Sistemi Operativi **(M. Cesati)**

Compito scritto del 21 settembre 2017

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

Scrivere una applicazione C/POSIX multiprocesso costituita da due processi P_1 e P_2. Il processo P_1 legge il contenuto di un file il cui nome è passato sulla linea comando. Il file contiene un numero finito e variabile di strutture del seguente tipo:

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

Il formato dei dati è quello nativo per l'architettura del calcolatore; in altre parole, la sequenza di byte nel file corrisponde alla rappresentazione in RAM delle varie strutture record, una dopo l'altra. I record entro il file non sono ordinati in alcun modo particolare.

Il processo P_1 legge dal file un record alla volta e lo inserisce in un buffer circolare condiviso con P_2. Il processo P_2 legge i record dal buffer circolare e li inserisce in una lista ordinata in base al valore del campo `key`.

Al termine, il processo P_1 scrive 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'`.

Si considerino gli eventuali problemi legati alle race condition ed alla sincronizzazione tra i due processi.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 21 settembre 2017

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo col definire le strutture di dati principali utilizzate dal programma:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <errno.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#include <sys/msg.h>
#include <semaphore.h>

struct record {
    int key;
    char name[32];
};
#define circbuf_size 17
struct circbuf {
    int E, S, fin;
    sem_t sem_write, sem_read;
    struct record rec[circbuf_size];
};
struct list_t {
    struct record rec;
    struct list_t *next;
};
struct msgbuf {
    long mtype;
    struct list_t elem;
};
```

La struttura `record` è come definita nel testo dell'esercizio. La struttura `circbuf` implementa il buffer circolare: i campi `E` e `S` indicano la prima locazione vuota e la prima locazione ancora da leggere, rispettivamente. Il semaforo POSIX `sem_write` contiene il numero di locazioni libere nel buffer circolare, mentre il semaforo `sem_read` contiene il numero di locazioni occupate. Il campo `fin` è un flag impostato ad uno quando il processo P_1 ha terminato di inserire i record nel buffer. La struttura `msgbuf` è utilizzata per trasferire un elemento della lista in una coda di messaggi.

La funzione `main()` è la seguente:

```
int main(int argc, char *argv[])
{
    struct circbuf *cb;
    int mq;
    struct list_t *lh;
    abort_on_error(argc != 2,
                   "Missing filename in command line");
    cb = create_shared_memory();
    initialize_circbuf(cb);
    mq = create_message_queue();
    spawn_process_P2(cb, mq);
    read_file(argv[1], cb);
    notify_end_of_work(cb);
    lh = get_list_from_P2(mq);
    print_sorted_list(lh);
    return EXIT_SUCCESS;
}
```

Il processo P_1 è il processo iniziale dell'applicazione, ossia quello che esegue la funzione `main()`. Inizialmente il processo controlla il numero di argomenti passato sulla linea comando, ed eventualmente termina prematuramente tramite la macro `abort_on_error`:

```
#define abort_on_error(cond, msg) do { \
    if (cond) { \
        fprintf(stderr, \
                "%s (errno=%d [%m])\n ", msg, errno); \
        exit(EXIT_FAILURE); \
    } \
} while (0)
```

Il buffer circolare deve trovarsi in un'area di memoria a cui entrambi i processi possano accedere. Si è scelto di utilizzare un segmento di memoria condivisa IPC:

```
struct circbuf *create_shared_memory(void)
{
    void *p;
    int d = shmget(IPC_PRIVATE, sizeof(struct circbuf),
                   0600);
    abort_on_error(d == -1,
                   "Cannot create a shared memory segment");
    p = shmat(d, NULL, 0);
    abort_on_error(p == (void *)-1,
                   "Cannot attach a shared memory segment");
    return p;
}
```

La funzione `main()` inizializza il buffer circolare invocando `initialize_circbuf()`:

```
void initialize_circbuf(struct circbuf *cb)
{
    int rc;
    cb->E = cb->S = cb->fin = 0;
    rc = sem_init(&cb->sem_write, 1, circbuf_size-1);
    abort_on_error(rc == -1,
                   "Cannot initialize the semaphore");
    rc = sem_init(&cb->sem_read, 1, 0);
    abort_on_error(rc == -1,
                   "Cannot initialize the semaphore");
}
```

Si osservi che il semaforo `sem_write` è impostato alla dimensione reale del buffer meno uno: infatti, la logica del programma dovrà lasciare sempre almeno una locazione del buffer circolare libera per evitare che la condizione di uguaglianza dei campi `E` e `S` possa verificarsi nel caso di buffer completamente pieno. Il semaforo `sem_read` è impostato a zero, in quanto inizialmente il buffer è vuoto.

È necessario anche definire una coda di messaggi tra P_1 e P_2, in quanto P_2 deve trasferire la lista di elementi ordinata a P_1. A tale scopo `main()` invoca la funzione `create_message_queue()`:

```
int create_message_queue(void)
{
    int q = msgget(IPC_PRIVATE, 0600);
    abort_on_error(q == -1, "Cannot create message queue");
    return q;
}
```

Successivamente `main()` invoca `spawn_process_P2()` per creare il processo P_2.

```
void spawn_process_P2(struct circbuf *cb, int mq)
{
    pid_t p = fork();
    abort_on_error(p == -1, "Cannot fork a new process");
    if (p == 0) {
        do_P2_work(cb, mq);
        exit(EXIT_SUCCESS);
    }
}
```

Esamineremo in seguito la funzione `do_P2_work()` eseguita dal processo P_2.

La funzione `main()` passa ad eseguire il lavoro del processo P_1 invocando la funzione `read_file()`:

```
void read_file(const char *fname, struct circbuf *cb)
{
    struct record rec;
    FILE *f = fopen(fname, "r");
    if (f == NULL) {
        perror(fname);
        exit(EXIT_FAILURE);
    }
    while (!feof(f)) {
        if (get_record(f, &rec))
            circbuf_in(&rec, cb);
        if (ferror(f)) {
            perror(fname);
            exit(EXIT_FAILURE);
        }
    }
}
```

Dopo aver aperto il file, `read_file()` legge un record alla volta invocando `get_record()`:

```
int get_record(FILE *f, struct record *buf)
{
    size_t rc = fread(buf, sizeof(struct record), 1, f);
    return (rc == 1);
}
```

Il record viene inserito nel buffer circolare con la funzione `circbuf_in()`:

```
void circbuf_in(struct record *rec, struct circbuf *cb)
{
    int nE = (cb->E + 1) % circbuf_size;
    int rc = sem_wait(&cb->sem_write);
    abort_on_error(rc == -1,
                   "Waiting on write sem failed");
    cb->rec[cb->E] = *rec;
    cb->E = nE;
    rc = sem_post(&cb->sem_read);
    abort_on_error(rc == -1,
                   "Increment of the read sem failed");
}
```

La funzione controlla che il buffer non sia completamente pieno tentando di acquisire il semaforo `sem_write`. Se l'acquisizione riesce esiste almeno una locazione libera nel buffer (e di conseguenza è anche garantito che `nE` è diverso da `cb->S`, poiché il valore massimo di `sem_write` è pari alla dimensione del buffer meno uno). Dopo aver inserito il nuovo record nel buffer la funzione segnala l'esistenza di una nuova posizione occupata nel buffer incrementando il semaforo `sem_read`.

Terminata la lettura del file la funzione `read_file()` ritorna, e `main()` invoca la funzione `notify_end_of_work()`:

```
void notify_end_of_work(struct circbuf *cb)
{
    int rc;
    cb->fin = 1;
    rc = sem_post(&cb->sem_read);
    abort_on_error(rc == -1,
                   "Final increment of read sem failed");
}
```

Dopo aver impostato il flag di terminazione viene incrementato di uno il valore del semaforo `sem_read`: in questo modo viene evitato un possibile stallo qualora P_1 impostasse ad uno `fin` in un momento intermedio tra il controllo di P_2 di `fin` e l'invocazione di `sem_wait()`.

Il processo P_1 passa poi a leggere la lista ordinata inviata da P_2 per mezzo della funzione `get_list_from_P2()`:

```
struct list_t * get_list_from_P2(int mq)
{
    struct list_t *p, *head, **pnext;
    struct msgbuf mbuf;
    pnext = &head;
    do {
        int rc = msgrcv(mq, &mbuf, sizeof(mbuf.elem), 0,0);
        abort_on_error(rc == -1,
                       "Cannot get a list element");
        p = malloc(sizeof(struct list_t));
        abort_on_error(!p, "Memory allocation error");
        *p = mbuf.elem;
        *pnext = p;
        pnext = &p->next;
    } while (*pnext);
    return head;
}
```

Si osservi come i puntatori della lista in P_2 non possono essere considerati validi nello spazio di memoria di P_1, però il valore `NULL` è identico per tutti e codifica la fine della lista.

Infine `main()` invoca la funzione `print_sorted_list()` per stampare in standard output il contenuto della lista ordinata col formato richiesto dal testo dell'esercizio:

```
void print_sorted_list(struct list_t *h)
{
    int key;
    while (h != NULL) {
        key = h->rec.key;
        printf("%d: %s", key, h->rec.name);
        for (;;) {
            h = h->next;
            if (h == NULL || key != h->rec.key)
                break;
            printf(" %s", h->rec.name);
        }
        printf("\n");
    }
}
```

Il processo P_2 inizia l'esecuzione dalla funzione `do_P2_work()`:

```
void do_P2_work(struct circbuf *cb, int mq)
{
    struct record rec;
    struct list_t *list_head = NULL;

    while (cb->fin == 0 || cb->E != cb->S) {
        circbuf_out(&rec, cb);
        insert_in_ordered_list(&rec, &list_head);
    }
    send_list_to_P1(list_head, mq);
}
```

Il processo P_2 esegue un ciclo che termina soltanto quando il buffer circolare risulta vuoto e il processo P_1 ha segnalato la fine degli inserimenti. In ogni iterazione del ciclo un record viene estratto dal buffer circolare ed inserito nella lista ordinata. La funzione `circbuf_out()` estrae un elemento dal buffer circolare:

```
void circbuf_out(struct record *rec, struct circbuf *cb)
{
    int rc = sem_wait(&cb->sem_read);
    abort_on_error(rc == -1,
                   "Waiting on read sem failed");
    if (cb->E == cb->S)
        return;
    *rec = cb->rec[cb->S];
    cb->S = (cb->S + 1) % circbuf_size;
    rc = sem_post(&cb->sem_write);
    abort_on_error(rc == -1,
                   "Increment of write sem failed");
}
```

La funzione tenta di acquisire il semaforo `sem_read`, pertanto P_2 viene bloccato se il buffer è vuoto fino al momento in cui P_1 non inserisce un nuovo elemento. La condizione `cb->E == cb->S` dopo la `sem_wait()` è in effetti sempre falsa tranne che nel caso speciale in cui P_1 ha impostato `fin` a uno. Dopo aver copiato il record la funzione incrementa il semaforo `sem_write` poiché una locazione del buffer è stata liberata.

Per inserire l'elemento nella lista ordinata `do_P2_work()` invoca la funzione `insert_in_ordered_list()`:

```
void insert_in_ordered_list(struct record *rec,
                            struct list_t **phead)
{
    int key = rec->key;
    struct list_t *new = malloc(sizeof(struct list_t));
    abort_on_error(!new, "Memory allocation failure");
    new->rec = *rec;
    new->next = NULL;
    while (*phead != NULL && key >= (*phead)->rec.key)
        phead = &(*phead)->next;
    insert_after_node(new, phead);
}
```

È necessario allocare un nuovo elemento della lista ordinata, e percorrere la lista fino a determinare la posizione in base al valore del campo `key`. Come al solito, la funzione di manipolazione della lista `insert_after_node()` è basata sull'indirizzo del campo `next` dei vari elementi in modo da gestire in modo uniforme anche il caso della lista vuota:

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

```
    *pnext = new;
}
```

Da ultimo, P_2 esegue `send_list_to_P1()` per inviare la lista ordinata al processo P_1:

```
void send_list_to_P1(struct list_t *lh, int mq)
{
    struct msgbuf mbuf;
    int rc;
    while (lh != NULL) {
        mbuf.mtype = 1;
        mbuf.elem = *lh;
        rc = msgsnd(mq, &mbuf, sizeof(mbuf.elem), 0);
        abort_on_error(rc == -1,
                       "Cannot send a list element");
        lh = lh->next;
    }
}
```