Sistemi Operativi (M. Cesati)

Compito scritto del 30 gennaio 2017

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 multiprocesso costituita da 26 processi concorrenti. L'applicazione legge un file di input il cui nome è specificato sulla linea comando, e che contiene un file di testo ASCII. Ciascuno dei 26 processi è associato ad una lettera dell'alfabeto, e conta il numero di parole nel file che iniziano con la corrispondente lettera. Una parola è definita come una sequenza di caratteri alfabetici consecutivi; numeri, spazi bianchi, segni di punteggiatura, e tutti gli altri caratteri non alfabetici separano le parole tra loro ma non ne fanno parte. Al termine l'applicazione scrive in standard output la lettera dell'alfabeto (o le lettere dell'alfabeto, in caso di parità) che appare con maggior frequenza tra le iniziali delle parole nel file. Curare gli aspetti di sincronizzazione tra i processi, ove necessario.

Esercizio 2

Si descrivano in modo esauriente scopo e principi di funzionamento dei file system "annotati" (journaling file systems) utilizzati nei moderni sistemi operativi.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 30 gennaio 2017

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 le lettere 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 v, msq, freqs[26];

    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_words(f, 'a');
    v = collect_values(freqs, msq);
    if (freqs[0] > v)
        v = freqs[0];
    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<26; ++i) {
        pid_t p = fork();
        abort_on_error(p == -1, "Error in fork()");
        if (p == 0)
            child_work(filename, 'a'+i, msq);
    }
}
```

Vengono creati venticinque processi, in quanto il ventiseiesimo 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_words()`, che legge il file di input e calcola il numero di parole inizianti con la lettera passata come secondo argomento (la lettera "a" in questo caso). Tale valore è memorizzato nel primo elemento del vettore di ventisei interi `freqs`.

```
int count_words(FILE *f, char letter)
{
    int c = 0, prev_c, words = 0;
    char Letter = (letter - 'a') + 'A';

    for (;;) {
        prev_c = c;
        c = fgetc(f);
        if (c == EOF)
            break;
        if (c != letter && c != Letter)
            continue;
        if (is_alfa(prev_c))
            continue;
        ++words;
    }
    abort_on_error(c == EOF && !feof(f),
                   "Error reading from input file");
    return words;
}
```

La funzione assume che come secondo argomento venga passato il codice di una lettera alfabetica minuscola. Per prima cosa viene memorizzata nella variabile `Letter` il codice della corrispondente lettera maiuscola. Si noti che un altro modo per ottenere il codice della lettera maiuscola è quello di utilizzare la funzione di libreria `toupper()`.

La funzione inizia poi a leggere il file di input un carattere alla volta. Per ciascun carattere la funzione verifica se si tratta del carattere alfabetico considerato, in minuscolo o maiuscolo. Nel caso in cui lo sia, la funzione controlla se il precedente carattere letto era *non* alfabetico. Solo in questo caso conclude di aver trovato l'inizio di una parola iniziante per la lettera considerata, ed incrementa quindi il contatore `words`. Il valore finale del contatore è infine restituito alla funzione chiamante.

La funzione `is_alfa()` restituisce il valore 1 se l'argomento è il codice di una lettera dell'alfabeto, il valore 0 altrimenti:

```
int is_alfa(char c)
{
    if (c >= 'a' && c <= 'z')
        return 1;
    if (c >= 'A' && c <= 'Z')
        return 1;
    return 0;
}
```

Sarebbe stato anche possibile utilizzare la funzione di libreria `isalpha()` per effettuare questo lavoro, ma va tenuto presente che `isalpha()` può avere un

comportamento che dipende dalle impostazioni "locali" del sistema. Ad esempio, in un sistema con lingua italiana `isalpha()` potrebbe riconoscere come caratteri alfabetici anche le lettere accentate `à`, `è`, ecc.

Continuiamo l'analisi della funzione `main()`: dopo aver ottenuto il numero di parole del file inizianti con la lettera "a", `main()` esegue `collect_values()` per memorizzare in `freqs` le frequenze comunicate dai venticinque processi figli:

```
struct msgbuf {
    long mtype;
    int count;
    char letter;
};

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

    int max = 0;
    for (int i=1; i<26; ++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.letter<'a' || m.letter>'z',
                       "Invalid letter in message");
        freqs[m.letter-'a'] = m.count;
        if (m.count > max)
            max = m.count;
    }
    return max;
}
```

Il messaggio ricevuto è costituito essenzialmente dal valore della frequenza e dal codice della lettera alfabetica (minuscola) al quale il valore si riferisce. 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 valore massimo è inferiore al valore della frequenza per la lettera "a" calcolato direttamente; in tal caso, il valore massimo viene aggiornato. Poi `main()` invoca la funzione `print_max()` per stampare le lettere iniziali che occorrono il maggior numero di volte tra tutte le parole del file:

```
void print_max(int *freqs, int max)
{
    fprintf(stdout, "Max freq. is %d\nLetter(s): ", max);
    for (int i=0; i<26; ++i)
```

```
        if (freqs[i] == max) {
            fputc('a'+i, stdout);
            fputc(' ', stdout);
        }
    fputc('\n', stdout);
}
```

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, char letter, int msq)
{
    FILE *f = open_file(filename);
    int freq = count_words(f, letter);
    send_value(msq, freq, letter);
    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_words()`, utilizzata anche dal processo padre, per contare le frequenze delle lettere. Successivamente scrive il messaggio sulla coda utilizzando la funzione `send_value()`:

```
void send_value(int msq, int freq, char letter)
{
    struct msgbuf m;
    int rc;
    m.mtype = 1; /* qualsiasi valore eccetto 0 */
    m.count = freq;
    m.letter = letter;
    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.