

Esercitazione sui segnali

Problema: creare un programma analizzatore di file testuali che prenda come argomenti il nome di un file e una sequenza di stringhe. Per ognuna delle stringhe deve venire generato un thread che conta il numero di occorrenze della stringa all'interno del file. Predisporre una serie di segnali (o messaggi-evento) per effettuare almeno le seguenti operazioni:

Sospensione regolare dell'analisi: se viene ricevuto un segnale di sospensione l'analizzatore stampa su un file la situazione provvisoria dell'analisi oltre all'informazione che il processo e' attualmente sospeso.

Terminazione regolare dell'analisi: se viene ricevuto un segnale di terminazione il processo stampa su un file la situazione dell'analisi al ricevimento del segnale, oltre all'informazione che il processo e' terminato prima del dovuto.

Struttura generale del codice Unix

```
void output_result ();
void output_final_result (int complete);
void output_partial_result();
void suspend();
void terminate ();
void lower_priority ();
void * VerifyFile(void * info);

int main (int argc, char *argv[])
{
    /* Dichiarazione di variabili */

    signal (SIGTERM, terminate);
    signal (SIGINT, terminate);
    signal (SIGTSTP, suspend);
    signal (SIGHUP, terminate);
    signal (SIGUSR1, output_partial_result);
    signal (SIGXCPU, lower_priority);

    /* codice */
}
```

Codice del main()

```
if (argc < 3) {
    printf("Usage: analyzer file first_string [other_strings ...]\n");
    exit(-1);
}

active_threads = malloc (sizeof(pthread_t) * (argc - 2));
partial_results = malloc (sizeof(int) * (argc - 2));

file_to_check = argv[1];
number_strings_to_check = argc - 2;
strings_to_check = &argv[2];

for (i=0; i<number_strings_to_check; i++) {
    thread_info = malloc (sizeof(struct _thread_info));
    thread_info->position = i;
    thread_info->string = argv[i+2];
    j=pthread_create(&active_threads[i], NULL, VerifyFile, (void *)thread_info);
    if (j) {
        printf("cannot create thread for error %d\n", j);
        exit(-1);
    }
}

for (i=0; i<(argc-2); i++) pthread_join(active_threads[i], &status);
output_final_result(1);

exit(0);
```

Codice del VerifyFile() - I

```
void * VerifyFile(void * info)
{
    int fd;
    int i,j;
    char buffer[2*READ_BUFFER];
    struct _thread_info * thread_info;
    int string_length = 0;

    thread_info = (struct _thread_info *) info;
    fd = open(file_to_check, O_RDONLY);
    //printf("Ordering thread active\n");
    //
    if (fd < 0) {
        printf("Can't open file %s\n",file_to_check);
        partial_results[thread_info->position] = -1;
        status = -1;
        pthread_exit((void *)&status);
    }

    string_length = strlen (thread_info->string);
    if (string_length > READ_BUFFER) {
        printf("String %s too long\n",thread_info->string);
        partial_results[thread_info->position] = -1;
        status = -1;
        pthread_exit((void *)&status);
    }
}
```

Codice del VerifyFile() - II

```
i = READ_BUFFER;
j = read(fd, &buffer[READ_BUFFER], READ_BUFFER);
while (j != 0) {

    for (; i < ((READ_BUFFER + j) - string_length + 1); i++) {
        if (strncmp(thread_info->string, &buffer[i], string_length) == 0) {
            partial_results[thread_info->position]++;
            i = i + string_length - 1;
        }
    }

    memcpy(&buffer[READ_BUFFER - string_length + 1],
           &buffer[(READ_BUFFER + j) - string_length + 1], (READ_BUFFER + j) - i);
    i = READ_BUFFER - string_length + 1;

    j = read(fd, &buffer[READ_BUFFER], READ_BUFFER);
}

status = 0;
pthread_exit((void *)&status);
}
```

Codice dei segnali semplici

```
void suspend() {  
    output_partial_result();  
    kill(getpid(), SIGSTOP);  
    return;  
}
```

```
void terminate () {  
    output_final_result(0);  
    exit(0);  
}
```

```
void lower_priority () {  
    nice(10);  
    return;  
}
```

Codice di stampa risultati

```
void output_final_result (int complete) {

    output_file = fopen(dump_file, "w");
    if (output_file == NULL) {
        printf("Cannot open dump file!\n");
        exit(-1);
    }

    if (complete) fprintf(output_file, "Analysis complete\nOccurrences:\n");
    else fprintf(output_file, "Analysis Interrupted\nAt the
                        interruption time I found the following occurrences:\n");

    output_result();
    fclose(output_file);
    return;
}

void output_partial_result() {

    output_file = fopen(dump_file, "w");
    if (output_file == NULL) {
        printf("Cannot open dump file!\n");
        exit(-1);
    }

    fprintf(output_file, "Analysis still ongoing\nAt this time I already found
                        the following occurrences:\n");

    output_result();
    fclose(output_file);
    return;
}
```

Versione per Windows

```
void output_result ();
void output_final_result (int complete);
void output_partial_result();
LRESULT CALLBACK WndProc (HWND hWnd, UINT message,
                           WPARAM wParam, LPARAM lParam);
DWORD WINAPI VerifyFile(void * info);
DWORD WINAPI Listener(void * nothing);

void main (int argc, char *argv[])
{
    /* Registrazione messaggi */

    /* Attivazione handler messaggi */

    /* Attivazione thread di ascolto dei messaggi */

    /* Attesa di terminazione dei thread di lavoro */

    output_final_result(1);

    ExitProcess(0);
}
```

Registrazione tipi di messaggi

```
term_mex_type = RegisterWindowMessage(termination_message);
if (!term_mex_type) {
    printf("Can't create term message for error %d\n",
        GetLastError());
    fflush(stdout);
    ExitProcess(-1);
}
```

```
print_mex_type = RegisterWindowMessage(print_message);
if (!print_mex_type) {
    printf("Can't create print message for error %d\n",
        GetLastError());
    fflush(stdout);
    ExitProcess(-1);
}
```

Attivazione handler e ascolto

```
RegisterClass();
CreateWindow ();

for (i=0; i<number_strings_to_check; i++) {

    CreateThread (NULL, 0, VerifyFile, (void *)thread_info, 0, &tid))
}

CreateThread (NULL, 0, Listener, NULL, 0, &tid)

        DWORD WINAPI Listener(void * nothing) {
            MSG msg;

            while (GetMessage(&msg, NULL, 0,0)) {
                TranslateMessage(&msg);
                DispatchMessage(&msg);
            }
            return(0);
        }
```