Introduzione alla programmazione multiprocesso e multithread in ambiente Unix attraverso algoritmi in C

Pre-requisiti per questo tutorial:

- Basi del linguaggio C
- Familiarità con i sistemi Unix

Obiettivi di questo tutorial:

- Multiprogrammazione in ambiente Unix
- Differenza tra thread e processo
- Gestione dei processi a basso livello da parte del kernel Unix (algoritmi di scheduling)

Introduzione ai sistemi multiprogrammati

Ciò che ha fatto la fortuna dei sistemi Unix già negli anni '80 è la propensione di questi ultimi nei confronti della programmazione multiprocesso e, in seguito, anche della programmazione multithread.

Per capire cosa voglia dire la programmazione multiprocesso, faccio un esempio. I sistemi DOS degli anni '80 erano sistemi monoprogrammati, ovvero consentivano alla CPU di eseguire un solo processo per volta. Se quindi volevo effettuare, ad esempio, la copia di una directory in un'altra su uno di questi sistemi, il processo di copia occupava interamente le risorse della CPU, e fino alla terminazione di questo processo all'utente era impossibile avviare nuovi processi.

I sistemi Unix invece hanno adottato vari stratagemmi per consentire, anche ai sistemi con una sola CPU, di eseguire più task in maniera concorrente. Questo si riduce, a livello di sistema, a una segmentazione della memoria in più processi. La CPU può sempre eseguire un solo processo per volta, nel caso dei sistemi monoprocessore, ma i sistemi Unix sono programmati in modo tale che il processo che in un dato momento occupa la CPU non “congeli” le risorse del sistema fino alla sua terminazione, adottando politiche di schedulng (a “rotazione” di processi).

Algoritmi di scheduling

Nel corso degli anni gli algoritmi di scheduling si sono sempre più evoluti, cercando di evitare da una parte problemi quali l'occupazione prolungata della CPU da parte di un solo processo e, dal lato opposto, problemi di *starvation* (ovvero il congelamento di un processo che, a causa di politiche errate nell'algoritmo di scheduling, quali un'errata gestione della priorità dei processi, non acquisterà mai una priorità sufficiente per essere eseguito, rimanendo per sempre in attesa).

L'algoritmo di scheduling più elementare è quello *round-robin*, un algoritmo che suddivide il tempo della CPU in quanti uguali. Esempio: ho 3 processi in esecuzione, con i seguenti tempi:

task1 = 250ms
task2 = 150ms
task3 = 200ms

con una politica round-robin che, mettiamo, suddivide il tempo di utilizzo della CPU in 50ms per ogni task, avrò:

1. task1 occupa la CPU per 50 ms (ovvero, passa dallo status READY in cui si trova prima di andare in esecuzione allo status RUNNING per 50ms, per poi essere fermato dal sistema operativo, passando in status SLEEPING e, dopo un certo intervallo, nuovamente in status READY)
2. task2 occupa la CPU per 50 ms
3. task3 occupa la CPU per 50 ms
4. task1 occupa la CPU per 50 ms
5. task2 occupa la CPU per 50 ms
6. task3 occupa la CPU per 50 ms
7. task1 occupa la CPU per 50 ms
8. task2 occupa la CPU per 50 ms (a questo punto il codice da eseguire all'interno di task2 è terminato)
9. task3 occupa la CPU per 50 ms
10. task1 occupa la CPU per 50 ms
11. task3 occupa la CPU per 50 ms (a questo punto il codice da eseguire all'interno di task3 è terminato)
12. task1 occupa la CPU per 50 ms (a questo punto anche il codice da eseguire all'interno di task3 è terminato)

Quest'algoritmo è semplice da implementare a livello di kernel ed evita problemi di starvation, in quanto non ha una politica predefinita per le priorità di un task. Tuttavia, attraverso quest'algoritmo più il task è grande (ovvero maggiore è il suo tempo di esecuzione), più viene premiato, in quanto può occupare la CPU per un periodo cumulativo di tempo maggiore dei task più piccoli.

Gli algoritmi round-robin, nelle varianti *weighted round-robin* e *deficit round-robin*, vengono anche utilizzati per lo scheduling dei pacchetti provenienti da connessioni multiple. Ad esempio, se il sistema riceve dei pacchetti da *n* fonti *f1,f2,...,fn*, è possibile attraverso algoritmi di questo tipo stabilire per quanto tempo ogni fonte è autorizzata a inviare pacchetti al sistema.

L'altra grande classe di algoritmi di scheduling, ideata per evitare i problemi degli algoritmi RR, sono gli *algoritmi a priorità*, ideati per evitare che i task che richiedono un tempo di esecuzione maggiore vengano maggiormente premiati, come negli algoritmi RR. Gli algoritmi di questo tipo si dividono a loro volta in

- **Algoritmi a priorità statica**. In questi algoritmi la priorità di un task viene stabilita all'atto della sua creazione, in base alle sue caratteristiche
- **Algoritmi a priorità dinamica**. In questi algoritmi la priorità di un task può variare

durante l'esecuzione. Questo è utile per i seguenti motivi:
- Per penalizzare i task che impegnano troppo la CPU
- Per evitare problemi di starvation (ovvero per evitare che nella coda di esecuzione dei processi non riescano mai ad andare in esecuzione)
- Per aumentare la priorità di un processo in base al suo tempo di attesa nella coda

Programmazione multiprocesso

Un sistema Unix è fortemente improntato sulla programmazione multiprocesso. In particolare, quando un sistema Unix viene avviato viene anche generato un processo, chiamato *init*, con la priorità massima. Questo processo è alla base di tutti i processi che vengono successivamente generati all'interno del sistema. Le shell altro non sono che processi figli del processo *init*, la procedura di autenticazione attraverso username e password è a sua volta gestita da altri due processi, generalmente generati dalla shell stessa, i processi *login* e *getty*. E ancora, ogni eseguibile avviato nel sistema non fa altro che generare un nuovo processo all'interno della shell che lo ha richiamato, e a sua volta l'eseguibile stesso può generare altri processi (vedremo presto come farlo).

Un processo eredita dal processo che lo ha richiamato l'area dati (ovvero le variabili presenti all'interno dello stack dell'eseguibile prima che venisse generato il processo figlio). Ma, una volta che viene mandato in esecuzione, ha una propria area dati (questo vuol dire che le modifiche attuate all'interno della propria area dati non modificano i dati all'interno del processo padre) e, ovviamente, una propria area di codice, che contiene il codice che il processo deve eseguire.

Sui sistemi Unix per generare un nuovo processo si utilizza la primitiva ***fork()***. Questa primitiva fa le operazioni appena descritte sopra, ovvero genera un nuovo processo, con un proprio PID (Process ID, ovvero un numero che identifica il processo) e copia all'interno della sua area dati l'area dati del processo padre. La primitiva fork() ritorna

- **0** nel caso del processo figlio (quindi, se il risultato della fork() è 0 so che lì ci devo andare a scrivere il codice che verrà eseguito dal processo figlio)
- un valore **> 0** nel caso del processo padre
- **-1** se c'è stato un errore (ad esempio, se la tabella dei processi è piena, se non ho abbastanza spazio in memoria per allocare un nuovo processo, se non ho i diritti per creare nuovi processi)

Facciamo un primo esempio di programma concorrente multiprocesso in C:

```
#include <stdio.h>
#include <unistd.h>

main()  {
     int pid;
     int status;
```

```
printf (“Sono il processo padre, il mio PID è %d\n”,
getpid());


// Genero un nuovo processo
pid=fork();


if (pid==-1)  {
     // ERRORE! Non è stato possibile creare il nuovo
     processo


     printf (“Impossibile creare un nuovo processo\n”);
     exit(1);
}


if (pid==0)  {
     // In questo caso la fork() ha ritornato 0, quindi qui
     ci scrivo il codice del figlio


     printf (“Sono il processo figlio di %d, il mio PID è
     %d\n”, getppid(), getpid());
     exit(0);
}


if (pid>0)  {
     // In questo caso la fork() ha ritornato un valore
     maggiore di 0, quindi qui scrivo il codice del processo
     padre


     printf (“Sono il processo padre e ho generato un
     processo figlio\n”);


     // Attendo che il processo figlio venga terminato, e
     salvo il suo valore di ritorno nella variabile status
     while ((pid=wait(&status)>0);


     // Dal valore di status ricavo il valore di ritorno del
     processo figlio
```

```
            status = (status & 0xFF) >> 8;


            printf (“Il processo %d è terminato con status %d\n”,
            pid, status);
      }
}
```

Un paio di commenti. Innanzitutto, un processo può conoscere in ogni momento il suo PID e il PID del processo che lo ha generato rispettivamente attraverso le primitive *getpid()* e *getppid()* (GET Parent PID). Lo studio dei valori di ritorno della primitiva fork() è già stato fatto precedentemente e commentato nel codice, quindi non sto qui a discuterlo nuovamente.

È invece interessante l'uso della primitiva *wait()*, utilizzata all'interno del codice. Questa primitiva mette il processo padre in attesa finché tutti i processi figli non vengono terminati, ritorna -1 se non ci sono processi figli da attendere o un valore > 0 che rappresenta il PID del processo figlio appena terminato. Come parametro prende invece un puntatore a una variabile int. Su questa variabile viene scritto lo status con cui è terminato il processo figlio (attraverso un return o la primitiva exit()) nel seguente formato (in esadecimale):

0xSS00

dove *SS* rappresenta lo status (in esadecimale) con cui il programma è terminato (nel nostro caso 0), e le ultime due cifre sono 2 zeri. Questo nel caso in cui il processo è terminato in modo “naturale”. Se invece dovesse essere terminato in modo “innaturale”, ovvero tramite un segnale da parte del padre, gli zeri e il valore dello status verrebbero invertiti. Per ottenere il valore di ritorno devo quindi ricorrere a uno stratagemma a “basso livello”. Faccio un AND tra la mia variabile e il numero esadecimale 0xFF00 (in binario 1111 1111 0000 0000), in modo da azzerare eventuali valori diversi da zero nelle due cifre esadecimali meno significative, quindi faccio uno shift a destra di 1 byte del valore attuale, in modo da ritrovarmi con un valore del tipo 0x00SS, che rappresenta lo status autentico ritornato dal processo figlio. La riga

```
while ((pid=wait(&status)>0);
```

dice quindi al processo padre “finché la primitva wait() ritorna un valore maggiore di zero, ovvero finché ci sono processi da attendere, salva questo valore nella variabile pid, quindi salva il valore dello status nella variabile status”.

Nel caso il valore di ritorno dei miei processi non mi interessi più di tanto, posso scrivere

```
while ((pid=wait((int*) 0)>0);
```

Vediamo ora un piccolo esempio di programma al quale vengono passati due argomenti, rappresentanti due nomi di file, e che genera due processi figli, ognuno dei quali legge

un carattere dal file ad esso associato e lo riporta su standard output. Ogni processo figlio ritorna al padre il numero di caratteri letti all'interno del file.

(N.B.: in questo esempio ho utilizzato le primitive a basso livello del kernel Unix, ovvero open, read, write e close, per l'apertura/lettura/scrittura/chiusura di un file, e non le funzioni ad alto livello specificate in stdio.h, proprio perché voglio creare un programma ottimizzato al 100% per sistemi Unix, e suppongo che il lettore sia familiare con queste primitive. In caso contrario, si possono visionare le pagine di manuale Unix associate, o la documentazione presente su internet)

```
#include <stdio.h>
#include <unistd.h>
#include <fcntl.h>

main(int argc, char **argv)  {
     // Descrittore dei file
     int fd;

     int i;
     int pid;
     int status;

     // Numero di caratteri letti
     int N=0;

     // Buffer in cui verrà salvato il carattere letto
     char buff[1];

     if (argc<3)  {
          printf (“Errore nel numero di argomenti passati\n”);
          exit(1);
     }

     // Creo i due processi
     for (i=0; i<2; i++)  {
          pid=fork();

          // Errore
          if (pid==-1)  {
```

```
            printf (“Errore nella creazione del processo
            figlio\n”);
            exit(2);
        }

        // Codice del figlio
        if (pid==0)  {
            // Provo ad aprire il file associato al processo
            if ((fd=open(argv[i+1], O_RDONLY)) < 0)  {
                printf (“Errore: impossibile leggere il file
                %s\n”, argv[i+1]);
                exit(3);
            }

            printf (“Contenuto del file %s:\n”, argv[i+1]);

            // Finché ci sono caratteri da leggere all'interno
            del file, li riporto su stdout
            while ( read(fd,buff,1) > 0 )  {
                printf (“%c”, buff[1]);
                N++;
            }

            close(fd);

            // Ritorno il numero di caratteri letti
            exit(N);
        }

        // Codice del processo padre
        if (pid>0)  {
            while((pid=wait(&status))>0)  {
                status = (status & 0xFF) >> 8;
                printf (“Il processo %d è terminato e ha letto
                %d caratteri dal file\n”, pid, status);
            }
        }
}
```

```
}
```

Comunicazione tra processi. Concetto di *pipe*

Due processi possono comunicare e scambiarsi dati tra loro attraverso un meccanismo di astrazione chiamato *pipe*. In italiano potremmo tradurlo come “tubo”, e questa parola rende molto bene l'idea. Una pipe è una struttura astratta per la comunicazione tra due o più processi che si può schematizzare proprio come una tubatura. Dal punto di vista logico-informatico è una struttura FIFO (First In First Out); questo vuol dire che, se un processo scrive sulla pipe e un altro legge i dati scritti, quest'ultimo legge i dati nell'ordine preciso in cui sono stati scritti. Dal punto di vista di sistema, una pipe viene descritta da un array di due interi: il primo valore dell'array identifica il canale di lettura della pipe, il secondo valore identifica quello di scrittura. Su un sistema Unix per inizializzare una pipe uso la primitiva *pipe()*, primitiva che ritorna un valore >=0 nel caso in cui la pipe è creata con successo, -1 in caso contrario, e prende come parametro l'identificatore della pipe. Piccolo esempio di utilizzo:

```
// Identifico il tipo pipe_t (pipe type) come un array di due
interi
typedef int pipe_t[2];


...


pipe_t pp;


if (pipe(pp)<0)  { Errore! }


// D'ora in avanti userò il canale pp[0] per leggere dalla pipe,
pp[1] per scrivere sulla pipe
```

Ovviamente, se provo a scrivere sul canale di lettura della pipe o viceversa ottengo un errore di broken pipe, in quanto sto tentando di eseguire un'operazione non consentita.

Vediamo ora un esempio più corposo, in cui un processo padre inizializza una pipe e crea un processo figlio. Il processo figlio prende da stdin una stringa, di lunghezza massima N, inserita dall'utente e la scrive sulla pipe. Il processo padre attende che il figlio termini e scrive la stringa su stdout, leggendola dal canale di lettura della pipe.

(N.B.: per leggere, scrivere o chiudere una pipe utilizzo sempre le primitive read, write e close, esattamente le stesse primitive che userei per un file o per un socket).

```
#include <stdio.h>
#include <string.h>
```

```
#include <unistd.h>

// Massima lunghezza dell'input inserito
#define   N    100

typedef int pipe_t[2];

main()  {
     pipe_t pp;
     char buff[N];

     // Creazione della pipe
     if (pipe(pp) < 0)  {
          printf ("Errore nella creazione della pipe\n");
          exit(1);
     }

     // Creazione di un processo figlio
     switch (fork())  {
          case -1:
               printf ("Impossibile creare un processo figlio\n");
               exit(2);
               break;

          // Codice del figlio
          case 0:
               // Chiudo il canale di lettura della pipe, in
               quanto il processo figlio deve solo scrivere sulla
               pipe e il canale di lettura non mi interessa
               close(pp[0]);

               // Chiedo all'utente di inserire una stringa
               printf ("Stringa da inviare sulla pipe: ");
               fgets(buff, N, stdin);
               buff[stlen(buff)]='\0';

               // Scrivo la stringa appena letta sulla pipe
```

```
                if (write(pp[1], buff, N) < 0)  {
                     printf (“Errore nella scrittura su pipe\n”);
                     exit(3);
                }

                close(pp[1]);
                exit(0);
                break;

          // Codice del padre
          default:
                // Chiudo il canale di scrittura dal lato del
                padre, dato che devo solo leggere dalla pipe
                close(pp[1]);

                printf ("Aspetto che il figlio venga
terminato...\n");
                wait( (int*) 0 );

                // Una volta che il figlio è terminato, leggo la
                stringa inserita dalla pipe
                if (read(pp[0], buff, N) < 0)  {
                     printf (“Errore nella lettura da pipe\n”);
                     exit(1);
                }

                printf ("Il processo figlio ha scritto %s sulla
pipe\n", buff);
                exit(0);
                break;
     }
}
```

È possibile anche effettuare la ridirezione di un certo canale su una pipe. Ricordiamo che in un sistema Unix stdin, stdout e stderr non sono altro che descrittore di file “speciali”, identificati rispettivamente dai valori 0, 1 e 2. Quindi posso chiudere uno di questi canali e ridirigere il traffico diretto da o verso uno di questi canali su una pipe, così come potrei fare la ridirezione su file. Esempio:

```
...
typedef int pipe_t[2];


pipe_t pp;


...


close(1); // Chiudo stdout
dup(pp[1]);    // Duplico il canale di scrittura della pipe, che
acquista il primo canale               disponibile. Poiché ho
appena chiuso il canale stdout, tutto il traffico
     diretto su stdout verrà ridiretto sulla pipe.
```

Questo meccanismo, che ora abbiamo visto implementato a basso livello, viene implementato ad alto livello dai comandi di pipe della shell. Ad esempio se do un comando del tipo `ps ax | grep init`, non fa altro che eseguire il comando ps. Il canale di output di questo comando viene chiuso, e viene ridiretto su una pipe costruita per la comunicazione tra ps e grep.

Interruzione di un processo. Concetto di segnale

Il processo padre può terminare, uccidere un processo figlio o imporgli l'esecuzione di un codice arbitrario in un certo momento attraverso il meccanismo dei segnali. Questa è la lista dei segnali standard su un sistema Unix, visibile anche dando il comando man 7 signal:

Signal	Value	Action	Comment
SIGHUP	1	Term	Hangup detected on controlling terminal or death of controlling process
SIGINT	2	Term	Interrupt from keyboard
SIGQUIT	3	Core	Quit from keyboard
SIGILL	4	Core	Illegal Instruction
SIGABRT	6	Core	Abort signal from abort(3)
SIGFPE	8	Core	Floating point exception
SIGKILL	9	Term	Kill signal
SIGSEGV	11	Core	Invalid memory reference
SIGPIPE	13	Term	Broken pipe: write to pipe with no readers

```
   SIGALRM      14        Term    Timer signal from alarm(2)
   SIGTERM      15        Term    Termination signal
   SIGUSR1   30,10,16     Term    User-defined signal 1
   SIGUSR2   31,12,17     Term    User-defined signal 2
   SIGCHLD   20,17,18     Ign     Child stopped or terminated
   SIGCONT   19,18,25     Cont    Continue if stopped
   SIGSTOP   17,19,23     Stop    Stop process
   SIGTSTP   18,20,24     Stop    Stop typed at tty
   SIGTTIN   21,21,26     Stop    tty input for background
                                   process
   SIGTTOU   22,22,27     Stop    tty output for background
                                   process
```

I segnali si installano con la primitiva *signal()* (standard SystemV, quello più utilizzato) o *sigset()* (standard BSD, meno utilizzato). Entrambe le primitive prendono come primo argomento il segnale associato (uno di quelli presenti nella lista), come secondo argomento una funzione di tipo void che prende un parametro di tipo int (che rappresenta il numero del segnale ricevuto); questa è la funzione che verrà richiamata quando viene lanciato un dato segnale. Il segnale viene invece "lanciato" con la primitiva *kill()*, una primitiva che prende come primo parametro il PID del processo che deve ricevere il segnale, come secondo il segnale da inviare. Esempio:

```
#include <stdio.h>
#include <signal.h>

void foo(int sig)  {
     printf ("Ricevuto segnale %d. Terminazione del
     processo...\n", sig);

     signal(SIGTERM,foo);
}

void do_nothing(int sig)  {
     signal(SIGUSR1,do_nothing);
}

main()  {
     int pid;

     // Installazione dei segnali
```

```
signal(SIGTERM,foo);
signal(SIGUSR1,do_nothing);

pid=fork();

switch(pid)  {
     case -1:
          printf (“Errore nella creazione del processo\n”);
          exit(1);
          break;

     // Figlio
     case 0:
          printf (“Sono il processo %d, generato da %d, e
          attendo un segnale da parte di mio padre\n”,
          getpid(), getppid());

          // Invio al processo padre il segnale SIGUSR1, un
          segnale “personalizzato”
          kill(getppid(), SIGUSR1);

          // Pongo il processo in attesa di un segnale
          attraverso la primitiva pause()
          pause();
          exit(0);
          break;

     // Padre
     default:
          // Mi metto in attesa di un segnale
          pause();

          // Una volta ricevuto il segnale SIGUSR1 da parte
          del figlio, mando al figlio il segnale SIGTERM
          kill(pid, SIGTERM);
          exit(0);
          break;
}
```

```
}
```

Programmazione multithread

Il concetto di thread, pur essendo operativamente molto simile a quello di processo, è in sostanza un concetto diverso. Fondamentalmente, l'inizializzazione di un nuovo processo è sempre qualcosa di oneroso per il sistema, in quanto un processo ha una sua area di memoria (ovvero una sua area di codice, una sua area di dati e un suo stack) e un suo PID che lo identifica all'interno della tabella dei processi, e alla sua inizializzazione il sistema operativo dovrà provvedere al nuovo processo le risorse di memoria richieste. Il thread invece lo possiamo vedere come un “mini-processo” (per usare una terminologia un po' grossolana ma che rende bene l'idea) che può essere richiamato all'interno di un processo stesso. Il thread condivide l'area di memoria con lo stesso processo chiamante (ovvero condivide con il processo chiamante gli stessi dati e la stessa area di stack, il che vuol dire che una modifica sulle variabili operata da un thread è visibile da tutti gli altri thread del processo stesso). Il vantaggio principale della programmazione multithread è la maggiore velocità di inizializzazione e di esecuzione di un thread rispetto a quella di un processo, a costo di una minore indipendenza in fatto di memoria condivisa tra i thread di un processo stesso.

Per ricorrere alla programmazione multithread in C in ambiente Unix useremo la libreria pthread (inclusa di default in molte installazioni Unix), e compileremo i sorgenti con l'opzione -lpthread.

Per creare un nuovo thread useremo la funzione *pthread_create()*, che prende come parametri un puntatore all'identificatore del thread (una variabile di tipo pthread_t, tipo definito nell'header sys/types.h), gli attributi del thread creato (generalmente NULL), il puntatore alla funzione contenente il codice che verrà eseguito dal thread (generalmente una funzione di tipo void* che prende un argomento di tipo void*) e un array contenente gli argomenti da passare alla funzione. Userò invece la funzione *pthread_exit()* per terminare l'esecuzione di un thread (questa funzione prende come parametro il valore da ritornare al processo chiamante) e *pthread_join()* per porre il processo chiamante in attesa finché il thread creato non viene terminato (questa funzione prende come argomenti l'identificatore del thread e un puntatore alla variabile in cui verrà salvato il valore di ritorno del thread).

Esempio:

```
#include <stdio.h>
#include <pthread.h>
#include <sys/types.h>

// Funzione che verrà eseguita dal thread
void* start(void* arg)  {
     printf (“Sono un thread richiamato dal processo padre\n”);
     pthread_exit(0);
}
```

```
main()  {
     // Identificatore del thread
     pthread_t t;
     int status;

     if (pthread_create(&t, NULL, start, NULL) != 0)  {
          printf (“Errore nella creazione del nuovo thread\n”);
          exit(1);
     }

     // Attendo che il thread venga terminato
     pthread_join(t, &status);

     printf (“Il thread a 0x%x è terminato con status %d\n”, &t,
     status);
}
```

Vediamo ora come passare degli argomenti alla funzione del thread:

```
#include <stdio.h>
#include <stdlib.h>
#include <pthread.h>
#include <sys/types.h>

// Funzione che verrà eseguita dal thread
void* start(void* arg)  {
     // L'argomento passato è sottoforma di dato void, ovvero un
     dato grezzo. Lo converto in int attraverso un operatore di
     cast
     int *my_arg = (int) arg;

     printf (“Sono un thread generato dal processo padre. Mi è
     stato passato l'argomento %d\n”, (*x));
     pthread_exit(0);
}

main(int argc, char **argv)  {
```

```
    pthread_t t;
    int status;
    int arg[1];

    if (argc<2)  {
        printf ("Passami almeno un parametro\n");
        exit(1);
    }

    arg[0]=atoi(argv[1]);

    if (pthread_create(&t, NULL, start, arg) != 0)  {
        printf ("Errore nella creazione del nuovo thread\n");
        exit(1);
    }

    // Attendo che il thread venga terminato
    pthread_join(t, &status);

    printf ("Il thread a 0x%x è terminato con status %d\n", &t,
    status);
}
```


Per suggerimenti, chiarimenti o errata corrige, contattate l'autore all'indirizzo
blacklight86@gmail.com