Sistemi Operativi (M. Cesati)

Compito scritto del 19 luglio 2013 (turno 1)

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.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Scrivere una applicazione C/POSIX che avvia l'esecuzione di un programma esterno con percorso "`/tmp/producer`" e legge i dati forniti in standard output da tale programma.

I dati prodotti dal programma esterno sono costituiti da un numero finito e variabile di strutture del seguente tipo:

```
struct record {
    int key;
    char name[32];
};
```

Il formato dei dati è nativo per l'architettura del calcolatore; in altre parole, la sequenza di byte prodotta dal programma esterno corrisponde alla rappresentazione in RAM delle varie strutture `record`, una dopo l'altra. I record forniti dal programma esterno non hanno un ordine particolare.

L'applicazione deve inserire i record forniti dal programma esterno in una lista dinamica ordinata in base al valore del campo `key`. Inoltre, prima di terminare l'esecuzione, l'applicazione deve stampare in standard output i record ordinati nella lista con il seguente formato: una riga di testo per ciascun differente valore di `key` contenente il valore di `key` e tutte le stringhe `name` associate a quel valore.

Si assuma che il campo `name` di ogni record contenga sempre una stringa di lunghezza variabile terminata da `'\0'`.

Esercizio 2. Si descriva in modo esauriente la differenza tra un sistema operativo monolitico ed uno basato su microkernel.

Sistemi Operativi (M. Cesati)

Compito scritto del 19 luglio 2013 (turno 2)

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.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Scrivere una applicazione C/POSIX che legge da un file, il cui percorso è passato come argomento del programma, un insieme di dati costituito da strutture del seguente tipo:

```
struct record {
    int key;
    char name[32];
};
```

Il formato del file è nativo per l'architettura del calcolatore; in altre parole, il file contiene solo la sequenza di byte corrispondente alla rappresentazione in RAM delle varie strutture `record`, una dopo l'altra. I record nel file non hanno un ordine particolare.

L'applicazione deve inserire i record letti dal file in una lista dinamica ordinata in base al valore del campo `key`. Diversi record nel file possono avere la stessa chiave `key`. Se una certa chiave `key` appare in totale un numero dispari di volte tra i record, la lista alla fine dovrà contenere l'ultimo record con tale chiave letto dal file; se invece una certa chiave `key` appare in totale un numero pari di volte tra i record nel file, la lista non dovrà contenere alcun record con tale chiave.

Inoltre, prima di terminare l'esecuzione, l'applicazione deve stampare in standard output i record ordinati nella lista (una riga di testo per ciascun elemento della lista). Si assuma che il campo `name` sia una stringa di lunghezza variabile terminata da `'\0'`.

Esercizio 2. Si descrivano in modo esauriente le possibili implementazioni a livello di kernel e/o librerie di sistema del supporto per le applicazioni "multi-thread".

Sistemi Operativi (M. Cesati)

Compito scritto del 19 luglio 2013 (turno 3)

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.
Tenere presente che saranno valutati anche l'ordine e la chiarezza dell'esposizione.

Esercizio 1. Scrivere una applicazione C/POSIX che legge da una regione di memoria condivisa IPC di dimensione pari a 4 pagine e la cui chiave è ricavabile utilizzando il percorso `/tmp` ed il codice `'2'`. La regione di memoria contiene un insieme di dati costituito da un numero variabile di strutture del seguente tipo:

```
struct record {
    int key;
    double value;
};
```

Il formato dei dati è nativo per l'architettura del calcolatore; in altre parole, la regione di memoria contiene solo la sequenza di byte corrispondente alla rappresentazione in RAM delle varie strutture `record`, una dopo l'altra. I record non hanno un ordine particolare, e l'ultimo record significativo nella regione è seguito da un record in cui `key` ha il valore speciale -1.

L'applicazione deve inserire i record letti dalla regione in una lista dinamica ordinata in base al valore del campo `key`. Diversi record corrispondenti allo stesso valore di `key` nella regione di memoria debbono essere rappresentati da un unico elemento nella lista, ed il campo `value` associato a tale elemento nella lista deve essere la somma dei campi `value` nei record originali.

Inoltre, prima di terminare l'esecuzione, l'applicazione deve stampare in standard output i record ordinati nella lista (una riga di testo per ciascun elemento della lista).

Esercizio 2. Descrivere le principali soluzioni adottate nei sistemi operativi moderni per gestire le architetture multicore e multiprocessore.

Sistemi Operativi (M. Cesati)

Esempio di programmi del compito scritto del 19 luglio 2013

Esercizio 1 — Turno 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo con la definizione delle strutture di dati occorrenti per modellare i record e gli elementi della lista:

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

struct record {
    int key;
    char name[32];
};

struct listel {
    struct record rec;
    struct listel *next;
};
```

La struttura `listel` è costituita da una sotto-struttura di tipo `record` e da un puntatore alla struttura `listel` successiva nella lista.

La funzione `main()` deve invocare il programma esterno facendo in modo di leggere il suo standard output, costruire la lista dinamica, ed infine stampare in standard output il contenuto della lista:

```
int main()
{
    FILE *fpipe;
    struct listel *list_head = NULL;

    fpipe = run_extern_program();
    build_list(fpipe, &list_head);
    print_list(list_head);

    return EXIT_SUCCESS;
}
```

La variabile `list_head` memorizza il puntatore al primo elemento della lista dinamica.

La funzione `run_extern_program()` invoca il programma esterno collegandolo tramite una pipe:

```
FILE *run_extern_program(void)
{
    const char *path = "/tmp/producer";
    FILE *f = popen(path, "r");
    if (f == NULL) {
        fprintf(stderr, "Error executing popen(\"%s\")\n",
                path);
        exit(EXIT_FAILURE);
    }
    return f;
}
```

La funzione `build_list()` legge dalla pipe ed inserisce man mano i record nella lista dinamica:

```
void build_list(FILE * f, struct listel **plh)
{
    struct listel *lep;
    for (;;) {
        lep = get_record(f);     /* abort on error */
        if (lep == NULL)
            break;               /* break on EOF */
        insert_sorted_list(lep, plh);
    }
}
```

La funzione `get_record()` legge un nuovo record dalla pipe. In caso di errore termina l'applicazione, altrimenti restituisce l'indirizzo di un nuovo elemento della lista (ancora da inserire), ovvero `NULL` in caso di EOF sulla pipe:

```
struct listel *get_record(FILE * f)
{
    struct listel *p = alloc_node();
    p->next = NULL;
    if (fread(&p->rec, sizeof(struct record), 1, f) == 1)
        return p;
    if (feof(f))
        return NULL;
    fprintf(stderr, "Error reading from the pipe\n");
    exit(EXIT_FAILURE);
}
```

Per allocare un nuovo elemento della lista viene utilizzata la funzione `alloc_node()`:

```
struct listel *alloc_node(void)
{
    struct listel *p;
    p = malloc(sizeof(struct listel));
    if (p == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    return p;
}
```

Per cercare la posizione corretta di un nuovo elemento da inserire in lista, si utilizza la funzione `insert_sorted_list()`:

```
void insert_sorted_list(struct listel *new,
                        struct listel **pnext)
{
    struct listel *p;
    for (p= *pnext; p!=NULL; pnext= &p->next, p= p->next)
        if (p->rec.key > new->rec.key) {
            /* inserisci new dopo pnext, prima di p */
            insert_after_node(new, pnext);
            return;
        }
    /* l'elemento va inserito in fondo alla lista */
    insert_after_node(new, pnext);
}
```

Nel momento in cui la posizione corretta, dipendente dal valore della chiave, è stata determinata, l'elemento è inserito nella lista tramite la funzione `insert_after_node()`:

```
void insert_after_node(struct listel *new,
                       struct listel **pnext)
{
    new->next = *pnext;
    *pnext = new;
}
```

Infine, per stampare prima del termine dell'esecuzione il contenuto della lista dinamica, la funzione `main()` invoca `print_list()`:

```c
void print_list(struct listel *p)
{
    while (p != NULL) {
        int key = p->rec.key;
        printf("%d :", key);
        while (p != NULL && p->rec.key == key) {
            printf(" %s", p->rec.name);
            p = p->next;
        }
        putchar('\n');
    }
}
```

Esercizio 1 — Turno 2

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo con la definizione delle strutture di dati occorrenti per modellare i record e gli elementi della lista:

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

struct record {
    int key;
    char name[32];
};

struct listel {
    struct record rec;
    struct listel *next;
};
```

La funzione `main()` deve leggere il nome del file dalla linea comando, aprire il file, leggere i record dal file e costruire la lista dinamica, ed infine stampare il contenuto della lista prima di terminare:

```c
int main(int argc, char *argv[])
{
    FILE *fin;
    struct listel *list_head = NULL;
```

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

    fin = open_file(argv[1]);
    build_list(fin, &list_head);
    print_list(list_head);

    return EXIT_SUCCESS;
}
```

La variabile `list_head` memorizza l'indirizzo del primo elemento della lista dinamica.

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

```
FILE *open_file(const char *path)
{
    FILE *f = fopen(path, "r");
    if (f == NULL) {
        fprintf(stderr, "Error opening file \"%s\"\n",
                path);
        exit(EXIT_FAILURE);
    }
    return f;
}
```

Per costruire la lista si utilizza la funzione `build_list()`. Essa è del tutto identica alla omonima funzione descritta nella soluzione dell'esercizio per il Turno 1: invoca ripetutamente la funzione `get_record()` per leggere un nuovo record e successivamente invoca `insert_sorted_list()` per inserire il record nella lista.

Anche la funzione `get_record()` è del tutto identica a quella omonima descritta nella soluzione dell'esercizio per il Turno 1.

Al contrario, la funzione `insert_sorted_list()` invocata da `build_list()` deve essere differente. Infatti, per ogni nuovo elemento da inserire nella lista è necessario controllare se la lista contiene già un elemento con la stessa chiave. In questo caso, il nuovo elemento non deve essere inserito, ed inoltre deve essere rimosso dalla lista l'elemento con la chiave. In questo modo, fissato un valore per la chiave, nella lista esisterà sempre al più un elemento con tale valore; alla fine la lista conterrà un elemento con tale chiave se e solo se il numero totale di record letti con tale chiave è dispari, e l'elemento in lista corrisponderà all'ultimo record letto dal file avente la stessa chiave.

```
void insert_sorted_list(struct listel *new,
                        struct listel **pnext)
{
    struct listel *p;
    for (p= *pnext; p!=NULL; pnext= &p->next, p= p->next) {
        if (p->rec.key == new->rec.key) {
            /* cancella p, l'elemento dopo pnext */
            remove_after_node(pnext);
            free(p);
            free(new);
            return;
        }
        if (p->rec.key > new->rec.key) {
            /* inserisci new dopo pnext, prima di p */
            insert_after_node(new, pnext);
            return;
        }
    }
    /* l'elemento va inserito in fondo alla lista */
    insert_after_node(new, pnext);
}
```

Nel caso in cui si trovi in lista un elemento con la stessa chiave del nuovo elemento, entrambe le strutture vengono de-allocate. Per rimuovere un elemento dalla lista si utilizza la semplice funzione `remove_after_node()`:

```
void remove_after_node(struct listel **pnext)
{
    *pnext = (*pnext)->next;
}
```

Infine la funzione `print_list()` che stampa il contenuto della lista:

```
void print_list(struct listel *p)
{
    while (p != NULL) {
        printf("%d : %s\n", p->rec.key, p->rec.name);
        p = p->next;
    }
}
```

Esercizio 1 — Turno 3

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo con la definizione delle strutture di dati occorrenti per modellare i record e gli elementi della lista:

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

struct record {
    int key;
    double value;
};

struct listel {
    struct record rec;
    struct listel *next;
};
```

La funzione `main()` apre la regione di memoria condivisa, legge i record dalla regione di memoria costruendo la lista dinamica, ed infine stampa i record nella lista:

```
int main()
{
    struct record *shared_region;
    struct listel *list_head = NULL;

    shared_region = open_shmem();
    build_list(shared_region, &list_head);
    print_list(list_head);

    return EXIT_SUCCESS;
}
```

La variabile `list_head` memorizza l'indirizzo del primo elemento della lista dinamica; la variabile `shared_region` punta all'inizio della regione di memoria condivisa.

Per aprire la regione di memoria condivisa viene utilizzata la funzione `open_shmem()`:

```
struct record * open_shmem(void)
{
    key_t key;
    int shmid;
    struct record *reg;
    int page_size = sysconf(_SC_PAGESIZE);
    if (page_size == -1) {
        perror("sysconf");
        exit(EXIT_FAILURE);
    }
    key = ftok("/tmp", '2');
    if (key == -1) {
        perror("ftok");
        exit(EXIT_FAILURE);
    }
    shmid = shmget(key, 4*page_size, 0);
    if (shmid == -1) {
        perror("shmget");
        exit(EXIT_FAILURE);
    }
    reg = shmat(shmid, NULL, 0);
    if (reg == (void *) -1) {
        perror("shmat");
        exit(EXIT_FAILURE);
    }
    return reg;
}
```

La funzione determina la dimensione della pagina per mezzo dell'API `sysconf()`, poi apre la regione di memoria per una lunghezza pari a 4 pagine, come richiesto dal testo dell'esercizio.

Per leggere i record e costruire la lista viene utilizzata la funzione `build_list()`:

```
void build_list(struct record *reg, struct listel **plh)
{
    struct listel *lep;

    for (;;) {
        lep = get_record(&reg);
        /* reg advanced by get_record() */
        if (lep == NULL)
            break;   /* break on end of records */
        insert_sorted_list(lep, plh);
    }
}
```

Si noti che il parametro `reg`, utilizzato per passare l'indirizzo iniziale della regione di memoria condivisa, è poi utilizzato come una variabile locale che tiene traccia della posizione corrente nella regione di memoria.

La funzione `get_record()` alloca un nuovo elemento per la lista, legge il record dalla regione di memoria e inserisce l'elemento in lista, se necessario:

```
struct listel *get_record(struct record **preg)
{
    struct listel *p = alloc_node();
    p->next = NULL;

    p->rec = *(*preg); /* copy the whole record */
    ++(*preg);
    return (p->rec.key == -1 ? NULL : p);
}
```

Il cuore della funzione è l'assegnazione `p->rec = *(*preg)`, che provoca la copia dell'intero record referenziato dall'indirizzo in `*preg` entro la sotto-struttura `rec` contenuta nell'elemento della lista indirizzato da `p`. Successivamente viene incrementato l'indirizzo memorizzato in `preg` in modo da farlo puntare al record successivo nella regione di memoria, e si ritorna l'indirizzo dell'elemento `p`. Se però l'elemento appena letto contiene in `key` il valore -1, allora il record è fittizio e segnala la fine dei record nella regione di memoria; in questo caso viene restituito il valore `NULL`. Si osservi che in base al testo dell'esercizio si può assumere che la regione di memoria contiene *sempre* un record finale con `key` uguale a -1, quindi è stato omesso il codice di controllo per evitare che `*preg` superi il limite della regione di memoria condivisa.

Per inserire gli elementi nella lista dinamica si utilizza `insert_sorted_list()`, simile alle funzioni omonime descritte precedentemente:

```
void insert_sorted_list(struct listel *new,
                        struct listel **pnext)
{
    struct listel *p;
    for (p= *pnext; p!=NULL; pnext= &p->next, p= p->next) {
        if (p->rec.key == new->rec.key) {
            p->rec.value += new->rec.value;
            free(new);
            return;
        }
        if (p->rec.key > new->rec.key) {
            /* inserisci new dopo pnext, prima di p */
            insert_after_node(new, pnext);
            return;
        }
    }
    /* l'elemento va inserito in fondo alla lista */
    insert_after_node(new, pnext);
}
```

Nel caso in cui si determini che la lista contiene già un elemento con lo stessa chiave di quello nuovo, il valore `value` dell'elemento pre-esistente viene aumentato, poi si de-alloca la memoria occupata dall'elemento nuovo.

Infine per stampare la lista dinamica si utilizza la funzione `print_list()`:

```
void print_list(struct listel *p)
{
    while (p != NULL) {
        printf("%d : %g\n", p->rec.key, p->rec.value);
        p = p->next;
    }
}
```