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 producer_args

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

Campos de Dados

- int thread_id
- int num_sales

3.1.1 Descrição detalhada

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

Definição na linha 57 do arquivo prod-cons.c.

3.1.2 Campos

3.1.2.1 num_sales

```
int num_sales
```

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

Definição na linha 60 do arquivo prod-cons.c.

3.1.2.2 thread_id

```
int thread_id
```

Um identificador único para a thread do caixa.

Definição na linha 59 do arquivo prod-cons.c.

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

- prod-cons.c

Capítulo 4

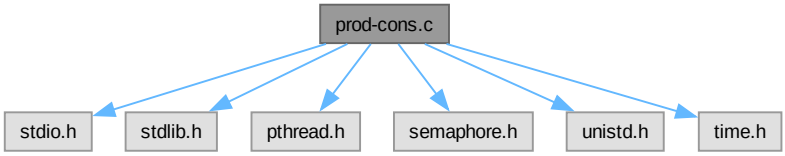
Arquivos

4.1 Referência do Arquivo prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.c.

4.1.3 Funções

4.1.3.1 consumer()

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

Função executada pela thread consumidora (gerente).

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

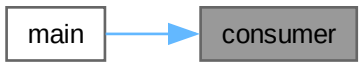
args	Não utilizado (NULL).

Retorna

NULL.

Definição na linha 197 do arquivo prod-cons.c.

Esse é o diagrama das funções que utilizam essa função:



4.1.3.2 main()

```
int main ()
```

Ponto de entrada principal do programa.

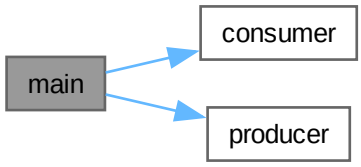
Inicializa o gerador de números aleatórios, o mutex, a variável de condição e os semáforos. Cria o número especificado de threads produtoras e consumidoras, passando os argumentos necessários. Aguarda a conclusão de todas as threads produtoras e consumidoras usando `pthread_join`. Por fim, destrói os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

Retorna

0 em caso de sucesso.

Definição na linha 266 do arquivo prod-cons.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).

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

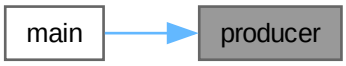
args	Um ponteiro para uma estrutura `producer_args` contendo o ID da thread e o número de vendas a produzir.

Retorna

NULL.

Definição na linha 132 do arquivo prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.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 prod-cons.c.

4.2 prod-cons.c

Ir para a documentação desse arquivo.

```
00001 #include <stdio.h>
00002 #include <stdlib.h>
00003 #include <pthread.h>
00004 #include <semaphore.h>
00005 #include <unistd.h>
00006 #include <time.h>
00007
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 }
```

Índice Remissivo