Sistemi Operativi (M. Cesati)

Compito scritto del 17 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. In un file di testo è memorizzata una sequenza di lunghezza indefinita di numeri interi di tipo `int` separati tra loro da spazi bianchi e/o caratteri di fine riga (`\n`). Scrivere un programma C/POSIX che riceve il nome del file di testo da linea comando; esso deve leggere i numeri dal file e memorizzarli in una lista, inserendo volta per volta l'elemento letto per ultimo in cima alla lista. La lista deve essere realizzata come una struttura di dati dinamica utilizzando i puntatori.

Dopo aver costruito la lista il programma deve effettuarne una scansione stampando i suoi elementi in standard output. In pratica quindi il programma deve stampare i numeri in ordine inverso rispetto al file.

Esercizio 2. Scrivere un programma C/POSIX che continuamente legge righe di testo dallo standard input e le scrive in un'area di memoria condivisa IPC. L'area di memoria IPC è già esistente ed è identificabile tramite la chiave ottenibile con il percorso "`/tmp`" e il carattere identificativo "`!`". Si assuma che ciascuna riga di testo abbia una lunghezza massima di 256 caratteri. Dopo ciascuna scrittura sull'area di memoria condivisa il programma attende di ricevere un segnale `SIGUSR1` prima di leggere la successiva riga di testo dallo standard input.

Esercizio 3. Si descrivano in modo esauriente il ruolo, il meccanismo di funzionamento e una possibile implementazione dei "device file" utilizzati nei sistemi operativi di tipo Unix.

Sistemi Operativi (M. Cesati)

Esempio dei programmi del compito scritto del 17 settembre 2012

Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Le operazioni principali del programma sono (1) apertura del file di ingresso, (2) lettura dal file ed inserimento nella lista, (3) stampa degli elementi della lista, e (4) chiusura del programma.

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

int main(int argc, char *argv[])
{
    FILE *inf;
    struct lista_t *head = NULL;

    if (argc != 2) {
        fprintf(stderr,"Usage: %s <nome file>\n",argv[0]);
        return EXIT_FAILURE;
    }

    inf = open_file(argv[1]);
    fill_list(inf, &head);
    print_list(head);
    close_file(inf);
    /* dynamic memory is freed by OS on process exit */
    return EXIT_SUCCESS;
}
```

Ciascun elemento della lista è costituito da una struttura di dati di tipo `struct lista_t`:

```
struct lista_t {
    struct lista_t *next;
    int val;
};
```

L'apertura e la chiusura del file di ingresso sono effettuate dalle funzioni `open_file()` e `close_file()`:

```
FILE * open_file(const char *name)
{
    FILE *f;

    f = fopen(name, "r");
    if (f == NULL) {
        perror(name);
        exit(EXIT_FAILURE);
    }
    return f;
}

void close_file(FILE *inf)
{
    if (fclose(inf) != 0) {
        perror("fclose");
        exit(EXIT_FAILURE);
    }
}
```

La funzione `fill_list()` legge i numeri interi dal file di ingresso e li passa alla funzione `insert_h()` perché siano inseriti in cima alla lista. Si noti che la funzione `fscanf()` assume che i numeri in ingresso siano separati da una sequenza di spazi, tab e/o caratteri '`\n`':

```
void fill_list(FILE *f, struct lista_t **phead)
{
    int v;

    for (;;) {
        if (fscanf(f, "%d", &v) == 1)
            insert_h(phead, v);
        else
            break;
    }
    if (!feof(f)) {
        fprintf(stderr, "Invalid number in input file\n");
        exit(EXIT_FAILURE);
    }
}
```

La funzione `insert_h()` inserisce un elemento `v` in cima alla lista il cui elemento di testa è puntato dalla variabile il cui indirizzo è in `ph`:

```
void insert_h(struct lista_t **ph, int v)
{
    struct lista_t *p;
```

```
    p = malloc(sizeof(struct lista_t));
    if (p == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }

    p->val = v;
    p->next = *ph;
    *ph = p;
}
```

Infine la funzione `print_list()` stampa in ordine tutti gli elementi della lista il cui primo elemento è passato in `head`:

```
void print_list(struct lista_t *head)
{
    while (head != NULL) {
        printf("%d ", head->val);
        head = head->next;
    }
    putchar('\n');
}
```

Si noti che l'argomento `head` è assimilabile ad una variabile locale il cui contenuto iniziale è impostato dalla funzione chiamante; perciò è possibile modificarne il valore all'interno di `print_list()` senza conseguenze per la funzione chiamante.

Esercizio 2

Adottiamo un approccio top-down: le operazioni principali svolte dal programma sono: (1) cattura del segnale `USR1`, (2) apertura della regione di memoria condivisa, e (3) copia delle righe di testo dallo standard input alla regione di memoria.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <string.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <sys/ipc.h>
#include <sys/shm.h>

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

```
    char *shm_area;

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

    catch_signal();
    shm_area = open_shared_memory();
    copy_messages(shm_area);
    return EXIT_SUCCESS; /* never reached */
}
```

Per catturare il segnale definiamo il gestore, `sig_handler()`, che in effetti non ha necessità di svolgere alcuna operazione. La modifica della gestione del segnale `SIGUSR1` è effettuata dalla funzione `catch_signal()`.

```
void sig_handler(int sig)
{
    sig=sig; /* do nothing */
}

void catch_signal(void)
{
    struct sigaction sa;
    sa.sa_handler = sig_handler;
    if (sigemptyset(&sa.sa_mask) == -1) {
        fprintf(stderr, "Error handling the signal set\n");
        exit(EXIT_FAILURE);
    }
    sa.sa_flags = 0;
    if (sigaction(SIGUSR1, &sa, NULL) == -1) {
        perror("sigaction");
        exit(EXIT_FAILURE);
    }
    return;
}
```

Per aprire la regione di memoria condivisa IPC utilizziamo la funzione `open_shared_memory()`:

```
char * open_shared_memory(void)
{
    int desc;
    key_t shm_id;
    void *addr;
```

```
    /* obtain the IPC key */
    shm_id = ftok("/tmp", '!');
    if (shm_id == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    /* open the shared memory area */
    desc = shmget(shm_id, 256, 0644);
    if (desc == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    /* attach to the process address space */
    addr = shmat(desc, NULL, 0);
    if (addr == (void *) -1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    return addr;
}
```

Infine, il cuore del programma è una funzione `copy_messages()` con ciclo che legge righe di testo dallo standard input per mezzo di `fgets()` e le copia sulla regione di memoria condivisa:

```
void copy_messages(char *shm)
{
    char buf[256];

    for (;;) {
        if (fgets(buf, 256, stdin) == NULL)
            break;
        strncpy(shm, buf, 256);
        pause(); /* wait for any catched signal */
    }
    if (!feof(stdin)) {
        perror("standard input");
        exit(EXIT_FAILURE);
    }
}
```

Si noti che la API `pause()` blocca il processo fino alla ricezione di un qualunque segnale catturato o che provoca la terminazione del processo stesso. Poiché è stato catturato esclusivamente il segnale `SIGUSR1` il programma per semplicità non controlla il tipo di segnale ricevuto. Tale controllo è facilmente inseribile, ad esempio definendo una variabile globale incrementata esclusivamente nel gestore del segnale `sig_handler()`.