Tutorato-Sistemi-Operativi-2025-main\Esame-[2025-09-12]\main_con_semafori.c

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

#define MAX_PATH 256
#define MAX_SIZE 10

typedef struct {
    char path[MAX_PATH];
    int occ;
}Record;

typedef struct {
    Record buffer[MAX_SIZE];
    int in;
    int out;
    int count;

    bool close;

    pthread_mutex_t mutex;
    sem_t sem_empty; // Conta gli slot liberi
    sem_t sem_full; // Conta gli elementi disponibili
}Shared_buffer;

void buffer_init(Shared_buffer *buff) {
    buff -> in = 0;
    buff -> out = 0;
    buff -> count = 0;
    buff -> close = false;

    pthread_mutex_init(&buff -> mutex, NULL);

    sem_init(&buff -> sem_empty, 0, MAX_SIZE);
    sem_init(&buff -> sem_full, 0, 0);
}

void buffer_close(Shared_buffer *buff) {
    pthread_mutex_lock(&buff -> mutex);

    buff -> close = true;

    pthread_mutex_unlock(&buff -> mutex);
```

```
52
53      sem_post(&buff -> sem_empty);
54      sem_post(&buff -> sem_full);
55  }
56
57  void buffer_destroy(Shared_buffer *buff) {
58      pthread_mutex_destroy(&buff->mutex);
59      sem_destroy(&buff->sem_empty);
60      sem_destroy(&buff->sem_full);
61  }
62
63  void buffer_in(Shared_buffer *buff, Record r) {
64      // 1. Aspetto che sia spazio libero
65      sem_wait(&buff -> sem_empty);
66
67      pthread_mutex_lock(&buff -> mutex);
68
69      // Controllo se il buffer è chiuso
70      if(buff -> close) {
71          pthread_mutex_unlock(&buff -> mutex);
72          sem_post(&buff -> sem_empty); // Restituisco il gettone per non bloccare altri
73          return;
74      }
75
76      // 2. Sezione critica: Inserimento senza bisogno di while/wait
77      buff -> buffer[buff -> in] = r;
78      buff -> in = (buff -> in + 1) % MAX_SIZE;
79      buff -> count++;
80
81      pthread_mutex_unlock(&buff -> mutex);
82
83      // 3. Segnalo che c'è un nuovo dato disponibile (sveglai consumatore)
84      sem_post(&buff -> sem_full);
85
86  }
87
88  // Da completare
89  bool buffer_out(Shared_buffer *buff, Record *r) {
90      // 1. Aseptto che ci sia un dato
91      sem_wait(&buff -> sem_full);
92
93      pthread_mutex_lock(&buff -> mutex);
94
95      // Controllo chiusura: se Chiuso e Vuoto, finiamo.
96      if(buff -> count == 0 && buff -> close) {
97          pthread_mutex_unlock(&buff -> mutex);
98
99          // IMPORTANTE: "Passaparola". Se siamo in tanti consumatori e il buffer chiude,
100         // devo svegliare il prossimo che potrebbe essere bloccato da sem_wait.
101         sem_post(&buff -> sem_full);
102         return false;
103     }
104
105     // Sezione critica: Estrazione
```

```
106     *r = buff -> buffer[buff -> out];
107     buff -> out = (buff -> out + 1) % MAX_SIZE;
108     buff -> count--;
109 
110     pthread_mutex_unlock(&buff -> mutex);
111 
112     // 3. Segnalo che c'è uno slot libero (Sveglia produttore)
113     sem_post(&buff -> sem_empty);
114 
115     return true;
116 }
117 
118 /* ------------------------------------------------------------------------ */
119 
120 bool read_dir(DIR *dir, const char *directory, Record *r) {
121     struct dirent *entry;
122     struct stat st;
123 
124     char path[MAX_PATH];
125 
126     while((entry = readdir(dir)) != NULL) {
127         if(!strcmp(entry -> d_name, ".") || !strcmp(entry -> d_name, "..")) {
128             continue;
129         }
130 
131         snprintf(path, sizeof(path), "%s/%s", directory, entry -> d_name);
132 
133         if(stat(path, &st) == 0 && S_ISREG(st.st_mode)) {
134             strncpy(r -> path, path, MAX_PATH);
135             r->path[MAX_PATH - 1] = '\0';
136             return true;
137         }
138     }
139 
140     return false;
141 }
142 
143 int check_parola(const char *filename, const char *word) {
144     FILE *file = fopen(filename, "r");
145 
146     if(!file) {
147         perror("Errore apertura file \n");
148         return -1;
149     }
150 
151     char buffer[100];
152     int count = 0;
153     int len = strlen(word);
154 
155     while(fscanf(file, "%255s", buffer) == 1) {
156         if(strcasecmp(buffer, word) == 0) {
157             count ++;
158         }
159     }
```

```
160
161     fclose(file);
162     return count;
163 }
164
165 /* ------------------------------------------------------------------------------ */
166
167 /*-------------- PRODUCER --------------*/
168 typedef struct {
169     Shared_buffer *proposte;
170     const char *directory;
171 }prod_args;
172
173 void *producer(void *arg) {
174     prod_args *args = (prod_args *)arg;
175     Record r;
176     r.occ = 0;
177
178     DIR *dir = opendir(args -> directory);
179     if(!dir) {
180         perror("Errore apertura directory...\n");
181         pthread_exit(NULL);
182     }
183
184     while(read_dir(dir, args -> directory, &r)) {
185         buffer_in(args -> proposte, r);
186     }
187
188     closedir(dir);
189     pthread_exit(NULL);
190 }
191
192 /*-------------- VERIFIER --------------*/
193
194 typedef struct {
195     Shared_buffer *proposte;
196     Shared_buffer *proposte_out;
197     const char *word;
198 }ver_args;
199
200 void *verifier(void *arg) {
201     ver_args *args = (ver_args *)arg;
202     Record r;
203
204     while(buffer_out(args -> proposte, &r)) {
205
206         int occorrenze = check_parola(r.path, args -> word);
207
208         if(occorrenze > 0) {
209             r.occ = occorrenze;
210             buffer_in(args -> proposte_out, r);
211         }
212     }
213
```

```
214     pthread_exit(NULL);
215 }
216 
217 /*-------------- CONSUMER --------------*/
218 // ...
219 
220 
221 int main(int argc, char *argv[]) {
222 
223     if(argc < 3) {
224         printf("Uso: <word> <dir1> <dir2> ... <dir-n> ", argv[0]);
225         return 1;
226     }
227 
228     const char *word_to_search = argv[1];
229     int N = argc - 2;
230 
231     Shared_buffer proposte, proposte_out;
232     buffer_init(&proposte);
233     buffer_init(&proposte_out);
234 
235     pthread_t thread_producers[N];
236     pthread_t thread_verifier;
237 
238     prod_args prod_r[N];
239 
240     for(int i = 0; i < N; ++i) {
241         prod_r[i].proposte = &proposte;
242         prod_r[i].directory = argv[i + 2];
243 
244         if (pthread_create(&thread_producers[i], NULL, producer, &prod_r[i]) != 0) {
245             perror("Errore creazione thread");
246             return 1;
247         }
248     }
249 
250     ver_args ver_r = {
251         .proposte = &proposte,
252         .proposte_out = &proposte_out,
253         .word = word_to_search
254     };
255 
256     pthread_create(&thread_verifier, NULL, verifier, &ver_r);
257 
258     for(int i = 0; i < N; ++i) {
259         pthread_join(thread_producers[i], NULL);
260     }
261 
262     buffer_close(&proposte);
263 
264     pthread_join(thread_verifier, NULL);
265 
266     buffer_close(&proposte_out);
267 
```

```
    // printf("Numero di occorenze: %d\n", r.occ);
    Record r;
    printf("--- Risultati per la parola '%s' ---\n", word_to_search);
    int total_files = 0;

    while(buffer_out(&proposte_out, &r)) {
        printf("File: %s | Occorrenze: %d\n", r.path, r.occ);
        total_files++;
    }

    if(total_files == 0) {
        printf("Nessuna occorrenza trovata.\n");
    }

    buffer_destroy(&proposte);
    buffer_destroy(&proposte_out);
    return 0;
}

```