*Sistemi Operativi* (M. Cesati)

**Compito scritto del 5 settembre 2014**

**Nome:** _____ **Cognome:** _____

**Matricola:** _____ **Corso di laurea:** _____

**Crediti da conseguire:** 5 6 9

Scrivere i dati richiesti in stampatello. Al termine consegnare tutti i fogli. Per conseguire un voto sufficiente è necessario ottenere un voto non gravemente insufficiente in entrambi gli esercizi. Tempo a disposizione: 2,5 ore.

**Esercizio 1**. Scrivere una applicazione C/POSIX multithread che legge dallo standard input linee con il seguente formato:

*numero_decimale spazio stringa_di_testo*

Il numero decimale è considerato un *ritardo in secondi*. Per ciascuna linea con formato corretto l'applicazione crea un thread che attende il numero di secondi indicato e poi scrive in standard output la stringa di testo.

Ad esempio, passando sullo standard input le tre linee seguenti:

```
10 ritardo massimo
1 piccolo ritardo
5 ritardo medio
```

verranno scritti sullo standard output:

```
> piccolo ritardo     (dopo circa 1 secondo)
> ritardo medio       (dopo circa 5 secondi)
> ritardo massimo     (dopo circa 10 secondi)
```

Si noti che l'applicazione deve continuare a leggere linee dallo standard input anche durante la stampa ritardata delle stringhe (sono attività da svolgere contemporaneamente). Utilizzare opportune primitive di sincronizzazione tra i thread per evitare race condition sullo standard output.

**Esercizio 2**. Si descrivano i principali algoritmi di schedulazione dei dischi rigidi, evidenziando i vantaggi e gli svantaggi di ciascuno di essi.

*Sistemi Operativi* (M. Cesati)

**Esempio dei programmi del compito scritto del 5 settembre 2014**

## Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Il cuore del programma è un ciclo che legge dallo standard input:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>

int main(int argc, char *argv[])
{
    if (argc != 1) {
        fprintf(stderr, "Usage: %s (no arguments)\n", argv[0]);
        return EXIT_FAILURE;
    }
    while (!feof(stdin))
        process_input_line();
    pthread_exit(EXIT_SUCCESS);
}
```

La funzione `process_input_line()` alloca spazio sullo stack per una linea di testo (vettore `buf`). Assumiamo che la linea sia lunga al massimo `MAX_LINE_LENGTH` caratteri:

```
#define MAX_LINE_LENGTH 128

void process_input_line(void)
{
    char buf[MAX_LINE_LENGTH];
    char *msg;
    unsigned long delay;

    msg = parse_input_line(buf, MAX_LINE_LENGTH, &delay);
    if (msg != NULL)
        activate_alarm(delay, msg);
}
```

La funzione `parse_input_line()` legge la linea dallo standard input e la analizza. Il ritardo in secondi viene memorizzato in `delay`, mentre `msg` punta entro `buf` al primo carattere della stringa di testo. In caso di linea malformata `parse_input_line()` restituisce `NULL`.

```
char * parse_input_line(char *line, int maxsize, unsigned long *v)
{
    char *msg;

    if (fgets(line, maxsize, stdin) == NULL) {
        if (!feof(stdin))
            fprintf(stderr, "Error reading from standard input\n");
        return NULL;
    }
    errno = 0;
    *v = strtoul(line, &msg, 10);
    if (errno != 0) {
        fprintf(stderr, "Invalid number in standard input line\n");
        return NULL;
    }
    if (*msg++ != ' ') {
        fprintf(stderr, "Invalid separator in standard input line\n");
        return NULL;
    }

    /* remove the trailing new line, if any */
    if (msg[strlen(msg)-1] == '\n')
        msg[strlen(msg)-1] = '\0';
    return msg;
}
```

Per attivare l'allarme viene invocata la funzione `activate_alarm()` passando come argomenti `delay` e `msg`:

```
void activate_alarm(unsigned long delay, char *message)
{
    pthread_t tid;
    struct thread_data *p;

    p = malloc(sizeof(struct thread_data));
    if (!p) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    p->delay = delay;
    strncpy(p->message, message, MAX_LINE_LENGTH - 1);
    p->message[MAX_LINE_LENGTH - 1] = '\0';
    if (pthread_create(&tid, NULL, thread_job, p) != 0) {
        fprintf(stderr, "Error in pthread_create()\n");
        exit(EXIT_FAILURE);
    }
}
```

La struttura di dati `thread_data` viene allocata dinamicamente ed inizializzata con i dati necessari al thread per svolgere il suo lavoro. In effetti la struttura consiste di due soli campi:

```
struct thread_data {
    unsigned long delay;
    char message[MAX_LINE_LENGTH];
};
```

Il thread appena creato inizia ad eseguire la funzione `thread_job()`:

```
pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;

void *thread_job(void *arg)
{
    struct thread_data *td = (struct thread_data *)arg;
    if (pthread_detach(pthread_self())) {
        fprintf(stderr, "Error in pthread_detach()\n");
        exit(EXIT_FAILURE);
    }
    sleep(td->delay);   /* safe on Linux */
    if (pthread_mutex_lock(&mtx) != 0) {
        fprintf(stderr, "Error in pthread_mutex_lock()\n");
        exit(EXIT_FAILURE);
    }
    printf("> %s\n", td->message);
    if (pthread_mutex_unlock(&mtx) != 0) {
        fprintf(stderr, "Error in pthread_mutex_unlock()\n");
        exit(EXIT_FAILURE);
    }
    free(td);
    pthread_exit(0);
}
```

La prima operazione eseguita dal thread non è strettamente necessaria: il thread invoca `pthread_detach()` per fare in modo che, in uscita, tutte le risorse del thread vengano immediatamente liberate (in caso contrario alcune risorse del thread rimarrebbero allocate in attesa che qualche altro thread dell'applicazione esegua `pthread_join()` sul thread terminato).

Poi il thread si auto-sospende per il numero di secondi richiesto. Si noti che l'uso della funzione `sleep()` in una applicazione multithread è sicuro in Linux ma in generale poco consigliato in altri sistemi POSIX.

Quando il thread si risveglia deve scrivere il messaggio in standard output. Per evitare che più thread scrivano contemporaneamente sullo standard output viene utilizzato il mutex `mtx`, allocato come variabile globale ed inizializzato staticamente.

Infine il thread termina dopo aver rilasciato la propria struttura di dati `thread_data`.