Sistemi Operativi (M. Cesati)

Compito scritto del 7 settembre 2012

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 1. Scrivere una applicazione C/POSIX 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 processo 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).

Esercizio 2. Scrivere un programma C/POSIX che continuamente attende messaggi di testo da una coda di messaggi IPC e li stampa sullo standard output. La coda di messaggi IPC è già esistente ed è identificabile tramite la chiave ottenibile con il percorso "`/tmp`" e il carattere identificativo "`!`". Si assuma che i messaggi di testo abbiano una lunghezza massima di 256 caratteri.

Esercizio 3. Si descrivano in modo esauriente le strutture di dati su disco necessarie per implementare un file system di un sistema operativo di tipo Unix.

Sistemi Operativi (M. Cesati)

Esempio dei programmi del compito scritto del 7 settembre 2012

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();

    return 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 128 caratteri:

```
void process_input_line(void)
{
    char buf[128];
    char *msg;
    unsigned long delay;

    msg = parse_input_line(buf, 128, &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`.

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

    /* read a line from standard input */
    if (fgets(line, maxsize, stdin) == NULL) {
        if (!feof(stdin))
            fprintf(stderr, "Error reading from standard input\n");
        return NULL;
    }

    /* parse the line */
    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`:

```c
void activate_alarm(unsigned long delay, char *message)
{
    pid_t p = fork();
    if (p == -1) {
        fprintf(stderr, "Error in fork()\n");
        exit(EXIT_FAILURE);
    }
    if (p != 0)
        return; /* the parent continues to wait for input */
    sleep(delay); /* don't care about signals */
    printf("> %s\n", message);
    exit(EXIT_SUCCESS);
}
```

Il processo principale crea un figlio e termina la funzione (tornando quindi a leggere dallo standard input). Il figlio invoca `sleep()` per auto-sospendersi per il numero richiesto di secondi, poi stampa il messaggio e termina.

Esercizio 2

Adottiamo ancora un approccio top-down. Il programma deve (1) aprire la coda di messaggi e (2) stampare i messaggi letti dalla coda:

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

int main(int argc, char *argv[])
{
    int mq_desc;

    if (argc != 1) {
        fprintf(stderr, "Usage: %s  (no arguments)\n", argv[0]);
        return EXIT_FAILURE;
    }

    mq_desc = open_message_queue();
    print_messages(mq_desc);
    return EXIT_SUCCESS; /* never reached */
}
```

La funzione `open_message_queue()` ricava la chiave IPC della coda di messaggi e apre la risorsa, restituendone il suo identificatore:

```
int open_message_queue(void)
{
    int desc;
    key_t mq_id;

    /* obtain the message queue IPC key */

    mq_id = ftok("/tmp", '!');
    if (mq_id == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }

    /* open the message queue */

    desc = msgget(mq_id, 0644);
    if (desc == -1) {
        perror("msgget");
        exit(EXIT_FAILURE);
    }
```

```
    return desc;
}
```

È ora necessario definire il formato del messaggio letto dalla coda. Oltre al campo obbligatorio `mtype` definiamo un campo `message` corrispondente ad un vettore di 256 caratteri (lunghezza massima della stringa di testo compreso il terminatore finale).

```
struct msgbuf {
    long mtype;             /* message type */
    char message[256]; /* the message (text string) */
};
```

La lettura e stampa dei messaggi della coda è effettuata dalla funzione `print_messages()`:

```
void print_messages(int mq)
{
    struct msgbuf msg;
    ssize_t len;

    for(;;) {
        len = msgrcv(mq, &msg, 256, 0, 0);
        if (len == -1) {
            perror("msgrcv");
            exit(EXIT_FAILURE);
        }
        msg.message[len-1]='\0'; /* force string terminator */
        printf("> %s\n", msg.message);
    }
}
```

Nel ciclo senza fine si legge il messaggio tramite `msgrcv()` e lo si stampa con `printf()`. Il quarto parametro di `msgrcv()` (0) indica che si vuole ricevere il primo messaggio in coda, senza considerazione per il suo tipo. Il quinto parametro invece sono i flag: non ne occorre alcuno.