**Producteur / Consommateur**

Un **producteur** fabrique des données et les place dans un **tampon (buffer)**.  
Un **consommateur** lit ces données et les traite.  
Un **buffer de taille m** contient au plus m éléments.  
Les **accès concurrentiels** au buffer doivent être **synchronisés** pour éviter :

* que le producteur n’écrive dans un tampon plein,
* que le consommateur ne lise dans un tampon vide,
* que plusieurs processus n’accèdent simultanément à la même zone mémoire.

**🔧 Sémaphores utilisés**

* vide : compteur de **places vides disponibles** dans le buffer (initialisé à m)
* plein : compteur de **cases pleines disponibles** (initialisé à 0)
* mutex : exclusion mutuelle pour l’accès concurrent au buffer (initialisé à 1)

**✅ CAS 1 : Un seul producteur, un seul consommateur, buffer de taille 1**

**Hypothèse :**

* 1 seule case dans le buffer, donc les sémaphores vide et plein suffisent,
* **mutex inutile**, car il n'y a **pas de concurrence**.

**🔧 Initialisation :**

* vide = 1, plein = 0

**Producteur :**

P(vide); // Attendre une case vide

buffer := donnée; // Placer l'élément

V(plein); // Signal qu'une case est pleine

**Consommateur :**

P(plein); // Attendre une case pleine

donnée := buffer; // Lire l'élément

V(vide); // Signal qu'une case est vide

✅ Simple et sûr sans mutex.

**✅ CAS 2 : Un seul producteur, un seul consommateur, buffer de taille m**

**Hypothèse :**

* Il y a plusieurs cases (m > 1), mais toujours **aucune concurrence**, donc pas besoin de mutex.

**🔧 Variables :**

* In : position d’écriture du producteur (modulo m)
* Out : position de lecture du consommateur (modulo m)
* buffer[m] : tableau circulaire

**🔧 Initialisation :**

* vide = m, plein = 0, mutex inutile

**Producteur :**

P(vide); // Attendre une case vide

buffer[In] := donnée; // Placer l’élément dans le buffer

In := (In + 1) mod m;

V(plein); // Signal qu’une case est pleine

**Consommateur :**

P(plein); // Attendre une case pleine

donnée := buffer[Out]; // Lire depuis le buffer

Out := (Out + 1) mod m;

V(vide); // Signal qu’une case est vide

✅ Toujours pas besoin de mutex car il n’y a pas de course d’accès.

**✅ CAS 3 : Plusieurs producteurs, plusieurs consommateurs, buffer de taille m**

**Hypothèse :**

* Il y a **concurrence** sur buffer, In, Out, donc mutex devient **obligatoire**.

**🔧 Initialisation :**

* vide = m, plein = 0, mutex = 1

**Producteur :**

P(vide); // Attendre une case vide

P(mutex); // Entrer en section critique

buffer[In] := donnée;

In := (In + 1) mod m;

V(mutex); // Sortie de section critique

V(plein); // Signale qu’une case est pleine

**Consommateur :**

P(plein); // Attendre une case pleine

P(mutex); // Entrer en section critique

donnée := buffer[Out];

Out := (Out + 1) mod m;

V(mutex); // Sortie de section critique

V(vide); // Signale qu’une case est vide

**Programme de simulation générale.**

/\*

\* producer\_consumer.c

\*

\* Producer-Consumer problem implementation in C using pthreads and POSIX semaphores.

\* Supports two choices:

\* 1. Execute with user-defined number of producers, consumers, buffer size, and simulation time.

\* 2. Quit the program.

\*

\* The user selects the choice from a menu.

\* After selection, producer and consumer threads run for the specified time, then stop and return to the menu.

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <stdbool.h>

int RUN\_TIME\_SECONDS = 10;

#define MAX\_BUFFER\_SIZE 100

#define MAX\_PRODUCERS 10

#define MAX\_CONSUMERS 10

// Buffer and synchronization variables

int \*buffer;

int buffer\_size;

int in = 0;

int out = 0;

sem\_t empty\_slots; // Counts empty slots in buffer

sem\_t full\_slots; // Counts full slots in buffer

sem\_t mutex; // Protects buffer access

volatile bool running = false;

// For multiple producers/consumers

int num\_producers = 0;

int num\_consumers = 0;

void \*producer(void \*arg) {

int id = \*(int \*)arg;

while (running) {

int item = rand() % 1000; // Produce a random item

sem\_wait(&empty\_slots);

sem\_wait(&mutex);

buffer[in] = item;

printf("Producer %d: produced item %d at buffer index %d\n", id, item, in);

in = (in + 1) % buffer\_size;

sem\_post(&mutex);

sem\_post(&full\_slots);

//usleep(100000); // Sleep for 100ms

}

return NULL;

}

void \*consumer(void \*arg) {

int id = \*(int \*)arg;

while (running) {

sem\_wait(&full\_slots);

sem\_wait(&mutex);

int item = buffer[out];

printf("Consumer %d: consumed item %d from buffer index %d\n", id, item, out);

out = (out + 1) % buffer\_size;

sem\_post(&mutex);

sem\_post(&empty\_slots);

//usleep(150000); // Sleep for 150ms

}

return NULL;

}

void run\_multiple\_producers\_consumers() {

pthread\_t prod\_threads[MAX\_PRODUCERS];

pthread\_t cons\_threads[MAX\_CONSUMERS];

int prod\_ids[MAX\_PRODUCERS];

int cons\_ids[MAX\_CONSUMERS];

running = true;

for (int i = 0; i < num\_producers; i++) {

prod\_ids[i] = i + 1;

pthread\_create(&prod\_threads[i], NULL, producer, &prod\_ids[i]);

}

for (int i = 0; i < num\_consumers; i++) {

cons\_ids[i] = i + 1;

pthread\_create(&cons\_threads[i], NULL, consumer, &cons\_ids[i]);

}

int elapsed = 0;

while (elapsed < RUN\_TIME\_SECONDS) {

// sleep(1);

elapsed++;

}

running = false;

// Unblock any waiting producers and consumers

for (int i = 0; i < num\_producers; i++) {

sem\_post(&empty\_slots);

}

for (int i = 0; i < num\_consumers; i++) {

sem\_post(&full\_slots);

}

for (int i = 0; i < num\_producers; i++) {

pthread\_join(prod\_threads[i], NULL);

}

for (int i = 0; i < num\_consumers; i++) {

pthread\_join(cons\_threads[i], NULL);

}

}

int main() {

while (1) {

printf("\nProducer-Consumer Problem Menu:\n");

printf("1 - Execute\n");

printf("2 - Quit\n");

printf("Enter choice: ");

int choice;

if (scanf("%d", &choice) != 1) {

printf("Invalid input. Exiting.\n");

break;

}

if (choice == 2) {

printf("Exiting program.\n");

break;

}

if (choice != 1) {

printf("Invalid choice. Try again.\n");

continue;

}

printf("Enter number of producers (1-%d): ", MAX\_PRODUCERS);

if (scanf("%d", &num\_producers) != 1 || num\_producers < 1 || num\_producers > MAX\_PRODUCERS) {

printf("Invalid number of producers. Try again.\n");

continue;

}

printf("Enter number of consumers (1-%d): ", MAX\_CONSUMERS);

if (scanf("%d", &num\_consumers) != 1 || num\_consumers < 1 || num\_consumers > MAX\_CONSUMERS) {

printf("Invalid number of consumers. Try again.\n");

continue;

}

printf("Enter buffer size m (1-%d): ", MAX\_BUFFER\_SIZE);

if (scanf("%d", &buffer\_size) != 1 || buffer\_size < 1 || buffer\_size > MAX\_BUFFER\_SIZE) {

printf("Invalid buffer size. Try again.\n");

continue;

}

printf("Enter simulation time in seconds (1-3600): ");

int simulation\_time;

if (scanf("%d", &simulation\_time) != 1 || simulation\_time < 1 || simulation\_time > 3600) {

printf("Invalid simulation time. Try again.\n");

continue;

}

buffer = (int \*)malloc(sizeof(int) \* buffer\_size);

if (buffer == NULL) {

printf("Failed to allocate buffer memory.\n");

continue;

}

sem\_init(&empty\_slots, 0, buffer\_size);

sem\_init(&full\_slots, 0, 0);

sem\_init(&mutex, 0, 1);

in = 0;

out = 0;

RUN\_TIME\_SECONDS = simulation\_time;

printf("Running %d producers and %d consumers for %d seconds...\n", num\_producers, num\_consumers, RUN\_TIME\_SECONDS);

run\_multiple\_producers\_consumers();

sem\_destroy(&empty\_slots);

sem\_destroy(&full\_slots);

sem\_destroy(&mutex);

free(buffer);

printf("Run finished. Returning to menu.\n");

}

return 0;

}