Sistemi Operativi (M. Cesati)

Compito scritto del 5 febbraio 2013

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 un programma C/POSIX che legge da una pipe (aperta tramite `pipe()` oppure `popen()`) una sequenza di numeri in formato testo decimale separati tra loro da spazi bianchi e/o caratteri di fine riga (`\n`). Ad intervalli di tempo regolari, ogni 60 secondi, il programma interrompe la lettura dalla pipe per scrivere i numeri in ordine crescente in standard output, seguiti dalla loro somma. Il programma ritorna poi a leggere numeri dalla pipe.

Si noti che la stampa e la somma debbono essere relativi soltanto ai numeri letti dalla pipe nell'ultimo periodo di tempo.

Esercizio 2. Si descrivano in modo esauriente il ruolo, le differenze e le relazioni esistenti tra i concetti di "chiamata di sistema", "API" e "funzioni di libreria".

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 5 febbraio 2013

Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Le operazioni principali del programma sono (1) la creazione della pipe, (2) la creazione del processo "scrittore", che dovrà scrivere numeri sulla pipe, (3) l'installazione di gestore per il segnale `SIGALRM` usato per realizzare la suddivisione in periodi da 60 secondi, e (4) l'esecuzione del ciclo di lettura dalla pipe.

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

int main(int argc, char *argv[])
{
    int fds[2];
    if (argc != 1) {
        fprintf(stderr, "Usage: %sn", argv[0]);
        return EXIT_FAILURE;
    }

    /* crea una pipe */
    if (pipe(fds) == -1) {
        fprintf(stderr, "Cannot create a pipe\n");
        return EXIT_FAILURE;
    }

    /* crea un processo tramite fork per lo scrittore
     *(fds[1] e' il descrittore per la scrittura) */
    /* non e' richiesto implementare questa funzione */
    spawn_writer(fds);

    /* chiudi il descrittore per la scrittura */
    close(fds[1]);

    /* installa un gestore per il segnale SIGALRM */
    catch_alarm_signal();

    /* effettua la lettura e l'ordinamento */
    timed_read_and_sort(fds[0]);

    /* not reached */
    return EXIT_SUCCESS;
```

```
}
```

Non era richiesto dal testo implementare la funzione `spawn_writer()`. Passiamo dunque ad analizzare la funzione `catch_alarm_signal()`:

```
unsigned int alarms = 0;

void alarm_hndl(int sig)
{
    sig=sig; /* no "unused argument" message from gcc */
    ++alarms;
}

void catch_alarm_signal(void)
{
    struct sigaction sa;

    sigemptyset(&sa.sa_mask);
    sa.sa_handler = alarm_hndl;
    if (sigaction(SIGALRM, &sa, NULL) == -1) {
        fprintf(stderr, "Cannot install the handler\n");
        exit(EXIT_FAILURE);
    }
}
```

Il gestore di segnali si limita ad incrementare la variabile globale `alarms`.

La funzione `timed_read_and_sort()` è il cuore del programma: esegue un ciclo senza fine in cui legge dati dalla pipe, ordina i numeri letti e li scrive in standard output:

```
void timed_read_and_sort(int fd)
{
    char *pbuf; /* puntatore ad un buffer allocato
       dinamicamente */

    for (;;) {
        /* invia un segnale SIGALRM tra 60 secondi */
        alarm(60);
        pbuf = timed_read_from_pipe(fd);
        if (pbuf) {
            sort_and_print(pbuf);
            free(pbuf);
        }
    }
}
```

Si noti che la funzione si limita ad accumulare i byte letti dalla pipe in un buffer allocato dinamicamente entro `timed_read_from_pipe()`:

```
char * timed_read_from_pipe(int fd)
{
    char line[80];
    ssize_t n, tot;
    char *pbuf = NULL;
    unsigned int old_alarms = alarms;

    tot = 0;
    while (old_alarms == alarms) {
        n = read(fd, line, 80);
        if (n == -1) {
            if (errno != EINTR) {
                fprintf(stderr,
                        "Error reading from the pipe\n");
                exit(EXIT_FAILURE);
            }
            /* il segnale e' arrivato e non sono stati
               letti nuovi dati */
            break;
        }
        if (n == 0) {
            /* EOF: la pipe e' stata chiusa */
            if (pbuf) {
                sort_and_print(pbuf);
                exit(EXIT_SUCCESS);
            }
        }
        tot += n;
        pbuf = realloc(pbuf, tot*sizeof(char));
        if (pbuf == NULL) {
            fprintf(stderr, "Memory reallocation error\n");
            exit(EXIT_FAILURE);
        }
        memcpy(pbuf+tot-n, line, n*sizeof(char));
    }
    return pbuf;
}
```

La funzione `sort_and_print()` legge i numeri dal buffer e li inserisce in ordine in una lista dinamica:

```
void sort_and_print(char *buf)
{
    struct list_element *list = NULL;
    char *p = buf, *q;
    long total = 0;
```

```
    while (*p != '\0') {
        long v;

        while (*p == ' ' || *p == '\n') {
            p++;
        v = read_number(p, &q);
        insert_ordered(v, &list);
        total += v;
        p = q;
    }

    print_list(list);
    printf("Total: %ld\n", total);
    free_list(list);
}
```

La funzione `read_number()` legge un numero intero dal buffer di testo e scrive nel secondo parametro il puntatore alla posizione successiva nel buffer:

```
long read_number(char *str, char **next)
{
    long v;
    char *p;

    errno = 0;
    v = strtol(str, &p, 0);
    if (errno != 0 || (*p != ' ' && *p != '\n')) {
        fprintf(stderr, "Invalid number read from pipe\n");
        exit(EXIT_FAILURE);
    }
    *next = p+1;
    return v;
}
```

Ciascun elemento della lista è costituito da una struttura di dati di tipo `struct list_element`. Per inserire gli elementi in lista si definisce la funzione `insert_ordered()`:

```
struct list_element {
    long v;
    struct list_element *next;
};

void insert_ordered(long v, struct list_element **phead)
{
    struct list_element *e, *t;

    e = malloc(sizeof(struct list_element));
    if (!e) {
```

```
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }

    e->v = v;

    t = *phead;
    if (t == NULL /* primo elemento in lista */ ||
            t->v > v /* inserimento in cima */) {
        *phead = e;
        e->next = t;
        return;
    }

    while (t->next != NULL && t->next->v < v)
        t = t->next;
    e->next = t->next; /* NULL se in fondo alla lista */
    t->next = e;
}
```

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

```
void print_list(struct list_element *head)
{
    while (head != NULL) {
        printf("%ld ", head->v);
        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.

Infine la funzione `free_list()` rilascia gli elementi della lista:

```
void free_list(struct list_element *head)
{
    struct list_element *t;

    while (head != NULL) {
        t = head->next;
        free(head);
        head = t;
    }
}
```