**Lecteurs / Rédacteurs**

Le **problème lecteur-rédacteur (reader-writer problem)** est un **problème classique de synchronisation** en programmation concurrente. Il s'agit de gérer **l'accès concurrent** à une **ressource partagée** (ex: base de données), par deux types de processus :

* **Lecteurs** (reader) : peuvent lire la ressource en parallèle s’il n’y a pas d’écriture.
* **Rédacteurs** (writer) : doivent avoir **l’exclusivité** d'accès à la ressource.

**Contrainte fondamentale**

* Plusieurs lecteurs peuvent lire **simultanément**.
* **Un seul rédacteur** peut accéder à la ressource, et **aucun lecteur ne doit lire** pendant l’écriture.

**🌐 SÉMANTIQUE DES SÉMAPHORES UTILISÉS**

* P(S) (proberen, "tester") :

S := S - 1

si S < 0 alors bloquer le processus appelant sur S

sinon continuer l’exécution

* V(S) (verhogen, "augmenter") :

S := S + 1

si S <= 0 alors réveiller un processus bloqué sur S

**Cas 1 : Priorité aux lecteurs**

**Idée :**

* Laisser passer les lecteurs tant qu’il n’y a pas de rédacteur actif.
* Bloquer les rédacteurs tant qu’il y a **au moins un lecteur**.

**🔧 Variables et sémaphores utilisés :**

* mutex : protège la variable nb\_lecteurs
* S : sémaphore binaire, protège la **section critique principale**
* nb\_lecteurs : entier partagé, compte le nombre de lecteurs actifs

**✍️ Code pour un lecteur :**

P(mutex);

nb\_lecteurs := nb\_lecteurs + 1;

si nb\_lecteurs == 1 alors P(S); // Premier lecteur bloque les rédacteurs

V(mutex);

// Lecture de val (section critique)

P(mutex);

nb\_lecteurs := nb\_lecteurs - 1;

si nb\_lecteurs == 0 alors V(S); // Dernier lecteur libère la ressource

V(mutex);

**✍️ Code pour un rédacteur :**

P(S); // Exclusivité totale

// Écriture de la ressource (section critique)

V(S);

**Cas 2 : Priorité aux rédacteurs**

**Problème :**

* Empêcher une **affluence de lecteurs** qui retarderait indéfiniment un rédacteur.

**🔧 Ajout de sémaphores :**

* mutex : protège nb\_lecteurs
* mutex2 : protège nb\_redacteurs
* S : section critique principale
* lecture\_ok : bloque les lecteurs si un rédacteur attend
* nb\_lecteurs, nb\_redacteurs : entiers partagés

**✍️ Code pour un lecteur :**

c

CopierModifier

P(lecture\_ok); // Vérifie s’il y a un rédacteur actif ou en attente

P(mutex);

nb\_lecteurs := nb\_lecteurs + 1;

si nb\_lecteurs == 1 alors P(S); // Premier lecteur bloque la section critique

V(mutex);

V(lecture\_ok);

// Lecture de la ressource

P(mutex);

nb\_lecteurs := nb\_lecteurs - 1;

si nb\_lecteurs == 0 alors V(S); // Dernier lecteur libère la section critique

V(mutex);

**✍️ Code pour un rédacteur :**

P(mutex2);

nb\_redacteurs := nb\_redacteurs + 1;

si nb\_redacteurs == 1 alors P(lecture\_ok); // Premier rédacteur empêche de nouveaux lecteurs

V(mutex2);

P(S); // Accès exclusif à la ressource

// Écriture de la ressource

V(S);

P(mutex2);

nb\_redacteurs := nb\_redacteurs - 1;

si nb\_redacteurs == 0 alors V(lecture\_ok); // Dernier rédacteur relâche les lecteurs

V(mutex2);

**Cas 3 : Priorité égale (pas de famine)**

**But :**

* Garantir une **équité** entre lecteurs et rédacteurs, souvent avec une **file d’attente (queue équitable)**.

**Solution typique :**

Utiliser un **sémaphore file** pour forcer les processus à passer dans **l’ordre d’arrivée**, lecteurs et rédacteurs inclus.

**🔧 Sémaphores :**

* file : pour garantir l'ordre d'arrivée
* mutex : pour protéger les compteurs
* S : section critique principale
* nb\_lecteurs

**📜 Algorithme pour le lecteur équitable :**

P(file); // Prendre un ticket

P(mutex);

nb\_lecteurs := nb\_lecteurs + 1;

si nb\_lecteurs == 1 alors P(S); // Premier lecteur bloque les rédacteurs

V(mutex);

V(file);

// Lecture de la ressource

P(mutex);

nb\_lecteurs := nb\_lecteurs - 1;

si nb\_lecteurs == 0 alors V(S); // Dernier lecteur libère la ressource

V(mutex);

**📜 Algorithme pour le rédacteur équitable :**

c

CopierModifier

P(file); // Prendre un ticket

P(S); // Accès exclusif

// Écriture

V(S);

V(file); // Libère le ticket

**✅ Résumé des trois cas**

| **Cas** | **Avantage** | **Inconvénient** |
| --- | --- | --- |
| Priorité aux lecteurs | Rapide pour lectures fréquentes | Rédacteurs peuvent être bloqués longtemps |
| Priorité aux rédacteurs | Bonne réactivité en écriture | Lecteurs peuvent être bloqués longtemps |
| Priorité équitable | Évite famine | Moins performant dans cas extrêmes |

**Programme simulateur Lecteurs / Rédacteurs**

\*

\* readers\_writers.c

\*

\* Readers-Writers problem implementation in C using pthreads and POSIX semaphores.

\* Supports three priority modes:

\* 1. Readers priority

\* 2. Writers priority

\* 3. Equal (fair) priority

\*

\* The user selects the mode from a menu.

\* After selection, reader and writer threads run for a fixed time, then stop and return to the menu.

\* Synchronization is done only using semaphores (no condition variables).

\*/

#include <stdio.h>

#include <stdlib.h>

#include <pthread.h>

#include <semaphore.h>

#include <unistd.h>

#include <stdbool.h>

#define NUM\_READERS 15

#define NUM\_WRITERS 33

#define RUN\_TIME\_SECONDS 10

typedef enum {

READERS\_PRIORITY = 1,

WRITERS\_PRIORITY,

EQUAL\_PRIORITY

} priority\_mode\_t;

priority\_mode\_t mode;

// Shared resource

int shared\_data = 0;

// Synchronization variables

sem\_t mutex; // Protects access to read\_count

sem\_t write\_sem; // Controls access to the shared resource

sem\_t read\_try; // Used in writers priority and equal priority to block readers when writers waiting

int read\_count = 0;

int write\_count = 0;

volatile bool running = false;

void \*reader(void \*arg) {

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

while (running) {

switch (mode) {

case READERS\_PRIORITY:

sem\_wait(&mutex);

read\_count++;

if (read\_count == 1) {

sem\_wait(&write\_sem);

}

sem\_post(&mutex);

// Reading section

printf("Reader %d: reading shared\_data = %d\n", id, shared\_data);

usleep(100000);

sem\_wait(&mutex);

read\_count--;

if (read\_count == 0) {

sem\_post(&write\_sem);

}

sem\_post(&mutex);

break;

case WRITERS\_PRIORITY:

sem\_wait(&read\_try);

sem\_wait(&mutex);

read\_count++;

if (read\_count == 1) {

sem\_wait(&write\_sem);

}

sem\_post(&mutex);

sem\_post(&read\_try);

// Reading section

printf("Reader %d: reading shared\_data = %d\n", id, shared\_data);

usleep(100000);

sem\_wait(&mutex);

read\_count--;

if (read\_count == 0) {

sem\_post(&write\_sem);

}

sem\_post(&mutex);

break;

case EQUAL\_PRIORITY:

sem\_wait(&read\_try);

sem\_wait(&mutex);

read\_count++;

if (read\_count == 1) {

sem\_wait(&write\_sem);

}

sem\_post(&mutex);

sem\_post(&read\_try);

// Reading section

printf("Reader %d: reading shared\_data = %d\n", id, shared\_data);

usleep(100000);

sem\_wait(&mutex);

read\_count--;

if (read\_count == 0) {

sem\_post(&write\_sem);

}

sem\_post(&mutex);

break;

}

usleep(200000);

}

return NULL;

}

void \*writer(void \*arg) {

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

while (running) {

switch (mode) {

case READERS\_PRIORITY:

sem\_wait(&write\_sem);

// Writing section

shared\_data++;

printf("Writer %d: wrote shared\_data = %d\n", id, shared\_data);

usleep(150000);

sem\_post(&write\_sem);

break;

case WRITERS\_PRIORITY:

sem\_wait(&mutex);

write\_count++;

if (write\_count == 1) {

sem\_wait(&read\_try);

}

sem\_post(&mutex);

sem\_wait(&write\_sem);

// Writing section

shared\_data++;

printf("Writer %d: wrote shared\_data = %d\n", id, shared\_data);

usleep(150000);

sem\_post(&write\_sem);

sem\_wait(&mutex);

write\_count--;

if (write\_count == 0) {

sem\_post(&read\_try);

}

sem\_post(&mutex);

break;

case EQUAL\_PRIORITY:

sem\_wait(&mutex);

write\_count++;

if (write\_count == 1) {

sem\_wait(&read\_try);

}

sem\_post(&mutex);

sem\_wait(&write\_sem);

// Writing section

shared\_data++;

printf("Writer %d: wrote shared\_data = %d\n", id, shared\_data);

usleep(150000);

sem\_post(&write\_sem);

sem\_wait(&mutex);

write\_count--;

if (write\_count == 0) {

sem\_post(&read\_try);

}

sem\_post(&mutex);

break;

}

usleep(300000);

}

return NULL;

}

void run\_threads() {

pthread\_t r\_threads[NUM\_READERS];

pthread\_t w\_threads[NUM\_WRITERS];

int r\_ids[NUM\_READERS];

int w\_ids[NUM\_WRITERS];

running = true;

for (int i = 0; i < NUM\_READERS; i++) {

r\_ids[i] = i + 1;

pthread\_create(&r\_threads[i], NULL, reader, &r\_ids[i]);

}

for (int i = 0; i < NUM\_WRITERS; i++) {

w\_ids[i] = i + 1;

pthread\_create(&w\_threads[i], NULL, writer, &w\_ids[i]);

}

sleep(RUN\_TIME\_SECONDS);

running = false;

for (int i = 0; i < NUM\_READERS; i++) {

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

}

for (int i = 0; i < NUM\_WRITERS; i++) {

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

}

}

int main() {

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

sem\_init(&write\_sem, 0, 1);

sem\_init(&read\_try, 0, 1);

while (1) {

printf("\nSelect priority mode:\n");

printf("1 - Readers priority\n");

printf("2 - Writers priority\n");

printf("3 - Equal priority\n");

printf("0 - Quit\n");

printf("Enter choice: ");

int choice;

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

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

break;

}

if (choice == 0) {

printf("Exiting program.\n");

break;

}

if (choice < 1 || choice > 3) {

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

continue;

}

mode = (priority\_mode\_t)choice;

printf("Running with mode %d for %d seconds...\n", choice, RUN\_TIME\_SECONDS);

run\_threads();

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

}

sem\_destroy(&mutex);

sem\_destroy(&write\_sem);

sem\_destroy(&read\_try);

return 0;

}