Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\buffer.c

```
//src\buffer.c

#include "buffer.h"
#include

int init_buffer(Buffer *buffer, int capacity) {
 buffer->data = (int*) malloc(capacity * sizeof(int));
 if (!buffer->data)
 return -1;
 buffer->capacity = capacity;
 buffer->count = 0;
 buffer->front = 0;
 buffer->rear = 0;
 return 0;
}

void destroy_buffer(Buffer *buffer) {
 if (buffer->data)
 free(buffer->data);
}

int insert_item(Buffer *buffer, int item) {
 if(buffer->count == buffer->capacity)
 return -1; // Buffer cheio
 buffer->data[buffer->rear] = item;
 buffer->rear = (buffer->rear + 1) % buffer->capacity;
 buffer->count++;
 return 0;
}

int remove_item(Buffer *buffer, int *item) {
 if(buffer->count == 0)
 return -1; // Buffer vazio
 *item = buffer->data[buffer->front];
 buffer->front = (buffer->front + 1) % buffer->capacity;
 buffer->count--;
 return 0;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\buffer.h

```
//src\buffer.h

#ifndef BUFFER_H
#define BUFFER_H

typedef struct {
 int *data; // Array que armazena os itens
 int capacity; // Tamanho máximo do buffer
 int count; // Número atual de itens no buffer
 int front; // Índice de remoção (próximo item a ser consumido)
 int rear; // Índice de inserção (próximo local livre)
} Buffer;

// Inicializa o buffer com a capacidade especificada.
```

```
// Retorna 0 em caso de sucesso, ou -1 em caso de erro.
int init_buffer(Buffer *buffer, int capacity);

// Libera a memória alocada pelo buffer.
void destroy_buffer(Buffer *buffer);

// Insere um item no buffer circular.
// Retorna 0 se a inserção foi bem-sucedida, -1 se o buffer estiver cheio.
int insert_item(Buffer *buffer, int item);

// Remove um item do buffer circular (seguindo a ordem FIFO).
// Armazena o item removido no parâmetro "item" e retorna 0 em sucesso,
// ou -1 se o buffer estiver vazio.
int remove_item(Buffer *buffer, int *item);

#endif
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\main.c

```
//src\main.c

#include
#include
#include
#include
#include
#include
#include
#include "buffer.h"

#define DEFAULT_BUFFER_SIZE 7
#define DEFAULT_PROD 2
#define DEFAULT_CONS 4
#define DEFAULT_RUNTIME 10

// Variáveis globais para sincronização
Buffer buffer;
pthread_mutex_t mutex;
sem_t empty, full;

// Métricas (contadores)
int produced_count = 0;
int consumed_count = 0;

// Parâmetros definidos na linha de comando
int numProd = DEFAULT_PROD;
int numCons = DEFAULT_CONS;
int buffer_size = DEFAULT_BUFFER_SIZE;
int runtime_seconds = DEFAULT_RUNTIME;

void *producer(void *param) {
 int item;
 while (1) {
 sleep(1); // Simula o tempo de produção
 item = rand() % 100; // Produz um item aleatório

 sem_wait(∅);
 pthread_mutex_lock(&mutex;);

 if (insert_item(&buffer;, item) == 0) {
```

```
 produced_count++;
 printf("Produzido: %d | Buffer count: %d\n", item, buffer.count);
 } else {
 printf("Erro: Buffer cheio (insercao falhou)\n");
 }

 pthread_mutex_unlock(&mutex;);
 sem_post(&full;);
 }
 return NULL;
}

void *consumer(void *param) {
 int item;
 while (1) {
 sleep(2); // Simula o tempo de consumo

 sem_wait(&full;);
 pthread_mutex_lock(&mutex;);

 if (remove_item(&buffer;, &item;) == 0) {
 consumed_count++;
 printf("Consumido: %d | Buffer count: %d\n", item, buffer.count);
 } else {
 printf("Erro: Buffer vazio (remoção falhou)\n");
 }

 pthread_mutex_unlock(&mutex;);
 sem_post(∅);
 }
 return NULL;
}

void print_usage(char *prog_name) {
 printf("Uso: %s [--prod ] [--cons ] [--buffer ] [--runtime ]\n", prog_name);
}

int main(int argc, char *argv[]) {
 srand((unsigned int)time(NULL));

 // Parsing dos parâmetros via linha de comando usando getopt_long
 static struct option long_options[] = {
 {"prod", required_argument, 0, 'p'},
 {"cons", required_argument, 0, 'c'},
 {"buffer", required_argument, 0, 'b'},
 {"runtime", required_argument, 0, 'r'},
 {"help", no_argument, 0, 'h'},
 {0,0,0,0}
 };

 int opt;
 int option_index = 0;
 while ((opt = getopt_long(argc, argv, "p:c:b:r:h", long_options, &option;_index)) !=
-1) {
 switch (opt) {
 case 'p':
 numProd = atoi(optarg);
 break;
 case 'c':
 numCons = atoi(optarg);
 break;
 case 'b':
 buffer_size = atoi(optarg);
```

```
 break;
 case 'r':
 runtime_seconds = atoi(optarg);
 break;
 case 'h':
 default:
 print_usage(argv[0]);
 exit(EXIT_SUCCESS);
 }
 }

 // Inicializa o buffer (buffer circular)
 if(init_buffer(&buffer;, buffer_size) != 0) {
 fprintf(stderr, "Erro ao alocar buffer\n");
 exit(EXIT_FAILURE);
 }

 // Inicializa as primitivas de sincronização
 pthread_mutex_init(&mutex;, NULL);
 sem_init(∅, 0, buffer.capacity);
 sem_init(&full;, 0, 0);

 // Cria os arrays para threads produtores e consumidores
 pthread_t *producers = malloc(numProd * sizeof(pthread_t));
 pthread_t *consumers = malloc(numCons * sizeof(pthread_t));

 // Cria as threads produtoras
 for (int i = 0; i < numProd; i++) {
 if (pthread_create(&producers;[i], NULL, producer, NULL) != 0) {
 perror("Erro ao criar thread produtor");
 }
 }

 // Cria as threads consumidoras
 for (int i = 0; i < numCons; i++) {
 if (pthread_create(&consumers;[i], NULL, consumer, NULL) != 0) {
 perror("Erro ao criar thread consumidor");
 }
 }

 // Executa a simulação pelo tempo definido
 sleep(runtime_seconds);

 // Exibe as métricas (simples contadores)
 printf("\n== Dados da Execucao ==\n");
 printf("Itens produzidos: %d\n", produced_count);
 printf("Itens consumidos: %d\n", consumed_count);


 free(producers);
 free(consumers);
 destroy_buffer(&buffer;);
 pthread_mutex_destroy(&mutex;);
 sem_destroy(∅);
 sem_destroy(&full;);

 return 0;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\main_nosync.c

```
// main_nosync.c (sem parsing)
#include
#include
#include
#include
#include
#include "buffer.h"

int numProd = 2, numCons = 4, buffer_size = 7, runtime_seconds = 10;
int produced_count = 0, consumed_count = 0;
Buffer buffer;

void *producer(void *arg) {
 while (1) {
 sleep(1);
 int item = rand() % 100;
 if (insert_item(&buffer;, item) == 0) {
 produced_count++;
 printf("[NOSYNC] Produzido: %d | count=%d\n", item, buffer.count);
 } else {
 printf("[NOSYNC] Buffer cheio, item %d descartado\n", item);
 }
 }
 return NULL;
}

void *consumer(void *arg) {
 while (1) {
 sleep(2);
 int item;
 if (remove_item(&buffer;, &item;) == 0) {
 consumed_count++;
 printf("[NOSYNC] Consumido: %d | count=%d\n", item, buffer.count);
 } else {
 printf("[NOSYNC] Buffer vazio\n");
 }
 }
 return NULL;
}

int main(int argc, char **argv) {
 srand((unsigned)time(NULL));
 init_buffer(&buffer;, buffer_size);

 pthread_t prod[numProd], cons[numCons];
 for (int i = 0; i < numProd; i++)
 pthread_create(∏[i], NULL, producer, NULL);
 for (int i = 0; i < numCons; i++)
 pthread_create(&cons;[i], NULL, consumer, NULL);

 sleep(runtime_seconds);
 printf("\n[NOSYNC] Itens produzidos=%d, consumidos=%d\n",
 produced_count, consumed_count);
 return 0;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\monitor.c

```
//src\monitor.c

#include "monitor.h"
#include
#include

// Buffer compartilhado
static Buffer buffer;
static pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
static pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
static pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void monitor_init() {
 buffer.count = 0;
 buffer.in = 0;
 buffer.out = 0;
}

void monitor_destroy() {
 pthread_mutex_destroy(&mutex;);
 pthread_cond_destroy(¬_full);
 pthread_cond_destroy(¬_empty);
}

void monitor_insert(int item) {
 pthread_mutex_lock(&mutex;);

 while (buffer.count == BUFFER_SIZE) {
 pthread_cond_wait(¬_full, &mutex;);
 }

 buffer.data[buffer.in] = item;
 buffer.in = (buffer.in + 1) % BUFFER_SIZE;
 buffer.count++;

 printf("[Prod %ld] Inserido: %d | count = %d\n", pthread_self(), item, buffer.count);

 pthread_cond_signal(¬_empty);
 pthread_mutex_unlock(&mutex;);
}

int monitor_remove() {
 pthread_mutex_lock(&mutex;);

 while (buffer.count == 0) {
 pthread_cond_wait(¬_empty, &mutex;);
 }

 int item = buffer.data[buffer.out];
 buffer.out = (buffer.out + 1) % BUFFER_SIZE;
 buffer.count--;

 printf("[Cons %ld] Removido: %d | count = %d\n", pthread_self(), item, buffer.count);

 pthread_cond_signal(¬_full);
 pthread_mutex_unlock(&mutex;);

 return item;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\monitor.h

```
//src\monitor.h

#ifndef MONITOR_H
#define MONITOR_H

#define BUFFER_SIZE 5

// Estrutura do buffer
typedef struct {
 int data[BUFFER_SIZE];
 int count;
 int in;
 int out;
} Buffer;

// Interface do monitor simulado
void monitor_init();
void monitor_destroy();
void monitor_insert(int item);
int monitor_remove();

#endif
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\monitorSimulado.c

```
//src\monitorSimulado.c

#include
#include
#include
#include
#include "monitor.h"

void* producer(void* arg) {
 while (1) {
 sleep(1);
 int item = rand() % 100;
 monitor_insert(item);
 }
 return NULL;
}

void* consumer(void* arg) {
 while (1) {
 sleep(2);
 int item = monitor_remove();
 }
 return NULL;
}

int main() {
 srand(time(NULL));
 monitor_init();
```

```
 int qtdProd = 2;
 int qtdCons = 3;
 pthread_t prod[qtdProd], cons[qtdCons];

 for (int i = 0; i < qtdProd; i++) {
 pthread_create(∏[i], NULL, producer, NULL);
 }

 for (int i = 0; i < qtdCons; i++) {
 pthread_create(&cons;[i], NULL, consumer, NULL);
 }

 sleep(10);
 monitor_destroy();
 pthread_exit(NULL);
 return 0;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\mutexCond.c

```
// src/mutexCond.c
#include
#include
#include
#include "buffer.h"
#include
#include

Buffer buffer;
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
pthread_cond_t not_full = PTHREAD_COND_INITIALIZER;
pthread_cond_t not_empty = PTHREAD_COND_INITIALIZER;

void* producer (void *param) {
 int item;
 while (1) {
 sleep(1); // Simula o tempo de produção
 item = rand() % 100;

 pthread_mutex_lock(&mutex;);
 while (buffer.count == buffer.capacity) {
 pthread_cond_wait(¬_full, &mutex;);
 }

 insert_item(&buffer;, item);
 printf("[Prod %lu] Inserido: %d | count = %d\n", (unsigned long)pthread_self(), item,
buffer.count);

 pthread_cond_signal(¬_empty);
 pthread_mutex_unlock(&mutex;);
 }
 return NULL;
}

void* consumer (void *param) {
 int item;
 while (1) {
 sleep(2); // Simula o tempo de consumo
```

```
 pthread_mutex_lock(&mutex;);
 while (buffer.count == 0) {
 pthread_cond_wait(¬_empty, &mutex;);
 }

 remove_item(&buffer;, &item;);
 printf("[Cons %lu] Removido: %d | count = %d\n", (unsigned long)pthread_self(), item,
buffer.count);

 pthread_cond_signal(¬_full);
 pthread_mutex_unlock(&mutex;);
 }
 return NULL;
}

int main() {
 srand((unsigned int)time(NULL));

 if (init_buffer(&buffer;, 5) != 0) {
 printf("Erro ao inicializar o buffer\n");
 exit(1);
 }

 int mainSleepTime = 10;
 int qtdProdutores = 2;
 int qtdConsumidores = 4;

 pthread_t producers[qtdProdutores];
 pthread_t consumers[qtdConsumidores];

 for (int i = 0; i < qtdProdutores; i++) {
 pthread_create(&producers;[i], NULL, producer, NULL);
 }
 for (int i = 0; i < qtdConsumidores; i++) {
 pthread_create(&consumers;[i], NULL, consumer, NULL);
 }

 sleep(mainSleepTime);

 printf("\n== Execucao Finalizada (mutexCond) ==\n");
 return 0;
}
```

Arquivo: \Users\joao-\Desktop\so\producer_consumer_problem\src\run.sh

```
#!/usr/bin/env bash
#
# Uso: ./run.sh  [--prod N] [--cons M] [--buffer S] [--runtime T]
# mod os: nosync | sem | mutex | monitor

if [ $# -lt 1 ]; then
 echo "Uso: $0  [--prod N] [--cons M] [--buffer S] [--runtime T]"
 exit 1
fi

mode=$1; shift

case "$mode" in
 nosync) exe=prodcons_nosync ;;
 sem) exe=prodcons_sem ;;
```

```
 mutex) exe=prodcons_mutex ;;
 monitor) exe=prodcons_monitor;;
 *)
 echo "Modo inválido: $mode"
 exit 1
 ;;
esac

# Passa todos os outros parâmetros para o executável escolhido
./$exe "$@"
```