D:\Lab_Sistemi-Operativi\Tutorato-Sistemi-Operativi-2025-main\Esame-[2025-09-17]\gemini_equisum_mmap.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <unistd.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <pthread.h>
#include <string.h>

// COSTANTI DEL PROBLEMA
#define M 12 // Dimensione fissa di ogni vettore
#define BUFFER_SIZE 10  // Dimensione della coda intermedia
#define NUM_VERIFIERS 3 // Numero fisso di verificatori

// Funzioni per il controllo dell'output
#define LOCK_IO()   flockfile(stdout)
#define UNLOCK_IO() funlockfile(stdout)

// ====================== STRUTTURE DATI ======================

// Record: il pacchetto di dati che viaggia tra i thread
// Contiene dati grezzi dei file
typedef struct {
    unsigned char dati[M];
    int file_id;
    int vector_idx;
} Record;

// SharedData: la struttura "MONITOR" che contiene le risorse condivise
typedef struct {
    // --- 1. CODA INTERMEDIA ---
    // Serve per passare dati da lettori a verificatori
    Record buffer[BUFFER_SIZE];
    int in; // Indice dove scrivere il prossimo dato
    int out; // Indice dove leggere il prossimo dato
    int count; // Quanti elementi ci sono nel buffer
    bool buffer_closed; // Diventa TRUE quando i lettori hanno finito

    // Sincronizzazione Coda Intermedia
    pthread_mutex_t mtx_buffer;
    pthread_cond_t cond_buf_not_empty;
    pthread_cond_t cond_buf_not_full;

    // Slot Finale
    // Serve per passare i risultati dei verificatori al main
    Record slot_finale;
    bool slot_full;
    bool slot_closed;

    // Sincronizzazione Slot Finale
```

```
52      pthread_mutex_t mtx_slot;
53      pthread_cond_t cond_slot_empty;
54      pthread_cond_t cond_slot_full;
55
56      // Contatori per gestione chiusura automatica
57      int readers_active; // Quanti lettori sono ancora vivi
58      int verifiers_active; // Quanti verificatori sono ancora vivi
59      pthread_mutex_t mtx_counts; // Protegge questi contatori
60
61  } SharedData;
62
63  // ====================== HELPER ======================
64  // Funzione per stampare il vettore
65  void print_vec(unsigned char *v) {
66      for (int i = 0; i < M; i++) {
67          printf("%u", v[i]);
68          if (i < M - 1) printf(", ");
69      }
70  }
71
72  // ====================== GESTIONE MONITOR ======================
73
74  // Inizializzo tutti i mutex e le condition
75  void init_shared(SharedData *s, int n_readers) {
76      // Coda intermedia
77      s->in = 0; s->out = 0; s->count = 0; s->buffer_closed = false;
78      pthread_mutex_init(&s->mtx_buffer, NULL);
79      pthread_cond_init(&s->cond_buf_not_empty, NULL);
80      pthread_cond_init(&s->cond_buf_not_full, NULL);
81
82      // Slot finale
83      s->slot_full = false; s->slot_closed = false;
84      pthread_mutex_init(&s->mtx_slot, NULL);
85      pthread_cond_init(&s->cond_slot_empty, NULL);
86      pthread_cond_init(&s->cond_slot_full, NULL);
87
88      // Contatori
89      s->readers_active = n_readers;
90      s->verifiers_active = NUM_VERIFIERS;
91      pthread_mutex_init(&s->mtx_counts, NULL);
92  }
93
94  // ---------------- FUNZIONI BUFFER ----------------
95
96  // ---------------- PRODUTTORE ----------------
97  // Inserisce un record nella coda intermedia. Se è piena, aspetta
98  void buffer_put(SharedData *s, Record r) {
99      pthread_mutex_lock(&s->mtx_buffer);
100
101     // Aspetto finchè è vuoto E non è ancora chiuso
102     while (s->count == BUFFER_SIZE && !s->buffer_closed)
103         pthread_cond_wait(&s->cond_buf_not_full, &s->mtx_buffer);
104     if (!s->buffer_closed) {
105         // SEZIONE CRITICA: scrivo nel buffer
```

```
106         s->buffer[s->in] = r;
107         s->in = (s->in + 1) % BUFFER_SIZE;
108         s->count++;
109 
110         // Informo i consumatori del nuovo dato in più
111         pthread_cond_signal(&s->cond_buf_not_empty);
112     }
113     pthread_mutex_unlock(&s->mtx_buffer);
114 }
115 
116 // ----------------- CONSUMATORE -----------------
117 // Preleva un record. Ritorna true se ha letto, false se è vuota o chiusa
118 bool buffer_get(SharedData *s, Record *r) {
119     pthread_mutex_lock(&s->mtx_buffer);
120     // Aspetto finchè è vuoto e non ancora chiuso
121     while (s->count == 0 && !s->buffer_closed)
122         pthread_cond_wait(&s->cond_buf_not_empty, &s->mtx_buffer);
123 
124     bool res = false;
125     // Se c'è qualcosa da leggere(count > 0), leggo!
126     if (s->count > 0) {
127         *r = s->buffer[s->out];
128         s->out = (s->out + 1) % BUFFER_SIZE;
129         s->count--;
130 
131         // Segnalo che si è liberato un posto ai produttori
132         pthread_cond_signal(&s->cond_buf_not_full);
133         res = true;
134     }
135     // Se count == 0 e buffer_closed == true, allora res rimane false (segnale di fine)
136 
137     pthread_mutex_unlock(&s->mtx_buffer);
138     return res;
139 }
140 
141 // Chiude la coda intermedia: sveglia tutti quelli che dormono
142 void buffer_close(SharedData *s) {
143     pthread_mutex_lock(&s->mtx_buffer);
144     s->buffer_closed = true;
145     pthread_cond_broadcast(&s->cond_buf_not_empty);
146     pthread_cond_broadcast(&s->cond_buf_not_full);
147     pthread_mutex_unlock(&s->mtx_buffer);
148 }
149 
150 // ----------------- FUNZIONI SLOT FINALE -----------------
151 
152 void slot_put(SharedData *s, Record r) {
153     pthread_mutex_lock(&s->mtx_slot);
154 
155     // Aspetto se è pieno(il main non ha ancora letto)
156     while (s->slot_full && !s->slot_closed)
157         pthread_cond_wait(&s->cond_slot_empty, &s->mtx_slot);
158     if (!s->slot_closed) {
159         s->slot_finale = r;
```

```
160         s->slot_full = true;
161         pthread_cond_signal(&s->cond_slot_full); // Sveglia main
162     }
163     pthread_mutex_unlock(&s->mtx_slot);
164 }
165 
166 bool slot_get(SharedData *s, Record *r) {
167     pthread_mutex_lock(&s->mtx_slot);
168     // Aspetto se è vuoto (i verificatori non hanno trovato nulla)
169     while (!s->slot_full && !s->slot_closed)
170         pthread_cond_wait(&s->cond_slot_full, &s->mtx_slot);
171     bool res = false;
172     if (s->slot_full) {
173         *r = s->slot_finale;
174         s->slot_full = false; // Ho letto, ora è vuoto
175         pthread_cond_signal(&s->cond_slot_empty); // Sveglia verificatori
176         res = true;
177     }
178     pthread_mutex_unlock(&s->mtx_slot);
179     return res;
180 }
181 
182 void slot_close(SharedData *s) {
183     pthread_mutex_lock(&s->mtx_slot);
184     s->slot_closed = true;
185     pthread_cond_broadcast(&s->cond_slot_full);
186     pthread_cond_broadcast(&s->cond_slot_empty);
187     pthread_mutex_unlock(&s->mtx_slot);
188 }
189 
190 // ===================== THREADS =====================
191 
192 // Strutture per passare argomenti multipli a pthread_create
193 typedef struct {
194     SharedData *S;
195     char *filename;
196     int id;
197 } ReaderArgs;
198 
199 typedef struct {
200     SharedData *S;
201     int id;
202 } VerifierArgs;
203 
204 // ---------------- THREAD LETTORE ----------------
205 void *reader_thread(void *arg) {
206     ReaderArgs *a = (ReaderArgs *)arg;
207 
208     LOCK_IO(); printf("[READER-%d] file '%s'\n", a->id, a->filename); UNLOCK_IO();
209 
210     // 1. Apertura File
211     int fd = open(a->filename, O_RDONLY);
212     if (fd == -1) { perror("open"); return NULL; } // Gestione errore base
213 
```

```c
    struct stat sb; fstat(fd, &sb); // Recupero dimensione file
    size_t fsize = sb.st_size;

    // 2. Mappatura in Memoria (MMAP)
    // Usiamo un puntatore map che punta ai dati del file
    unsigned char *map = mmap(NULL, fsize, PROT_READ, MAP_PRIVATE, fd, 0);
    close(fd); // Il descrittore non serve più una volta mappato

    int num_vec = fsize / M; // Numero totale di vettori nel file

    // 3. Ciclo di lettura
    for (int i = 0; i < num_vec; i++) {
        Record r;

        // Copia di memoria grezza: dall'area mappata (file) alla struct locale
        memcpy(r.dati, map + (i*M), M);
        r.file_id = a->id; r.vector_idx = i + 1;

        LOCK_IO();
        printf("[READER-%d] vettore candidato n.%d: ", a->id, i + 1);
        print_vec(r.dati);
        printf("\n");
        UNLOCK_IO();

        // Invio alla Coda Intermedia (si blocca se piena)
        buffer_put(a->S, r);
    }
    // Pulizia mmap
    if (map != MAP_FAILED) munmap(map, fsize);

    LOCK_IO(); printf("[READER-%d] terminazione con %d vettori letti\n", a->id, num_vec);
UNLOCK_IO();

    // Logica Last Man Standing per i lettori
    // Ogni lettore che finisce decrementa il contatore.
    pthread_mutex_lock(&a->S->mtx_counts);
    a->S->readers_active--;
    // Se sono l'ultimo lettore rimasto, devo chiudere la coda.
    if (a->S->readers_active == 0) buffer_close(a->S); // Sblocco i verificatori
    pthread_mutex_unlock(&a->S->mtx_counts);

    return NULL;
}

// ----------------- THREAD VERIFICATORE -----------------
void *verifier_thread(void *arg) {
    VerifierArgs *a = (VerifierArgs *)arg;
    Record r;
    int count = 0;

    // 1. Ciclo di Prelievo
    // buffer_get ritorna true finché ci sono dati.
    // Ritorna false solo quando la coda è vuota E buffer_close() è stato chiamato
dall'ultimo lettore.
```

```c
    while (buffer_get(a->S, &r)) {
        count++;
        int p = 0, d = 0;
        for (int i=0; i<M; i++) {
            if(i%2==0) p+=r.dati[i];
            else d+=r.dati[i];
        }

        LOCK_IO();
        printf("[VERIF-%d] verifico vettore: ", a->id);
        print_vec(r.dati);
        printf("\n");
        if (p == d) {
            printf("[VERIF-%d] si tratta di un vettore equisomma con somma %d!\n", a->id, p);
            UNLOCK_IO(); // Sblocco prima di put per evitare attese con lock output

            // 2. Trovato! Invio allo slot Finale (per il Main)
            slot_put(a->S, r);
        } else {
            printf("[VERIF-%d] non è un vettore equisomma (somma pari %d vs. dispari %d)\n", a->id, p, d);
            UNLOCK_IO();
        }
    }

    LOCK_IO(); printf("[VERIF-%d] terminazione con %d vettori verificati\n", a->id, count); UNLOCK_IO();

    // 3. Logica Last Man Standing per i verificatori
    pthread_mutex_lock(&a->S->mtx_counts);
    a->S->verifiers_active--;
    // Se sono l'ultimo verificatore, chiudo lo slot finale
    if (a->S->verifiers_active == 0) slot_close(a->S); // Questo sbloccherà il Main
    pthread_mutex_unlock(&a->S->mtx_counts);

    free(a); // Libero la memoria allocata nel main per gli argomenti
    return NULL;
}

// ----------------- MAIN -----------------
int main(int argc, char *argv[]) {
    // Controllo argomenti
    if (argc < 2) { printf("Uso: %s <f1> ...\n", argv[0]); return 1; }
    int n_readers = argc - 1; // Un reader per ogni file passato
    SharedData S;

    // 1. Inizializzazione Strutture Condivise
    init_shared(&S, n_readers);

    LOCK_IO();
    printf("[MAIN] creazione di %d thread lettori e %d thread verificatori\n", n_readers, NUM_VERIFIERS);
    UNLOCK_IO();
```

```c

    pthread_t readers[n_readers], verifiers[NUM_VERIFIERS];
    ReaderArgs r_args[n_readers];

    // 2. Avvio Thread Lettori
    for (int i=0; i<n_readers; i++) {
        r_args[i].S = &S; r_args[i].filename = argv[i+1]; r_args[i].id = i+1;
        pthread_create(&readers[i], NULL, reader_thread, &r_args[i]);
    }

    // 3. Avvio Thread Verificatori
    for (int i=0; i<NUM_VERIFIERS; i++) {
        // Uso malloc per passare argomenti diversi ad ogni thread
        VerifierArgs *va = malloc(sizeof(VerifierArgs));
        va->S = &S; va->id = i+1;
        pthread_create(&verifiers[i], NULL, verifier_thread, va);
    }

    // 4. Ciclo Consumatore del Main
    Record r;
    int total = 0;

    // Il Main si comporta come un consumatore dello Slot Finale.
    // slot_get ritorna false solo quando l'ultimo verificatore chiama slot_close.
    while (slot_get(&S, &r)) {
        total++;
        LOCK_IO();
        printf("[MAIN] ricevuto nuovo vettore equisomma: ");
        print_vec(r.dati);
        printf("\n");
        UNLOCK_IO();
    }

    // 5. Attesa (Join) e Pulizia
    for(int i=0;i<n_readers;i++) pthread_join(readers[i], NULL);
    for(int i=0;i<NUM_VERIFIERS;i++) pthread_join(verifiers[i], NULL);

    LOCK_IO(); printf("[MAIN] terminazione con %d vettori equisomma trovati\n", total);
UNLOCK_IO();

    // Distruzione mutex e cond (buona prassi)
    // (Omesse le singole chiamate a destroy per brevità, ma andrebbero qui)
    return 0;
}
```