Sistemi Operativi **(M. Cesati)**

Compito scritto del 22 gennaio 2018

Nome:		**Cognome:**	
Matricola:		**Corso di laurea:**	

Crediti da conseguire: 5 6 9

Scrivere i dati richiesti in stampatello. Al termine consegnare tutti i fogli. Tempo a disposizione: 2 ore.

Esercizio

Scrivere una applicazione C/POSIX multiprocesso costituita da 256 processi concorrenti. L'applicazione legge un file di input il cui nome è specificato sulla linea comando contenente valori numerici interi a 8 bit. Ciascuno dei 256 processi è associato ad un distinto valore tra 0 e 255, e conta il numero di occorrenze di tale valore all'interno del file. Al termine l'applicazione scrive in standard output il valore numerico (o i valori numerici, in caso di parità) che appare con maggior frequenza nel file. Curare gli aspetti di sincronizzazione tra i processi, ove necessario.

Sistemi Operativi **(M. Cesati)**

Esempio di programma del compito scritto del 22 gennaio 2018

Esercizio 1

Descriviamo il programma utilizzando un approccio "top-down". La funzione `main()` apre il file di input, crea una coda di messaggi per la comunicazione con gli altri processi dell'applicazione, crea i processi figli, analizza il file, attende i valori inviati dagli altri processi, seleziona i valori massimi trovati da tutti i processi, stampa i valori corrispondenti, ed infine termina distruggendo la coda di messaggi.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/msg.h>
#include <fcntl.h>

#define abort_on_error(cond, msg) do { \
    if (cond) { \
        fprintf(stderr, "%s (errno=%d [%m])\n ", \
            msg, errno); \
        exit(EXIT_FAILURE); \
    } \
} while (0)

int main(int argc, char *argv[])
{
    FILE *f;
    int freqs[256];
    int v, msq;

    abort_on_error(argc != 2,
        "Specify a file name as argument");
    f = open_file(argv[1]);
    msq = msgget(IPC_PRIVATE, S_IRUSR|S_IWUSR);
    abort_on_error(msq == -1,
        "Message queue creation failure");
    fork_children(argv[1], msq);
    freqs[0] = count_values(f, 0);
    v = collect_values(freqs, msq);
    print_max(freqs, v);
    v = msgctl(msq, IPC_RMID, NULL);
    abort_on_error(v == -1,
        "Message queue destruction failure");
```

```
    return EXIT_SUCCESS;
}
```

La macro `abort_on_error()` viene utilizzata per controllare le condizioni d'errore e, se necessario, interrompere prematuramente l'esecuzione del programma.

Per aprire il file di input si utilizza la semplice funzione `open_file()`:

```
FILE * open_file(const char *name)
{
    FILE *f;
    f = fopen(name, "r");
    abort_on_error(f == NULL, "Cannot open input file");
    return f;
}
```

Per creare la coda di messaggi viene utilizzata la API POSIX `msgget()`. Si noti l'uso dell'identificatore `IPC_PRIVATE`: poiché la coda di messaggi sarà utilizzata esclusivamente da processi generati dal processo che crea la coda, non c'è necessità di utilizzare una chiave IPC per identificare la risorsa. Le macro `S_IRUSR` e `S_IWUSR` garantiscono diritti di lettura e scrittura per tutti i processi dell'utente che ha lanciato l'applicazione.

Per creare i processi figli la funzione `main()` esegue `fork_children()`:

```
void fork_children(const char *filename, int msq)
{
    for (int i=1; i<256; ++i) {
        pid_t p = fork();
        abort_on_error(p == -1, "Error in fork()");
        if (p == 0)
            child_work(filename, i, msq);
    }
}
```

Vengono creati duecentocinquantacinque processi, in quanto il duecentocinquantaseiesimo è il processo lanciato da riga comando dall'utente. Ciascun processo figlio esegue la funzione `child_work()`, che analizzeremo in seguito. Ad ogni modo, questa funzione non fa mai ritorno e si conclude con la terminazione del processo figlio.

L'esecuzione di `main()` continua con l'invocazione della funzione `count_values()`, che legge il file di input e calcola il numero di valori a 8 bit nel file uguali a quello del secondo argomento della funzione (il valore 0 in questo caso). Tale numero è memorizzato nel primo elemento del vettore di duecentocinquantasei interi `freqs`.

```
int count_values(FILE *f, int value)
{
    int c = 0, counter = 0;
    for (;;) {
        c = fgetc(f);
        if (c == EOF)
            break;
        if (c != value)
            continue;
        ++counter;
    }
    abort_on_error(c == EOF && !feof(f),
        "Error reading from input file");
    return counter;
}
```

Continuiamo l'analisi della funzione `main()`: dopo aver ottenuto il numero di valori nel file pari a 0, `main()` esegue `collect_values()` per memorizzare in `freqs` le frequenze comunicate dai duecentocinquantacinque processi figli:

```
struct msgbuf {
    long mtype;
    int count;
    int value;
};

int collect_values(int *freqs, int msq)
{
    struct msgbuf m;

    int max = 0;
    for (int i=1; i<256; ++i) {
        ssize_t s = msgrcv(msq, &m, sizeof(m)-sizeof(long),
                           0, 0);
        abort_on_error(s == -1,
            "Error reading from message queue");
        abort_on_error(m.value<1 || m.value>255,
            "Invalid value in message");
        freqs[m.value] = m.count;
        if (m.count > max)
            max = m.count;
    }
    return max;
}
```

Il messaggio ricevuto è costituito essenzialmente dal valore considerato e dalla sua frequenza nel file. Si badi che il processo padre non può fare alcuna assunzione sull'ordine in cui i processi figli verranno posti in esecuzione, né tantomeno sull'ordine in cui i processi figli scriveranno il proprio valore nella coda

di messaggi. La funzione termina restituendo il massimo tra tutte le frequenze ricevute dai processi figli.

La funzione `main()` continua controllando se tale frequenza massima è inferiore alla frequenza del valore 0 calcolato direttamente; in tal caso, la frequenza massima viene aggiornata. Poi `main()` invoca la funzione `print_max()` per stampare i valori che occorrono il maggior numero di volte nel file:

```
void print_max(int *freqs, int max)
{
    printf("Max frequency is %d\nValue(s): ", max);
    for (int i=0; i<256; ++i)
        if (freqs[i] == max)
            printf("%hhu ", i);
    putchar('\n');
}
```

Infine, `main()` distrugge la coda di messaggi utilizzando la funzione di libreria `msgctl()` con il comando `IPC_RMID`.

Terminiamo l'analisi dell'applicazione descrivendo la funzione `child_work()` eseguita dai processi "figli":

```
void child_work(const char *filename, int value, int msq)
{
    FILE *f = open_file(filename);
    int freq = count_values(f, value);
    send_value(msq, freq, value);
    exit(EXIT_SUCCESS);
}
```

Un particolare molto importante da osservare è che la funzione non riceve la variabile di tipo `FILE *` già inizializzata dal processo padre; al contrario, il processo figlio invoca `open_file()` e apre nuovamente il file di input. Se infatti i processi figli operassero sui file aperti dal padre, utilizzerebbero tutti lo stesso puntatore alla posizione corrente nel file. Si renderebbe pertanto necessario una sincronizzazione rigorosa degli accessi di lettura al file. Poiché invece ciascuno dei processi figli effettua una apertura indipendente del file, ciascuno di essi opera con un diverso puntatore alla posizione corrente del file, e non è necessario implementare alcun meccanismo di sincronizzazione.

La funzione `child_work()` invoca `count_values()`, utilizzata anche dal processo padre, per contare le frequenze dei valori a 8 bit. Successivamente scrive il messaggio sulla coda utilizzando la funzione `send_value()`:

```c
void send_value(int msq, int freq, int value)
{
    struct msgbuf m;
    int rc;
    m.mtype = 1; /* qualsiasi valore eccetto 0 */
    m.count = freq;
    m.value = value;
    rc = msgsnd(msq, &m, sizeof(m)-sizeof(long), 0);
    abort_on_error(rc == -1,
        "Error writing to message queue");
}
```

Infine il processo figlio termina l'esecuzione.