Sistemi Operativi (M. Cesati)

Compito scritto del 22 settembre 2015 (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. 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 che legge da una regione di memoria condivisa IPC di dimensione pari a 4 pagine e la cui chiave è ricavabile utilizzando il percorso `"."` ed il codice `'1'`. La regione di memoria contiene un insieme di dati costituito da un numero variabile di strutture del seguente tipo:

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

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. La lunghezza del vettore di caratteri `name` è memorizzata nel corrispondente campo `length`. I record non hanno un ordine particolare nella regione di memoria. Il valore di `key` è sempre un numero maggiore di zero, tranne per il fatto che l'ultimo record significativo nella regione è seguito da un record in cui `key` ha il valore speciale 0.

L'applicazione deve inserire i record letti dalla regione in una lista dinamica ordinata in base al valore del campo `key`. Diversi record nella regione di memoria possono avere la stessa chiave `key`; tali record corrispondenti allo stesso valore di `key` debbono essere rappresentati da un unico elemento nella lista; il campo `name` associato a tale elemento nella lista deve essere la concatenazione delle stringhe nei campi `name` 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

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

Sistemi Operativi **(M. Cesati)**

Compito scritto del 22 settembre 2015 (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. 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 costituita da 3 processi P_1, P_2 e P_3 collegati tra loro mediante 3 pipe A, B e C secondo lo schema:

$$\text{(stdin)} \longrightarrow P_1 \xrightarrow{A} P_2 \xrightarrow{B} P_3 \xrightarrow{C} P_1 \longrightarrow \text{(stdout)}$$

- Il processo P_1 continuamente legge linee di testo dallo standard input, ciascuna contenente un numero intero senza segno in base 10. Per ciascun numero letto:
  - converte il numero in formato binario a 32 bit e lo invia a P_2 tramite la pipe A;
  - attende una lista di coppie *(primo,esponente)* di numeri in formato binario a 32 bit dalla pipe C e la scrive in standard output in formato testuale.
- Il processo P_2 continuamente legge numeri in formato binario a 32 bit dalla pipe A; per ciascuno di essi, fattorizza il numero inviando al processo P_3 i fattori primi elementari in formato binario a 32 bit per mezzo della pipe B.
- Il processo P_3 continuamente legge dalla pipe B una lista di primi (in ordine non decrescente) e ritrasmette al processo P_1 la equivalente lista nel formato *(primo,esponente)*.

Il formato binario è quello nativo del calcolatore. Le liste di numeri trasmesse sulle pipe B e C sono terminate dal valore zero. Ad esempio:

- P_1 legge in standard input una linea con il numero "`365904`"
- P_1 invia sulla pipe A: [365904] (in formato binario a 32 bit);
- P_2 legge da A il numero inviato da P_1 e scrive su B i numeri

  [2] [2] [2] [2] [3] [3] [3] [7] [11] [11] [0] (in formato binario);
- P_3 legge da B la lista di numeri inviata da P_2 e scrive su C la lista di numeri

  [2] [4] [3] [3] [7] [1] [11] [2] [0] (in formato binario);
- P_1 legge dalla pipe C la lista di numeri inviata da P_3 e scrive sullo standard output la stringa di testo "`2^4 * 3^3 * 7 * 11^2`".

Esercizio 2

Si descrivano in modo chiaro ed esauriente le finalità ed i principi di funzionamento del componente del nucleo conosciuto come "Virtual File System".

Sistemi Operativi (M. Cesati)

Esempio di programmi del compito scritto del 22 settembre 2015

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>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>

struct record {
    unsigned int key;
    int length;
    char name[];
};

struct listel {
    struct listel *next;
    unsigned int key;
    int length;
    char *name;
};
```

Il campo `name` in `record` è un vettore di dimensione non specificata: in pratica, poiché appare come ultimo campo della struttura, il record può avere una lunghezza variabile.

La struttura `listel` rappresenta i nodi della lista contenente i record letti dalla regione di memoria IPC. Oltre al puntatore alla struttura `listel` successiva nella lista, vi sono i valori di `key` e `length`, nonché un puntatore ad una stringa corrispondente a `name`. Si noti la differenza sostanziale tra la struttura `record` e la struttura `listel`: nel primo caso i caratteri di `name` sono posizionati in memoria subito dopo il campo `length`; nel secondo caso invece la struttura include un puntatore ad un'area di memoria che contiene i caratteri di `name`. Usare il puntatore in `listel` consente di semplificare il meccanismo di concatenazione delle stringhe di caratteri aventi la stessa chiave.

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;
    page_size = sysconf(_SC_PAGESIZE);
    if (page_size == -1)
        error_exit("sysconf");
    key = ftok(".", '1');
    if (key == -1)
        error_exit("ftok");
    shmid = shmget(key, 4*page_size, 0);
    if (shmid == -1)
        error_exit("shmget");
    reg = shmat(shmid, NULL, 0);
    if (reg == (void *) -1)
        error_exit("shmat");
    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.

In caso di errore restituito da una chiamata di sistema il programma viene terminato con la funzione `error_exit()`:

```
void error_exit(const char *msg)
{
    perror(msg);
    exit(EXIT_FAILURE);
}
```

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);
        if (lep == NULL)
            break;
        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 record *r = *preg;
    struct listel *p;
    if (r->key == 0)
        return NULL;
    p = alloc_node(r->length);
    p->next = NULL;
    p->key = r->key;
    p->length = r->length;
    p->name =  strndup(r->name, r->length);
    *(char *)preg += sizeof(*r) + r->length;
    return p;
}
```

Se il campo `key` del record corrente nella regione è uguale a zero, la funzione termina restituendo `NULL`: i record nella regione sono terminati. 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 0, quindi è stato omesso il codice

di controllo per evitare che `*preg` superi il limite della regione di memoria condivisa.

Per allocare un nuovo elemento della lista viene invocata `alloc_node()` passando come argomento la lunghezza del vettore di caratteri indicata dal campo `length`. Poi l'elemento della lista viene inizializzato. Per copiare il vettore di caratteri si utilizza la funzione di libreria `strndup()`, che alloca memoria per una stringa C con una lunghezza massima prefissata e vi copia il contenuto della stringa passata come argomento (nel testo non è specificato se la stringa di caratteri in `name` è sempre terminata da '`\0`').

Infine la funzione aggiorna il puntatore alla posizione corrente nella regione di memoria: si considera con un cast l'indirizzo precedente come un puntatore a carattere, e si aggiunge la dimensione in byte della struttura `record` e del vettore `name`.

La funzione `alloc_node()` alloca la memoria dinamica necessaria per il nuovo elemento della lista:

```
struct listel *alloc_node(size_t strl)
{
    struct listel *p;
    p = malloc(sizeof(struct listel));
    if (p == NULL)
        error_exit("malloc");
    p->name = malloc(strl+1); /* +1 for '\0' */
    if (p->name == NULL)
        error_exit("malloc");
    p->length=strl+1;
    return p;
}
```

Le stringhe negli elementi della lista sono sempre terminate da '`\0`', quindi è necessario allocare un byte in più rispetto alla dimensione restituita da `strlen()`.

Per inserire gli elementi nella lista dinamica `build_list()` utilizza la funzione `insert_sorted_list()`:

```
void insert_sorted_list(struct listel *new,
                        struct listel **pnext)
{
    struct listel *p;
    char *q;
    for (p=*pnext; p!=NULL; pnext=&p->next, p=p->next) {
        if (p->key == new->key) {
            q = malloc(p->length+new->length-1);
            if (q == NULL)
                error_exit("malloc");
            strncpy(q, p->name, p->length);
```

```
            strncat(q, new->name, new->length+1);
            free(p->name);
            p->name = q;
            p->length = p->length+new->length-1;
            free(new->name);
            free(new);
            return;
        }
        if (p->key > new->key) {
            insert_after_node(new, pnext);
            return;
        }
    }
    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, la stringa nel nuovo elemento viene concatenata a quella già presente nel vecchio elemento. Ciò comporta la allocazione dinamica di un nuovo vettore di caratteri di dimensioni pari alla somma delle lunghezze precedenti (meno 1, che corrisponde al terminatore '\0' della vecchia stringa). La memoria occupata dalla stringa nell'elemento in lista, dalla stringa nel nuovo elemento, e dal nuovo elemento stesso viene infine rilasciata.

La funzione `insert_after_node()` effettua invece l'inserimento nella lista di un nuovo elemento dopo l'elemento specificato:

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

Infine per stampare la lista dinamica `main()` invoca la funzione `print_list()`:

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

Esercizio 1 — Turno 2

Svolgiamo l'esercizio secondo un approccio "top-down". Le strutture di dati essenziali del programma sono le tre pipe di comunicazione tra i tre processi. Per loro natura, queste pipe debbono essere create dal primo processo ed ereditate dai processi figli. Quindi l'ordine delle operazioni per il processo padre è: (1) creare le pipe, (2) creare i figli, (3) chiudere i descrittori dei file non utilizzati, e (4) eseguire il lavoro proprio di P_1.

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

int pipeA[2]; /* data dir: from P1 to P2 */
int pipeB[2]; /* data dir: from P2 to P3 */
int pipeC[2]; /* data dir: from P3 to P1 */

int main(int argc, char *argv[])
{
    if (argc != 1) {
        fprintf(stderr, "Usage: %s (no arguments)\n",
                argv[0]);
        return EXIT_FAILURE;
    }
    make_pipes(pipeA, pipeB, pipeC);
    spawn_children();
    close_fds(pipeC[1], pipeA[0], pipeB[0], pipeB[1]);
    do_P1_work(pipeC[0], pipeA[1]);
    return EXIT_SUCCESS;
}
```

La funzione `make_pipes()` crea le tre pipe A, B e C:

```c
void make_pipes(int pipeA[2], int pipeB[2], int pipeC[2])
{
    if (pipe(pipeA) || pipe(pipeB) || pipe(pipeC)) {
        perror("creating pipes");
        exit(EXIT_FAILURE);
    }
}
```

La funzione `spawn_children()` crea ed avvia l'esecuzione dei processi figli:

```
void spawn_children(void)
{
    pid_t pid = fork();
    if (pid == -1) {
        perror("fork(P2)");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        close_fds(pipeA[1], pipeB[0], pipeC[0], pipeC[1]);
        do_P2_work(pipeA[0], pipeB[1]);
    }
    pid = fork();
    if (pid == -1) {
        perror("fork(P3)");
        exit(EXIT_FAILURE);
    }
    if (pid == 0) {
        close_fds(pipeB[1], pipeC[0], pipeA[0], pipeA[1]);
        do_P3_work(pipeB[0], pipeC[1]);
    }
}
```

La funzione `close_fds()` chiude quattro descrittori di file passati in argomento:

```
void close_fds(int fd1, int fd2, int fd3, int fd4)
{
    if (close(fd1) || close(fd2) ||
        close(fd3) || close(fd4)) {
        perror("close");
        exit(EXIT_FAILURE);
    }
}
```

Le funzioni `do_P1_work()`, `do_P2_work()` e `do_P3_work()` sono specifiche per ciascuno dei tre processi.

`do_P1_work()` continuamente legge righe di testo dallo standard input. Per ciascuna linea, interpreta il suo contenuto come un numero senza segno in base 10, lo converte in formato nativo (binario) a 32 bit, e lo trasmette sulla pipe A al processo P_2. Infine legge una lista di numeri dalla pipe C, e li stampa nel formato opportuno in standard output:

```
void do_P1_work(int piper, int pipew)
{
    unsigned int v, w, flag;
    for (;;) {
        v = read_number_from_stdin();
        write_u32_to_pipe(v, pipew);
```

```
        if (v == 0) /* EOF in stdin */
            return;
        for (flag=0; ;flag=1) {
            v = read_u32_from_pipe(piper);
            if (v == 0)
                break; /* end of list */
            w = read_u32_from_pipe(piper);
            if (v == 0) {
                fprintf(stderr,
                    "Unexpected end of list from pipe\n");
                exit(EXIT_FAILURE);
            }
            if (flag)
                printf(" * ");
            if (w > 1)
                printf("%u^%u", v, w);
            else
                printf("%u", v);
        }
        fputc('\n', stdout);
    }
}
```

La funzione `read_number_from_stdin()` legge una linea di testo dallo standard input ed interpreta il contenuto come un numero in base 10 utilizzando la funzione `convert_number()`:

```
unsigned int read_number_from_stdin(void)
{
    const int max_line = 1024;
    char line[max_line];
    unsigned int v;
    if (fgets(line, max_line, stdin) == NULL) {
        if (feof(stdin))
            return 0;
        fprintf(stderr, "Error reading from stdin\n");
        exit(EXIT_FAILURE);
    }
    v = convert_number(line);
    return v;
}

unsigned int convert_number(char *str)
{
    unsigned int v;
    char *p;
    errno = 0;
    v = (unsigned int) strtoul(str, &p, 10);
    if (errno != 0 || (*p != '\0' && *p != '\n')) {
        fprintf(stderr,
```

```
                "Invalid number in line: %s (%d)\n",
                str, *p);
        exit(EXIT_FAILURE);
    }
    return v;
}
```

La funzione `write_u32_to_pipe()` scrive un numero binario su di una pipe, facendo attenzione ad eventuali scritture terminate prematuramente:

```
void write_u32_to_pipe(unsigned int v, int fd)
{
    int len = sizeof(v);
    char *b = (char *) &v;
    do {
        int rc = write(fd, b, len);
        if (rc == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        len -= rc;
        b += rc;
    } while (len > 0);
}
```

La funzione `read_u32_from_pipe()` è analoga: legge un numero in formato nativo da una pipe:

```
unsigned int read_u32_from_pipe(int fd)
{
    unsigned int v;
    int len = sizeof(v);
    char *b = (char *) &v;
    do {
        int rc = read(fd, b, len);
        if (rc == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        len -= rc;
        b += rc;
    } while (len > 0);
    return v;
}
```

La funzione `do_P2_work()` esegue il lavoro richiesto al processo P_2: legge un numero dalla pipe A e scrive sulla pipe B la lista dei suoi fattori primi elementari:

```
void do_P2_work(int piper, int pipew)
{
    unsigned int v, prime;
    for (;;) {
        v = read_u32_from_pipe(piper);
        if (v == 0) {
            write_u32_to_pipe(0, pipew);
            exit(EXIT_SUCCESS); /* end of work */
        }
        /* factorize v */
        prime = 2;
        while (v > 1) {
            if (v % prime == 0) { /* prime divides v */
                write_u32_to_pipe(prime, pipew);
                v /= prime;
                /* try again the same prime */
            } else
                prime += 1 + (prime & 1);
                    /* skip even numbers */
        }
        write_u32_to_pipe(0, pipew); /* end of list */
    }
}
```

Si noti che ricevere il numero 0 dalla pipe A viene interpretato come segnale di terminazione per il processo. Il primo fattore provato è 2, poi vengono provati tutti i numeri dispari (questo è ovviamente un algoritmo di fattorizzazione molto inefficiente, ma adeguato per gli scopi di questo esercizio).

La funzione `do_P3_work()` esegue il lavoro richiesto al processo P_3: legge una lista di fattori primi elementari dalla pipe B e scrive sulla pipe C una lista equivalente nel formato *primo, esponente, primo, esponente, ...* 0.

```
void do_P3_work(int piper, int pipew)
{
    unsigned int v, w, exp;
    for (;;) {
        v = read_u32_from_pipe(piper);
        if (v == 0)
            exit(EXIT_SUCCESS);
        exp = 1;
        do {
            w = read_u32_from_pipe(piper);
            if (v == w)
                exp++;
            if (v != w) {
```

```
                write_u32_to_pipe(v, pipew);
                write_u32_to_pipe(exp, pipew);
                exp = 1;
            }
            v = w;
        } while (w != 0);
        write_u32_to_pipe(0, pipew); /* end of list */
    }
}
```

Anche in questo caso ricevere come primo numero di una lista il valore 0 viene intepretato dal processo P_3 come segnale di terminazione.