inicializado com 0.

Definição na linha 103 do arquivo q1_1.c.

4.1.4.7 in_idx

```
int in_idx = 0
```

Índice onde o próximo produtor irá inserir um item no buffer.

Definição na linha 79 do arquivo q1_1.c.

4.1.4.8 mutex

```
pthread_mutex_t mutex
```

Mutex para garantir o acesso atômico às variáveis compartilhadas e ao buffer.

Definição na linha 91 do arquivo q1_1.c.

4.1.4.9 out_idx

```
int out_idx = 0
```

Índice de onde o próximo consumidor irá remover um item do buffer.

Definição na linha 85 do arquivo q1_1.c.

4.2 q1_1.c

Ir para a documentação desse arquivo.

```
00001
00022
00023 #include <stdio.h>
00024 #include <stdlib.h>
00025 #include <pthread.h>
00026 #include <semaphore.h>
00027 #include <unistd.h>
00028 #include <time.h>
00029
00034 #define BUFFER_SIZE 5
00035
00040 #define NUM_PRODUCERS 3
00041
00046 #define NUM_CONSUMERS 1
00047
00057 typedef struct
00058 {
00059     int thread_id;
00060     int num_sales;
00061 } producer_args;
00062
00067 double buffer[BUFFER_SIZE];
00068
00073 int count = 0;
00074
00079 int in_idx = 0;
00080
00085 int out_idx = 0;
00086
00091 pthread_mutex_t mutex;
00092
00097 sem_t empty_slots;
00098
00103 sem_t full_slots;
00104
00109 pthread_cond_t buffer_full_cond;
00110
```

```
00115 int active_producers = NUM_PRODUCERS;
00116
00132 void *producer(void *args)
00133 {
00134     producer_args *p_args = (producer_args *)args;
00135     int tid = p_args->thread_id;
00136     int sales_to_produce = p_args->num_sales;
00137
00138     for (size_t i = 0; i < sales_to_produce; i++)
00139     {
00140         double sale_value = (rand() % 100000) / 100.0 + 1.0; // Gera um valor de venda aleatório entre
      1.00 e 1000.00
00141
00142         sem_wait(&empty_slots);
00143
00144         pthread_mutex_lock(&mutex);
00145
00146         buffer[in_idx] = sale_value;
00147         in_idx = (in_idx + 1) % BUFFER_SIZE;
00148         count++;
00149
00150         printf("(P) TID %ld | Caixa %d | VENDA: R$ %.2f | ITERAÇÃO: %d/%d | Buffer: %d/%d\n",
00151                pthread_self(), tid, sale_value, i + 1, sales_to_produce, count, BUFFER_SIZE);
00152
00153         if (count == BUFFER_SIZE)
00154         {
00155             printf("--- BUFFER CHEIO! Notificando o gerente. ---\n");
00156             pthread_cond_signal(&buffer_full_cond);
00157         }
00158
00159         pthread_mutex_unlock(&mutex);
00160
00161         sem_post(&full_slots);
00162
00163         sleep((rand() % 5) + 1);
00164     }
00165
00166     pthread_mutex_lock(&mutex);
00167     active_producers--;
00168
00169     printf("(P) TID %ld | Caixa %d finalizou sua produção. Produtores ativos: %d\n",
00170            pthread_self(), tid, active_producers);
00171
00172     if (active_producers == 0)
00173     {
00174         pthread_cond_broadcast(&buffer_full_cond);
00175     }
00176     pthread_mutex_unlock(&mutex);
00177
00178     free(p_args);
00179     pthread_exit(NULL);
00180 }
00181
00197 void *consumer(void *args)
00198 {
00199     int iteration = 1;
00200
00201     while (1)
00202     {
00203         pthread_mutex_lock(&mutex);
00204
00205         while (count < BUFFER_SIZE && active_producers > 0)
00206         {
00207             printf("(C) TID %ld | Gerente esperando o buffer encher (Atual: %d/%d)...\n",
00208                    pthread_self(), count, BUFFER_SIZE);
00209             pthread_cond_wait(&buffer_full_cond, &mutex);
00210         }
00211
00212         if (active_producers == 0 && count == 0)
00213         {
00214             pthread_mutex_unlock(&mutex);
00215             break;
00216         }
00217
00218         if (count > 0)
00219         {
00220             printf("(C) TID %ld | Gerente iniciando processamento de %d vendas. ITERAÇÃO: %d\n",
00221                    pthread_self(), count, iteration);
00222
00223             double total_sum = 0.0;
00224             int items_consumed = count;
00225
00226             for (int i = 0; i < items_consumed; i++)
00227             {
00228                 double sale_value = buffer[out_idx];
00229                 total_sum += sale_value;
00230                 out_idx = (out_idx + 1) % BUFFER_SIZE;
```

```
00231             }
00232             count = 0;
00233
00234             double average = total_sum / items_consumed;
00235             printf("(C) TID %ld | MÉDIA das %d vendas: R$ %.2f | ITERAÇÃO: %d\n",
00236                    pthread_self(), items_consumed, average, iteration++);
00237
00238             pthread_mutex_unlock(&mutex);
00239
00240             for (int i = 0; i < items_consumed; i++)
00241             {
00242                 sem_post(&empty_slots);
00243             }
00244         }
00245         else
00246         {
00247             pthread_mutex_unlock(&mutex);
00248         }
00249     }
00250
00251     printf("(C) TID %ld | Gerente finalizou. Não há mais produtores nem vendas a processar.\n",
      pthread_self());
00252     pthread_exit(NULL);
00253 }
00254
00266 int main()
00267 {
00268     pthread_t producers[NUM_PRODUCERS];
00269     pthread_t consumers[NUM_CONSUMERS];
00270
00271     srand(time(NULL));
00272
00273     pthread_mutex_init(&mutex, NULL);
00274     pthread_cond_init(&buffer_full_cond, NULL);
00275
00276     sem_init(&empty_slots, 0, BUFFER_SIZE);
00277     sem_init(&full_slots, 0, 0);
00278
00279     printf("--- Iniciando Simulação de Gerenciamento de Caixas ---\n");
00280     printf("Configuração: %d Produtores (Caixas), %d Consumidor (Gerente), Tamanho do Buffer: %d\n\n",
00281            NUM_PRODUCERS, NUM_CONSUMERS, BUFFER_SIZE);
00282
00283     for (size_t i = 0; i < NUM_PRODUCERS; i++)
00284     {
00285         producer_args *args = malloc(sizeof(producer_args));
00286         args->thread_id = i + 1;
00287         args->num_sales = (rand() % 11) + 20; // Cada produtor fará entre 20 e 30 vendas
00288         pthread_create(&producers[i], NULL, producer, (void *)args);
00289     }
00290
00291     for (size_t i = 0; i < NUM_CONSUMERS; i++)
00292     {
00293         pthread_create(&consumers[i], NULL, consumer, NULL);
00294     }
00295
00296     for (size_t i = 0; i < NUM_PRODUCERS; i++)
00297     {
00298         pthread_join(producers[i], NULL);
00299     }
00300
00301     for (size_t i = 0; i < NUM_CONSUMERS; i++)
00302     {
00303         pthread_join(consumers[i], NULL);
00304     }
00305
00306     pthread_mutex_destroy(&mutex);
00307     pthread_cond_destroy(&buffer_full_cond);
00308     sem_destroy(&empty_slots);
00309     sem_destroy(&full_slots);
00310
00311     printf("\n--- Simulação Concluída ---\n");
00312
00313     return 0;
00314 }
```

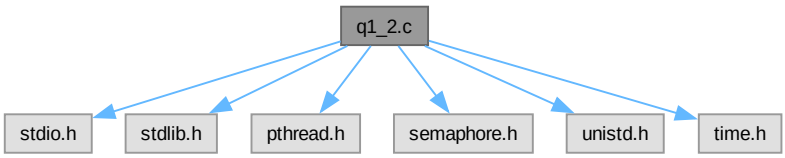
4.3 Referência do Arquivo q1_2.c

Simulação do problema Produtor-Consumidor com múltiplos produtores e consumidores.

```
#include <stdio.h>
#include <stdlib.h>
```

```
#include <pthread.h>
#include <semaphore.h>
#include <unistd.h>
#include <time.h>
```

Gráfico de dependência de inclusões para q1_2.c:



Estruturas de Dados

- struct producer_args

  Estrutura para encapsular os argumentos a serem passados para cada thread produtora.
- struct consumer_args

  Estrutura para encapsular os argumentos a serem passados para cada thread consumidora.

Definições e Macros

- #define BUFFER_SIZE 5

  Define a capacidade máxima do buffer compartilhado.
- #define NUM_PRODUCERS 6

  Define o número de threads produtoras (caixas) a serem criadas.
- #define NUM_CONSUMERS 2

  Define o número de threads consumidoras (gerentes) a serem criadas.

Funções

- void ∗ producer (void ∗args)
- void ∗ consumer (void ∗args)
- int main ()

Variáveis

- double buffer [BUFFER_SIZE]
- int count = 0
- int in_idx = 0
- int out_idx = 0
- pthread_mutex_t mutex
- sem_t empty_slots
- sem_t full_slots
- pthread_cond_t buffer_empty_cond
- volatile int active_producers = NUM_PRODUCERS

4.3.1 Descrição detalhada

Simulação do problema Produtor-Consumidor com múltiplos produtores e consumidores.

Este programa implementa uma solução para o problema clássico do Produtor-Consumidor utilizando múltiplas threads para produtores (caixas de uma loja) e consumidores (gerentes). A comunicação entre eles é feita através de um buffer circular compartilhado.

A sincronização é gerenciada pelos seguintes primitivos:

- Mutex (`mutex`): Garante acesso exclusivo ao buffer compartilhado e às variáveis de controle (`count`, `in_idx`, `out_idx`, `active_producers`), prevenindo condições de corrida.
- Semáforo (`empty_slots`): Controla o número de posições vazias no buffer. Produtores esperam neste semáforo se o buffer estiver cheio.
- Semáforo (`full_slots`): Controla o número de itens disponíveis no buffer. Consumidores esperam neste semáforo se o buffer estiver vazio.

A lógica de término é coordenada pela variável `active_producers`. Cada produtor, ao concluir seu trabalho, decrementa este contador. O último produtor a terminar notifica todas as threads consumidoras (via `sem_post`) para que elas possam verificar a condição de término (não há produtores ativos e o buffer está vazio) e encerrar sua execução.

Definição no arquivo q1_2.c.

4.3.2 Definições e macros

4.3.2.1 BUFFER_SIZE

```
#define BUFFER_SIZE 5
```

Define a capacidade máxima do buffer compartilhado.

Definição na linha 34 do arquivo q1_2.c.

4.3.2.2 NUM_CONSUMERS

```
#define NUM_CONSUMERS 2
```

Define o número de threads consumidoras (gerentes) a serem criadas.

Definição na linha 46 do arquivo q1_2.c.

4.3.2.3 NUM_PRODUCERS

```
#define NUM_PRODUCERS 6
```

Define o número de threads produtoras (caixas) a serem criadas.

Definição na linha 40 do arquivo q1_2.c.

4.3.3 Funções

4.3.3.1 consumer()

```
void * consumer (
            void * args)
```

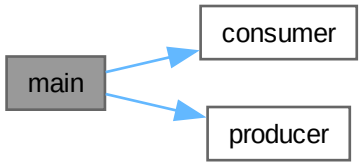
Definição na linha 152 do arquivo q1_2.c.

4.3.3.2 main()

```
int main ()
```

Definição na linha 210 do arquivo q1_2.c.

Este é o diagrama das funções utilizadas por essa função:



4.3.3.3 producer()

```
void * producer (
            void * args)
```

Definição na linha 94 do arquivo q1_2.c.

4.3.4 Variáveis

4.3.4.1 active_producers

```
volatile int active_producers = NUM_PRODUCERS
```

Definição na linha 78 do arquivo q1_2.c.

4.3.4.2 buffer

```
double buffer[BUFFER_SIZE]
```

Definição na linha 67 do arquivo q1_2.c.

4.3.4.3 buffer_empty_cond

```
pthread_cond_t buffer_empty_cond
```

Definição na linha 75 do arquivo q1_2.c.

4.3.4.4 count

```
int count = 0
```

Definição na linha 68 do arquivo q1_2.c.

4.3.4.5 empty_slots

```
sem_t empty_slots
```

Definição na linha 73 do arquivo q1_2.c.

4.3.4.6 full_slots

```
sem_t full_slots
```

Definição na linha 74 do arquivo q1_2.c.

4.3.4.7 in_idx

```
int in_idx = 0
```

Definição na linha 69 do arquivo q1_2.c.

4.3.4.8 mutex

```
pthread_mutex_t mutex
```

Definição na linha 72 do arquivo q1_2.c.

4.3.4.9 out_idx

```
int out_idx = 0
```

Definição na linha 70 do arquivo q1_2.c.

4.4 q1_2.c

Ir para a documentação desse arquivo.

```
00001
00022
00023 #include <stdio.h>
00024 #include <stdlib.h>
00025 #include <pthread.h>
00026 #include <semaphore.h>
00027 #include <unistd.h>
00028 #include <time.h>
00029
00034 #define BUFFER_SIZE 5
00035
00040 #define NUM_PRODUCERS 6
00041
00046 #define NUM_CONSUMERS 2
00047
00052 typedef struct
00053 {
00054     int thread_id;
00055     int num_sales;
00056 } producer_args;
00057
00062 typedef struct
00063 {
00064     int thread_id;
00065 } consumer_args;
00066
00067 double buffer[BUFFER_SIZE];
00068 int count = 0;
00069 int in_idx = 0;
00070 int out_idx = 0;
00071
00072 pthread_mutex_t mutex;
00073 sem_t empty_slots;
00074 sem_t full_slots;
00075 pthread_cond_t buffer_empty_cond; // Usada para garantir o término correto
00076
00077 // Volatile para garantir que a leitura mais recente seja usada por todas as threads
00078 volatile int active_producers = NUM_PRODUCERS;
00079
00094 void *producer(void *args)
00095 {
00096     producer_args *p_args = (producer_args *)args;
00097     int tid = p_args->thread_id;
00098     int sales_to_produce = p_args->num_sales;
00099
00100     for (size_t i = 0; i < sales_to_produce; i++)
00101     {
00102         double sale_value = (rand() % 100000) / 100.0 + 1.0;
00103
00104         sem_wait(&empty_slots); // Espera por um slot vazio
00105
00106         pthread_mutex_lock(&mutex);
00107         buffer[in_idx] = sale_value;
00108         in_idx = (in_idx + 1) % BUFFER_SIZE;
00109         count++;
00110         printf("(P) TID %d | VENDA: R$ %.2f | Buffer: %d/%d\n",
00111                tid, sale_value, count, BUFFER_SIZE);
00112         pthread_mutex_unlock(&mutex);
00113
00114         sem_post(&full_slots); // Sinaliza que um slot foi preenchido
00115
00116         sleep((rand() % 3) + 1); // Pausa menor para aumentar a concorrência
00117     }
00118
00119     // No final do producer
00120     pthread_mutex_lock(&mutex);
00121     active_producers--;
00122     printf("»» (P) Caixa %d finalizou. Produtores ativos: %d ««\n", tid, active_producers);
00123     if (active_producers == 0)
00124     {
00125         // Acorda TODOS os consumidores que possam estar esperando no sem_wait.
00126         // Eles irão acordar, verificar a condição de término e sair.
00127         for (int i = 0; i < NUM_CONSUMERS; i++)
00128         {
00129             sem_post(&full_slots);
00130         }
00131     }
00132     pthread_mutex_unlock(&mutex);
00133
00134     free(p_args);
00135     pthread_exit(NULL);
00136 }
```

```
00137
00152 void *consumer(void *args)
00153 {
00154     consumer_args *c_args = (consumer_args *)args;
00155     int tid = c_args->thread_id;
00156     int sales_processed = 0;
00157
00158     while (1)
00159     {
00160         // Espera por um item. Este é o ponto de bloqueio.
00161         sem_wait(&full_slots);
00162
00163         // Após acordar, a primeira coisa é verificar se devemos terminar.
00164         // Bloqueamos o mutex para ler 'count' e 'active_producers' de forma segura.
00165         pthread_mutex_lock(&mutex);
00166         if (active_producers == 0 && count == 0)
00167         {
00168             // Não há mais produtores e o buffer está vazio. O trabalho acabou.
00169             // Precisamos liberar o mutex antes de sair.
00170             pthread_mutex_unlock(&mutex);
00171
00172             // Como consumimos um 'sem_wait' para entrar aqui,
00173             // mas não vamos consumir um item, precisamos devolver o "ticket"
00174             // para que outra thread consumidora também possa sair.
00175             sem_post(&full_slots);
00176
00177             break;
00178         }
00179
00180         // Se chegamos aqui, há um item para consumir.
00181         double sale_value = buffer[out_idx];
00182         out_idx = (out_idx + 1) % BUFFER_SIZE;
00183         count--;
00184         sales_processed++;
00185
00186         printf("    (C) TID %d | PROCESSOU: R$ %.2f | Buffer: %d/%d\n",
00187                tid, sale_value, count, BUFFER_SIZE);
00188
00189         pthread_mutex_unlock(&mutex);
00190
00191         // Libera um slot vazio para os produtores.
00192         sem_post(&empty_slots);
00193     }
00194
00195     printf("»» (C) Gerente %d finalizou. Total de vendas processadas: %d ««\n", tid, sales_processed);
00196     free(c_args);
00197     pthread_exit(NULL);
00198 }
00199
00210 int main()
00211 {
00212     pthread_t producers[NUM_PRODUCERS];
00213     pthread_t consumers[NUM_CONSUMERS];
00214
00215     srand(time(NULL));
00216
00217     pthread_mutex_init(&mutex, NULL);
00218     pthread_cond_init(&buffer_empty_cond, NULL);
00219
00220     // Inicializa semáforos
00221     sem_init(&empty_slots, 0, BUFFER_SIZE); // Começa com N slots vazios
00222     sem_init(&full_slots, 0, 0);            // Começa com 0 slots preenchidos
00223
00224     printf("--- Iniciando Simulação com %d Produtores e %d Consumidores ---\n\n",
00225            NUM_PRODUCERS, NUM_CONSUMERS);
00226
00227     // Cria as threads produtoras
00228     for (int i = 0; i < NUM_PRODUCERS; i++)
00229     {
00230         producer_args *args = malloc(sizeof(producer_args));
00231         args->thread_id = i + 1;
00232         args->num_sales = (rand() % 6) + 5; // Menos vendas para a simulação ser mais rápida
00233         pthread_create(&producers[i], NULL, producer, (void *)args);
00234     }
00235
00236     // Cria as threads consumidoras
00237     for (int i = 0; i < NUM_CONSUMERS; i++)
00238     {
00239         consumer_args *args = malloc(sizeof(consumer_args));
00240         args->thread_id = i + 1;
00241         pthread_create(&consumers[i], NULL, consumer, (void *)args);
00242     }
00243
00244     // Espera todas as threads terminarem
00245     for (int i = 0; i < NUM_PRODUCERS; i++)
00246     {
00247         pthread_join(producers[i], NULL);
```

```
00248     }
00249
00250     // Após os produtores terminarem, precisamos garantir que os consumidores acordem
00251     // caso estejam esperando em sem_wait.
00252     for (int i = 0; i < NUM_CONSUMERS; i++)
00253     {
00254         sem_post(&full_slots);
00255     }
00256
00257     for (int i = 0; i < NUM_CONSUMERS; i++)
00258     {
00259         pthread_join(consumers[i], NULL);
00260     }
00261
00262     // Destrói os primitivos de sincronização
00263     pthread_mutex_destroy(&mutex);
00264     pthread_cond_destroy(&buffer_empty_cond);
00265     sem_destroy(&empty_slots);
00266     sem_destroy(&full_slots);
00267
00268     printf("\n--- Simulação Concluída ---\n");
00269
00270     return 0;
00271 }
```

Índice Remissivo