Sistemi Operativi (M. Cesati)

Compito scritto del 2 luglio 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

Un file contiene record di lunghezza variabile memorizzati con il seguente formato:

$$n_1 \texttt{ record1 } n_2 \texttt{ record2 } n_3 \texttt{ record3 } \cdots$$

ove n_i rappresenta la dimensione in byte dell'i-esimo record nel formato nativo del calcolatore, e `recordi` è la sequenza di byte corrispondente al contenuto dell'i-esimo record. Scrivere una applicazione C/POSIX costituita da due processi che comunicano tramite una pipe. Il primo processo legge il file ed trasmette sulla pipe al secondo processo un record alla volta rovesciando l'ordine dei suoi byte. Il secondo processo deve scrivere sullo standard output il contenuto dei record ricevuti, evidenziando chiaramente l'inizio e la fine di ciascun record.

Esercizio 2

Si descriva in modo chiaro ed esauriente lo scopo ed i principi di funzionamento dell'algoritmo *slab allocator* utilizzato nei moderni sistemi operativi.

Sistemi Operativi (M. Cesati)

Compito scritto del 2 luglio 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 Un file contiene record di lunghezza variabile memorizzati con il seguente formato:

$$q\ n_1\ n_2 \cdots n_q\ \texttt{record1}\ \texttt{record2} \cdots \texttt{record}q$$

ove q rappresenta il numero di record nel file nel formato nativo del calcolatore, n_i rappresenta la dimensione in byte dell'i-esimo record nel formato nativo del calcolatore, e `record`i è la sequenza di byte corrispondente al contenuto dell'i-esimo record. Scrivere una applicazione C/POSIX costituita da 100 processi che mappano in memoria il file e ne riscrivono il suo contenuto rovesciando l'ordine dei byte di ciascun record. Il processo j-esimo deve rovesciare l'ordine del record j, del record $100 + j$, del record $200 + j$, ecc. Pertanto il primo processo deve rovesciare l'ordine del record 1, del record 101, del record 201, ecc.; il secondo processo deve rovesciare l'ordine del record 2, del record 102, del record 202, ...; e così via.

Esercizio 2

Si descrivano in modo chiaro ed esauriente le operazioni svolte dal nucleo di un moderno sistema operativo in conseguenza di una eccezione di pagina (*page fault*) avvenuta durante l'esecuzione di un processo in User Mode.

Sistemi Operativi (M. Cesati)

Esempi dei programmi del compito scritto del 2 luglio 2015

Turno 1

Risolviamo l'esercizio con un approccio top-down. Le operazioni principali da svolgere nella funzione `main()` sono: creazione della pipe, creazione di un processo figlio, apertura e processamento del file di ingresso.

Poiché il testo dell'esercizio non specifica la modalità con cui il file di input viene selezionato, scegliamo di leggerne il nome dalla linea comando.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>

int main(int argc, char *argv[])
{
    int pfd[2], fd;
    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return EXIT_FAILURE;
    }
    create_pipe(pfd);
    spawn_child(pfd);
    close_fd(pfd[0]);
    fd = open_file(argv[1]);
    send_records(fd, pfd[1]);
    close_fd(pfd[1]);
    close_fd(fd);
    return EXIT_SUCCESS;
}
```

Dopo aver controllato il numero di parametri in ingresso, `main()` invoca la funzione `create_pipe()` per creare una pipe:

```
void create_pipe(int pfd[2])
{
    int rc = pipe(pfd);
    if (rc == -1) {
        perror("pipe");
        exit(EXIT_FAILURE);
    }
}
```

Successivamente `main()` invoca la funzione `spawn_child()` per creare un processo figlio:

```
void spawn_child(int pfd[2])
{
    pid_t p = fork();
    if (p == -1) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    if (p != 0)
        return;
    print_records(pfd);
    /* NEVER REACHED */
}
```

Dal punto di vista del processo figlio, la funzione `spawn_child()` non termina mai. Sospendiamo momentaneamente l'analisi del processo figlio, e continuiamo invece l'analisi della funzione `main()`.

Assegniamo al padre il compito di leggere i record dal file di ingresso, invertirli, e di scriverli sulla pipe. Pertanto, nella funzione `main()` viene invocata la funzione `close_fd()` per chiudere il descrittore di file associato al terminale di lettura della pipe:

```
void close_fd(int fd)
{
    if (close(fd) == -1) {
        perror("close");
        exit(EXIT_FAILURE);
    }
}
```

Successivamente in `main()` viene invocata la funzione `open_file()` per aprire il file di ingresso:

```
int open_file(const char *name)
{
    int fd = open(name, O_RDWR);
    if (fd == -1) {
        perror(name);
        exit(EXIT_FAILURE);
    }
    return fd;
}
```

Per processare i record contenuti nel file di ingresso si utilizza la funzione `send_records()`, a cui vengono passati i descrittori del file di ingresso e del terminale di scrittura della pipe:

```
void send_records(int ifd, int ofd)
{
    for (;;) {
        char *buf;
        int n = read_int(ifd);
        if (n == 0)
            return;
        buf = read_record(ifd, n);
        invert_bytes(buf, n);
        write_buf(ofd, (char *)&n, sizeof(n));
        write_buf(ofd, buf, n);
        free(buf);
    }
}
```

La funzione legge un intero dal file, corrispondente alla dimensione del successivo record. Poi legge il record stesso, inverte i suoi byte, e scrive il record sulla pipe.

La funzione `read_int()` legge un intero in formato nativo del calcolatore dal file:

```
int read_int(int fd)
{
    size_t len = sizeof(int);
    int rc, v;
    char *p = (char *) &v;
    do {
        rc = read(fd, p, len);
        if (rc == -1) {
            perror(NULL);
            exit(EXIT_FAILURE);
        }
        if (rc == 0) {
            if (len != sizeof(int))
                fprintf(stderr, "Unexpected end of file\n")
                    ;
            return 0;
        }
        len -= rc;
        p += rc;
    } while (len > 0);
    return v;
}
```

La funzione legge una sequenza di `sizeof(int)` byte dal file, e la memorizza in una variabile di tipo `int`. Nel far ciò, si considerano eventuali letture "corte" che restituiscono meno byte di quanti effettivamente richiesti. Se il file non termina in corrispondenza della fine di un record viene mostrato un messaggio di avvertimento, in quanto il formato del file non è quello previsto dal programma.

La funzione `read_record()` viene invece utilizzata per leggere dal file di ingresso un record di lunghezza conosciuta:

```
char * read_record(int fd, int size)
{
    char *p, *q;
    p = q = malloc(sizeof(char)*size);
    if (p == NULL) {
        perror("malloc");
        exit(EXIT_FAILURE);
    }
    while (size > 0) {
        int rc = read(fd, q, size);
        if (rc == -1) {
            perror("read");
            exit(EXIT_FAILURE);
        }
        if (rc == 0) {
            fprintf(stderr, "Unexpected end of file\n");
            exit(EXIT_FAILURE);
        }
        q += rc;
        size -= rc;
    }
    return p;
}
```

La funzione dapprima alloca dinamicamente un buffer in grado di memorizzare l'intero record. Successivamente il buffer viene letto dal file, gestendo in modo analogo a quanto fatto in `read_int()` le letture "corte" e la fine prematura del file di ingresso.

Per invertire l'ordine dei byte del record `send_records()` invoca la funzione `invert_bytes()`:

```
void invert_bytes(char *rec, int len)
{
    char *last;
    for (last=rec+(len-1); rec < last; ++rec, --last) {
        char c = *rec;
        *rec = *last;
        *last = c;
    }
```

```
}
```

Successivamente, `send_records()` invoca due volte la funzione `write_buf()` per scrivere sulla pipe dapprima la dimensione del record, e poi il record stesso (invertito). A questo proposito si deve osservare che in linea generale non è ammissibile nello svolgimento dell'esercizio trascurare che la dimensione del record è indicata esplicitamente nel file stesso insieme al record. A titolo di esempio, non è ammissibile imporre una dimensione massima per il record, oppure imporre un formato al suo contenuto, quale una stringa di testo terminata da "`\n`" oppure da "`\0`". Di conseguenza, il modo più semplice per realizzare l'applicazione richiesta è trasmettere sulla pipe la dimensione del record prima del record stesso.

```
void write_buf(int fd, char *rec, int size)
{
    while (size > 0) {
        int rc = write(fd, rec, size);
        if (rc == -1) {
            perror("write");
            exit(EXIT_FAILURE);
        }
        rec += rc;
        size -= rc;
    }
}
```

La funzione `send_records()` termina l'esecuzione dell'iterazione del ciclo liberando la memoria dinamica utilizzata per memorizzare il contenuto del buffer. L'esecuzione ritorna in `main()` soltanto dopo aver terminato il processamento dell'intero file di ingresso. `main()` invoca due volte `close_fd()` per chiudere i descrittori del file di ingresso e del terminale di scrittura della pipe, e termina l'esecuzione del processo.

Torniamo ora ad analizzare il codice eseguito dal processo figlio. In questo caso `spawn_child()` invoca la funzione `print_records()`:

```
void print_records(int pfd[2])
{
    char *buf;
    close_fd(pfd[1]);
    for (;;) {
        int n = read_int(pfd[0]);
        if (n == 0)
            exit(EXIT_SUCCESS);
        buf = read_record(pfd[0], n);
        printf("--- START OF RECORD ---\n");
```

```
        if (fwrite(buf, n, 1, stdout) != 1u) {
            perror("stdout");
            exit(EXIT_FAILURE);
        }
        printf("\n--- END  OF RECORD ---\n");
        free(buf);
    }
}
```

La funzione chiude il terminale di scrittura della pipe, ed entra in un ciclo in cui legge dalla pipe la dimensione di un record, poi legge il record stesso memorizzandolo in un buffer allocato dinamicamente, stampa il contenuto del buffer in standard output, e libera la memoria occupata dal buffer. Si noti che il record viene scritto in standard output per mezzo di `fwrite()`, in quanto non è possibile assumere che il record stesso contenga testo stampabile.

Il processo figlio termina dopo aver riconosciuto la condizione EOF sulla pipe, ovvero in seguito alla chiusura della pipe da parte del processo padre.

Turno 2

Risolviamo l'esercizio con un approccio top-down. Le operazioni principali da svolgere nella funzione `main()` sono: apertura del file, lettura della dimensione del file, mapping del file in memoria, creazione dei processi, e svolgimento del lavoro di ciascun processo.

Poiché il testo dell'esercizio non specifica la modalità con cui il file di input viene selezionato, scegliamo di leggerne il nome dalla linea comando.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>

int main(int argc, char *argv[])
{
    int fd, p;
    size_t len;
    char *map;

    if (argc != 2) {
        fprintf(stderr, "Usage: %s <file>\n", argv[0]);
        return EXIT_FAILURE;
    }
    fd = open_file(argv[1]);
```

```
    len = compute_file_size(fd);
    map = map_file(fd, len);
    if (close(fd) == -1)
        perror(argv[1]);
    p = fork_children(100);
    invert_records(p+1, map, 100);
    unmap_file(map, len);
    return EXIT_SUCCESS;
}
```

Si noti che dopo aver creato il memory mapping del file è possibile chiudere il file stesso: tutte le successive letture e scritture del contenuto del file sono effettuate accedendo direttamente all'area di memoria associata al memory mapping.

Per aprire il file utilizziamo la funzione `open_file()`:

```
int open_file(const char *name)
{
    int fd = open(name, O_RDWR);
    if (fd == -1) {
        perror(name);
        exit(EXIT_FAILURE);
    }
    return fd;
}
```

Per semplicità scegliamo di mappare il file per intero, prima di creare gli altri processi. A tale scopo abbiamo la necessità di determinare la dimensione totale del file. Possiamo farlo in due modi: tramite le API POSIX (ad esempio, utilizzando `lseek()`), oppure leggendo i valori interi memorizzati all'inizio del file. In questo esempio adottiamo la seconda soluzione:

```
size_t compute_file_size(int fd)
{
    size_t i, q, tot = 0;
    q = read_int(fd);
    for (i=0; i<q; ++i)
        tot += read_int(fd);
    return tot + (1+q)*sizeof(int);
}
```

La funzione legge dapprima il numero di record q di cui è costituito il file, e poi effettua la somma dei successivi q interi, in modo da ottenere la dimensione totale di tutti i record. Infine si calcola la dimensione del file sommando la dimensione dei valori interi all'inizio del file.

Per leggere i valori dal file utilizziamo la funzione `read_int()`:

```
int read_int(int fd)
{
    int rc, v;
    size_t len = sizeof(int);
    char *p = (char *) &v;
    do {
        rc = read(fd, p, len);
        if (rc == -1) {
            perror(NULL);
            exit(EXIT_FAILURE);
        }
        if (rc == 0) {
            fprintf(stderr, "Unexpected end of file\n");
            exit(EXIT_FAILURE);
        }
        len -= rc;
        p += rc;
    } while (len > 0);
    return v;
}
```

La funzione legge una sequenza di `sizeof(int)` byte dal file, e la memorizza in una variabile di tipo `int`. Nel far ciò, si considerano eventuali letture "corte" che restituiscono meno byte di quanti effettivamente richiesti.

Per mappare il file in memoria (sia in lettura che in scrittura) utilizziamo la funzione `map_file()`:

```
char * map_file(int fd, size_t len)
{
    char *p = mmap(NULL, len, PROT_READ|PROT_WRITE,
                   MAP_SHARED, fd, 0);
    if (p == MAP_FAILED) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    return p;
}
```

Per creare i 100 processi utilizziamo la funzione `fork_children()`, che restituisce il numero logico del processo, da 0 (il processo originale) a 99:

```
int fork_children(int n)
{
    int i;
    pid_t p;
    for (i=1; i<n; ++i) {
        p = fork();
        if (p == -1) {
            perror(NULL);
            exit(EXIT_FAILURE);
        }
        if (p == 0)
            return i;
    }
    return 0;
}
```

Il lavoro di inversione del contenuto dei record è affidato alla funzione `invert_records()`, eseguita da ciascuno dei 100 processi. Il primo parametro corrisponde al numero del processo, traslato da 1 a 100. Oltre all'indirizzo dell'area di memoria con il memory mapping viene anche passato il numero di processi, e quindi la distanza tra i record che ciascun processo deve invertire.

```
void invert_records(int p, char *map, int nproc)
{
    int *n = (int *) map;
    int i, j, q = n[0];
    char *rec = map + sizeof(int) * (q+1);
    for (j=1, i=p; i<=q; i+=nproc) {
        for (; j<i; ++j)
            rec += n[j];
        invert_bytes(rec, n[j]);
    }
}
```

La variabile `p` contiene l'indice del processo; la variabile `i` contiene il numero del record che si deve invertire; la variabile `j` serve a ricordare l'ultimo offset aggiunto alla posizione corrente nel file per arrivare alla posizione del record da invertire. Per leggere con facilità i valori interi all'inizio del file, il puntatore `map` viene assegnato con un cast al puntatore ad interi `n`; perciò `n[0]` corrisponde al valore q, `n[1]` al valore n_1, ecc.

Per invertire il record viene utilizzata la funzione `invert_bytes()`, che riceve l'indirizzo iniziale del record e la sua lunghezza in byte:

```
void invert_bytes(char *rec, int len)
{
    char *last;
    for (last=rec+(len-1); rec < last; ++rec, --last) {
        char c = *rec;
        *rec = *last;
        *last = c;
    }
}
```

Infine, prima di terminare il programma, il memory mapping viene rilasciato utilizzando la funzione `unmap_file()`:

```
void unmap_file(char *map, size_t len)
{
    if (munmap(map, len) != 0) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
}
```