Tutorato-Sistemi-Operativi-2025-main\Esame-[2025-09-17]\main.c

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

#define M 12          // lunghezza dei vettori richiesti dall'esame
#define QUEUE_CAP 10  // capienza coda intermedia (vincolo d'esame)
#define NUM_VERIF 3   // numero di thread verificatori

// ================================
// == Record della coda (didatt.) ==
// ================================
// Usiamo direttamente un array di 12 byte.
// Aggiungiamo un flag didattico "is_end": se vale 1 il record è una sentinella
// di terminazione (poison pill) per i consumatori.
typedef struct {
    uint8_t v[M]; // il vettore da 12 byte
    int     is_end; // 0 = normale, 1 = sentinella di terminazione
} record_t;

// ================================
// == Coda circolare bounded FIFO ==
// ================================
// Producer-Consumer con semafori (sem_wait/sem_post) e un mutex.
// - sem_slots: quanti slot liberi ho per inserire
// - sem_items: quanti elementi disponibili per prelevare
// Push e pop sono *bloccanti* (nessun timed-wait).
typedef struct {
    record_t buf[QUEUE_CAP];
    int head;               // prossimo elemento da leggere
    int tail;               // prossimo slot dove scrivere
    int count;              // numero elementi attuali
    pthread_mutex_t mtx;    // mutua esclusione su head/tail/count
    sem_t sem_items;        // elementi disponibili
    sem_t sem_slots;        // slot liberi
} ring_t;

// ================================
// == Record finale (buffer 1-slot) ==
// ================================
// Anche qui aggiungiamo is_end per la terminazione del main.
typedef struct {
    record_t slot;        // singolo slot
    pthread_mutex_t mtx;  // protezione accesso a slot
```

```
52      sem_t sem_empty;      // 1 se slot vuoto
53      sem_t sem_full;       // 1 se slot pieno
54  } final_slot_t;
55
56  // ================================
57  // == Stato condiviso tra thread ==
58  // ================================
59  typedef struct {
60      // Code di comunicazione
61      ring_t ring;          // coda intermedia di 10 posti
62      final_slot_t out;     // record finale a 1 slot
63
64      // File da leggere
65      int n_readers;
66      char **filenames;
67
68      // Contatori per coordinare la terminazione (protetti da mutex)
69      int readers_left;     // lettori ancora attivi
70      int verifiers_left;   // verificatori ancora attivi
71      pthread_mutex_t mtx_counts;
72
73      // Statistiche
74      unsigned long main_equisum_total;
75
76  } shared_t;
77
78  // Argomento generico per i thread
79  typedef struct {
80      shared_t *S;
81      int idx; // indice umano (1..N per reader, 1..NUM_VERIF per verificatori)
82  } thread_arg_t;
83
84  // ============== Utilità di stampa ordinate ==============
85  // Evita mescolamento righe tra thread.
86  static inline void lock_stdout(void)   { flockfile(stdout); }
87  static inline void unlock_stdout(void) { funlockfile(stdout); }
88
89  // Stampa compatta di un vettore
90  static void print_vec12(const uint8_t v[M]) {
91      printf("(%u", v[0]);
92      for (int i = 1; i < M; ++i) {
93          printf(", %u", v[i]);
94      }
95      printf(")");
96  }
97
98  // ================== Inizializzazione strutture ==================
99  static void ring_init(ring_t *r) {
100     r->head = r->tail = r->count = 0;
101     pthread_mutex_init(&r->mtx, NULL);
102     sem_init(&r->sem_items, 0, 0);
103     sem_init(&r->sem_slots, 0, QUEUE_CAP);
104 }
105 static void ring_destroy(ring_t *r) {
```

```
106     pthread_mutex_destroy(&r->mtx);
107     sem_destroy(&r->sem_items);
108     sem_destroy(&r->sem_slots);
109 }
110 
111 static void final_init(final_slot_t *f) {
112     pthread_mutex_init(&f->mtx, NULL);
113     sem_init(&f->sem_empty, 0, 1); // all'inizio lo slot è vuoto
114     sem_init(&f->sem_full,  0, 0);
115 }
116 static void final_destroy(final_slot_t *f) {
117     pthread_mutex_destroy(&f->mtx);
118     sem_destroy(&f->sem_empty);
119     sem_destroy(&f->sem_full);
120 }
121 
122 // ================== Operazioni sulla coda ==================
123 // PUSH (bloccante): aspetta uno slot libero, scrive, segnala un item.
124 static void ring_push(ring_t *r, const record_t *rec) {
125     sem_wait(&r->sem_slots);              // attendo spazio
126     pthread_mutex_lock(&r->mtx);
127     r->buf[r->tail] = *rec;
128     r->tail = (r->tail + 1) % QUEUE_CAP;
129     pthread_mutex_unlock(&r->mtx);
130     sem_post(&r->sem_items);              // segnalo nuovo elemento
131 }
132 
133 // POP (bloccante): aspetta un item, legge, libera uno slot.
134 static void ring_pop(ring_t *r, record_t *out) {
135     sem_wait(&r->sem_items);              // attendo elemento
136     pthread_mutex_lock(&r->mtx);
137     *out = r->buf[r->head];
138     r->head = (r->head + 1) % QUEUE_CAP;
139     pthread_mutex_unlock(&r->mtx);
140     sem_post(&r->sem_slots);              // segnalo slot libero
141 }
142 
143 // ================== Operazioni sul record finale ==================
144 static void final_put(final_slot_t *f, const record_t *rec) {
145     sem_wait(&f->sem_empty);              // attendo che sia vuoto
146     pthread_mutex_lock(&f->mtx);
147     f->slot = *rec;
148     pthread_mutex_unlock(&f->mtx);
149     sem_post(&f->sem_full);               // segnalo che è pieno
150 }
151 
152 static void final_get(final_slot_t *f, record_t *out) {
153     sem_wait(&f->sem_full);               // attendo che sia pieno
154     pthread_mutex_lock(&f->mtx);
155     *out = f->slot;
156     pthread_mutex_unlock(&f->mtx);
157     sem_post(&f->sem_empty);              // segnalo che è tornato vuoto
158 }
159 
```

```c
// ================== Logica “equisomma” ==================
static inline void sums_even_odd(const uint8_t v[M], unsigned *sum_even, unsigned *sum_odd) {
    unsigned se = 0, so = 0;
    for (int i = 0; i < M; ++i) {
        if ((i & 1) == 0) se += v[i]; else so += v[i]; // indici: 0,2,4,... pari
    }
    *sum_even = se; *sum_odd = so;
}

// ================== THREAD: LETTORE (produttore) ==================
// - mappa il file con mmap
// - per ogni blocco di 12 byte, stampa il candidato e lo inserisce in coda
// - quando *l’ultimo lettore* termina, inserisce NUM_VERIF poison pill in coda
static void *reader_main(void *arg) {
    thread_arg_t *A = (thread_arg_t*)arg;
    shared_t *S = A->S;
    int idx = A->idx;
    const char *fname = S->filenames[idx - 1];

    lock_stdout(); printf("[READER-%d] file '%s'\n", idx, fname); unlock_stdout();

    int fd = open(fname, O_RDONLY);
    struct stat st;
    fstat(fd, &st);

    size_t sz = (size_t)st.st_size;
    size_t nrec = sz / M;
    void *map = NULL;

    if (sz > 0) {
        map = mmap(NULL, sz, PROT_READ, MAP_PRIVATE, fd, 0);
        if (map == MAP_FAILED) {
            perror("mmap");
        }
    }

    size_t produced = 0;
    for (size_t i = 0; i < nrec; ++i) {
        record_t rec = { .is_end = 0 };
        memcpy(rec.v, (uint8_t*)map + i * M, M);

        lock_stdout();
        printf("[READER-%d] vettore candidato n.%zu: ", idx, i + 1);

        print_vec12(rec.v);
        printf("\n");
        unlock_stdout();

        ring_push(&S->ring, &rec);
        produced++;
    }

    if (map && map != MAP_FAILED) munmap(map, sz);
```

```
213     close(fd);
214
215     lock_stdout(); printf("[READER-%d] terminazione con %zu vettori letti\n", idx,
    produced); unlock_stdout();
216
217     // Se questo è l'ULTIMO lettore a finire, metti NUM_VERIF poison pill in coda.
218     pthread_mutex_lock(&S->mtx_counts);
219     S->readers_left--;
220     int last = (S->readers_left == 0);
221     pthread_mutex_unlock(&S->mtx_counts);
222
223     if (last) {
224         record_t poison = { .is_end = 1 };
225         for (int k = 0; k < NUM_VERIF; ++k) ring_push(&S->ring, &poison);
226     }
227     return NULL;
228 }
229
230 // ================== THREAD: VERIFICATORE (consumatore) ==================
231 // - preleva dalla coda (bloccante)
232 // - se sentinella → esce dal ciclo
233 // - altrimenti verifica equisomma; se sì, invia il vettore allo slot finale
234 // - l'ULTIMO verificatore che termina inserisce una poison nello slot finale
235 static void *verifier_main(void *arg) {
236     thread_arg_t *A = (thread_arg_t*)arg;
237     shared_t *S = A->S;
238     int idx = A->idx;
239
240     size_t verified = 0;
241
242     for (;;) {
243         record_t rec;
244         ring_pop(&S->ring, &rec); // bloccante
245
246         if (rec.is_end) {
247             // Fine lavori per questo verificatore
248             break;
249         }
250
251         verified++;
252
253         unsigned se, so;
254         sums_even_odd(rec.v, &se, &so);
255
256         // Log verifica in una singola sezione protetta, ma con una sola printf finale
257         lock_stdout();
258         if (se == so) {
259             printf("[VERIF-%d] verifico vettore: ", idx);
260             print_vec12(rec.v);
261             printf("\n[VERIF-%d] si tratta di un vettore equisomma con somma %u!\n", idx,
    se);
262         } else {
263             printf("[VERIF-%d] verifico vettore: ", idx);
264             print_vec12(rec.v);
```

```
265             printf("\n[VERIF-%d] non è un vettore equisomma (somma pari %u vs. dispari %u)\n",
266                    idx, se, so);
267         }
268         unlock_stdout();
269 
270         if (se == so) {
271             // Passa al record finale (bloccante se lo slot è occupato)
272             record_t out = rec; // is_end=0
273             final_put(&S->out, &out);
274         }
275     }
276 
277     // Segno che questo verificatore ha finito; se è l'ULTIMO, sveglio il main
278     // inserendo una poison nello slot finale (così il main smette di aspettare).
279     int last = 0;
280     pthread_mutex_lock(&S->mtx_counts);
281     S->verifiers_left--;
282     last = (S->verifiers_left == 0);
283     pthread_mutex_unlock(&S->mtx_counts);
284 
285     lock_stdout();
286     printf("[VERIF-%d] terminazione con %zu vettori verificati\n", idx, verified);
287     unlock_stdout();
288 
289     if (last) {
290         record_t poison = { .is_end = 1 }; // segnale di chiusura per il main
291         final_put(&S->out, &poison);
292     }
293     return NULL;
294 }
295 
296 // ================== MAIN (stampa i vettori equisomma) ==================
297 int main(int argc, char **argv) {
298     if (argc < 2) {
299         fprintf(stderr, "Uso: %s <file-bin-1> [file-bin-2 ... file-bin-N]\n", argv[0]);
300         return 1;
301     }
302     setvbuf(stdout, NULL, _IONBF, 0);
303 
304     shared_t S;
305     ring_init(&S.ring);
306     final_init(&S.out);
307     pthread_mutex_init(&S.mtx_counts, NULL);
308 
309     S.n_readers = argc - 1;
310     S.filenames = &argv[1];
311     S.readers_left = S.n_readers;
312     S.verifiers_left = NUM_VERIF;
313     S.main_equisum_total = 0;
314 
315     pthread_t *readers = calloc(S.n_readers, sizeof(pthread_t));
316     pthread_t verifiers[NUM_VERIF];
317 
```

```
318     lock_stdout();
319     printf("[MAIN] creazione di %d thread lettori e %d thread verificatori\n",
320            S.n_readers, NUM_VERIF);
321     unlock_stdout();
322 
323     // Avvia lettori
324     for (int i = 0; i < S.n_readers; ++i) {
325         thread_arg_t *arg = malloc(sizeof(thread_arg_t));
326         arg->S = &S; arg->idx = i + 1;
327         int rc = pthread_create(&readers[i], NULL, reader_main, arg);
328         if (rc != 0) {
329             lock_stdout(); printf("[MAIN] ERRORE creazione READER-%d: %s\n", i + 1,
    strerror(rc)); unlock_stdout();
330             free(arg);
331             // In caso di errore, consideriamo quel lettore "già terminato"
332             pthread_mutex_lock(&S.mtx_counts);
333             S.readers_left--;
334             int last = (S.readers_left == 0);
335             pthread_mutex_unlock(&S.mtx_counts);
336             if (last) {
337                 record_t poison = { .is_end = 1 };
338                 for (int k = 0; k < NUM_VERIF; ++k) ring_push(&S.ring, &poison);
339             }
340         }
341     }
342 
343     // Avvia verificatori
344     for (int i = 0; i < NUM_VERIF; ++i) {
345         thread_arg_t *arg = malloc(sizeof(thread_arg_t));
346         arg->S = &S; arg->idx = i + 1;
347         int rc = pthread_create(&verifiers[i], NULL, verifier_main, arg);
348         if (rc != 0) {
349             lock_stdout(); printf("[MAIN] ERRORE creazione VERIF-%d: %s\n", i + 1,
    strerror(rc)); unlock_stdout();
350             free(arg);
351             // Se fallisce la creazione, riduciamo il numero atteso di verificatori.
352             pthread_mutex_lock(&S.mtx_counts);
353             S.verifiers_left--;
354             int last = (S.verifiers_left == 0);
355             pthread_mutex_unlock(&S.mtx_counts);
356             if (last) {
357                 // Nessun verificatore esiste: sblocco il main con poison sullo slot
    finale
358                 record_t poison = { .is_end = 1 };
359                 final_put(&S.out, &poison);
360             }
361         }
362     }
363 
364     // MAIN: consuma i vettori equisomma e li stampa finché arriva la poison dallo slot
    finale.
365     for (;;) {
366         record_t rec;
367         final_get(&S.out, &rec); // bloccante
```

```
368         if (rec.is_end) break;   // ultimo verificatore ha segnalato fine
369         S.main_equisum_total++;
370 
371         lock_stdout();
372         printf("[MAIN] ricevuto nuovo vettore equisomma: ");
373         print_vec12(rec.v);
374         printf("\n");
375         unlock_stdout();
376     }
377 
378     // Join di tutti i thread creati (i verificatori potrebbero essere già usciti)
379     for (int i = 0; i < S.n_readers; ++i) {
380         if (readers[i]) pthread_join(readers[i], NULL);
381     }
382     for (int i = 0; i < NUM_VERIF; ++i) {
383         pthread_join(verifiers[i], NULL);
384     }
385 
386     lock_stdout();
387     printf("[MAIN] terminazione con %lu vettori equisomma trovati\n",
    S.main_equisum_total);
388     unlock_stdout();
389 
390     // Cleanup
391     free(readers);
392     pthread_mutex_destroy(&S.mtx_counts);
393     ring_destroy(&S.ring);
394     final_destroy(&S.out);
395     return 0;
396 }
397 
```