Tutorato-Sistemi-Operativi-2025-main\Esame-[2025-09-12]\find_word_condition.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>

#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;
    pthread_cond_t not_empty;
    pthread_cond_t not_full;
}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);
    pthread_cond_init(&buff -> not_empty, NULL);
    pthread_cond_init(&buff -> not_full, NULL);
}

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

    buff -> close = true;

    pthread_cond_broadcast(&buff -> not_empty);
    pthread_cond_broadcast(&buff -> not_full);
    pthread_mutex_unlock(&buff -> mutex);
```

```
52  }
53
54  void buffer_destroy(Shared_buffer *buff) {
55      pthread_mutex_destroy(&buff->mutex);
56      pthread_cond_destroy(&buff->not_empty);
57      pthread_cond_destroy(&buff->not_full);
58  }
59
60  void buffer_in(Shared_buffer *buff, Record r) {
61      pthread_mutex_lock(&buff -> mutex);
62
63      while(buff -> count == MAX_SIZE && !buff -> close) {
64          pthread_cond_wait(&buff -> not_full, &buff -> mutex);
65      }
66      if(buff -> close) {
67          pthread_mutex_unlock(&buff -> mutex);
68          return;
69      }
70
71      buff -> buffer[buff -> in] = r;
72      buff -> in = (buff -> in + 1) % MAX_SIZE;
73      buff -> count++;
74
75      pthread_cond_signal(&buff -> not_empty);
76      pthread_mutex_unlock(&buff -> mutex);
77
78  }
79
80  bool buffer_out(Shared_buffer *buff, Record *r) {
81      pthread_mutex_lock(&buff -> mutex);
82
83      while(buff -> count == 0 && !buff -> close) {
84          pthread_cond_wait(&buff -> not_empty, &buff -> mutex);
85      }
86
87      if(buff -> count == 0 && buff -> close) {
88          pthread_mutex_unlock(&buff -> mutex);
89          return false;
90      }
91
92      *r = buff -> buffer[buff -> out];
93      buff -> out = (buff -> out + 1) % MAX_SIZE;
94      buff -> count--;
95
96      pthread_cond_signal(&buff -> not_full);
97      pthread_mutex_unlock(&buff -> mutex);
98
99      return true;
100 }
101
102 /* ------------------------------------------------------------------------ */
103
104 bool read_dir(DIR *dir, const char *directory, Record *r) {
105     struct dirent *entry;
```

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

```
160     r.occ = 0;
161
162     DIR *dir = opendir(args -> directory);
163     if(!dir) {
164         perror("Errore apertura directory...\n");
165         pthread_exit(NULL);
166     }
167
168     while(read_dir(dir, args -> directory, &r)) {
169         buffer_in(args -> proposte, r);
170     }
171
172     closedir(dir);
173     pthread_exit(NULL);
174 }
175
176 /*-------------- VERIFIER --------------*/
177
178 typedef struct {
179     Shared_buffer *proposte;
180     Shared_buffer *proposte_out;
181     const char *word;
182 }ver_args;
183
184 void *verifier(void *arg) {
185     ver_args *args = (ver_args *)arg;
186     Record r;
187
188     while(buffer_out(args -> proposte, &r)) {
189
190         int occorrenze = check_parola(r.path, args -> word);
191
192         if(occorrenze > 0) {
193             r.occ = occorrenze;
194             buffer_in(args -> proposte_out, r);
195         }
196     }
197
198     pthread_exit(NULL);
199 }
200
201 /*-------------- CONSUMER --------------*/
202 // ...
203
204
205 int main(int argc, char *argv[]) {
206
207     if(argc < 3) {
208         printf("Uso: <word> <dir1> <dir2> ... <dir-n> ", argv[0]);
209         return 1;
210     }
211
212     const char *word_to_search = argv[1];
213     int N = argc - 2;
```

```
214
215     Shared_buffer proposte, proposte_out;
216     buffer_init(&proposte);
217     buffer_init(&proposte_out);
218
219     pthread_t thread_producers[N];
220     pthread_t thread_verifier;
221
222     prod_args prod_r[N];
223
224     for(int i = 0; i < N; ++i) {
225         prod_r[i].proposte = &proposte;
226         prod_r[i].directory = argv[i + 2];
227
228         if (pthread_create(&thread_producers[i], NULL, producer, &prod_r[i]) != 0) {
229             perror("Errore creazione thread");
230             return 1;
231         }
232     }
233
234     ver_args ver_r = {
235         .proposte = &proposte,
236         .proposte_out = &proposte_out,
237         .word = word_to_search
238     };
239
240     pthread_create(&thread_verifier, NULL, verifier, &ver_r);
241
242     for(int i = 0; i < N; ++i) {
243         pthread_join(thread_producers[i], NULL);
244     }
245
246     buffer_close(&proposte);
247
248     pthread_join(thread_verifier, NULL);
249
250     buffer_close(&proposte_out);
251
252     // printf("Numero di occorenze: %d\n", r.occ);
253     Record r;
254     printf("--- Risultati per la parola '%s' ---\n", word_to_search);
255     int total_files = 0;
256
257     while(buffer_out(&proposte_out, &r)) {
258         printf("File: %s | Occorrenze: %d\n", r.path, r.occ);
259         total_files++;
260     }
261
262     if(total_files == 0) {
263         printf("Nessuna occorrenza trovata.\n");
264     }
265
266     buffer_destroy(&proposte);
267     buffer_destroy(&proposte_out);
```

```
    return 0;
}

```