Sistemi Operativi (M. Cesati)

Compito scritto del 1 febbraio 2016

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 due processi P_1 e P_2 collegati con una pipe. Il processo P_1 legge dalla linea comando il nome di un file di testo contenente righe con il seguente formato:

`<numero> <operazione> <numero> <operazione> <numero> ...`

ove `<numero>` è la rappresentazione decimale di un numero intero, e `<operazione>` è uno dei caratteri "`+`", "`-`", "`*`", "`/`". Tra ogni elemento della linea (`<numero>` o `<operazione>`) possono essere presenti zero, uno o più caratteri spazio bianco (" ").

Per ciascuna riga del file, il processo P_1 controlla la sua correttezza formale, e invia sulla pipe l'espressione numerica corrispondente con il seguente formato:

`valore` `<operazione>` `valore` `<operazione>` `valore` `...` =

ove `valore` è la codifica del numero corrispondente in formato nativo del calcolatore (`int`), `<operazione>` ha la medesima interpretazione descritta sopra, ed il carattere "`=`" indica la fine dell'espressione.

Il processo P_2 legge dalla pipe ciascuna espressione e scrive in standard output una linea di testo con il risultato corrispondente.

Esercizio 2

Si descrivano in modo chiaro, ordinato ed esaustivo le operazioni logiche effettuate dal nucleo di un moderno sistema operativo Unix-like (ad esempio, Linux) per gestire una richiesta di apertura di un file su disco tramite la chiamata di sistema `open`.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 1 febbraio 2016

Esercizio 1

Svolgiamo l'esercizio seguendo un approccio "top-down". Iniziamo dunque con la funzione `main()`:

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

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

La funzione `main()` controlla il numero di argomenti sulla linea comando, ed invoca le funzioni per creare la pipe, creare il processo figlio, ed avviare l'elaborazione del file di input.

La funzione `create_pipe()` crea la pipe di comunicazione:

```
void create_pipe(int fds[])
{
    int rc = pipe(fds);
    exit_on_error(rc == -1, "pipe");
}
```

La funzione `exit_on_error()` è utilizzata in tutto il programma per controllare l'esito di una API e, se necessario, interrompere l'esecuzione:

```
inline void exit_on_error(int condition,
                          const char *message)
{
    if (condition) {
        perror(message);
        exit(EXIT_FAILURE);
    }
}
```

La funzione `spawn_child()` crea il processo figlio:

```
void spawn_child(int fds[])
{
    int rc;
    pid_t p = fork();
    exit_on_error(p == -1, "fork");
    if (p == 0) {
        rc = close(fds[1]);
        exit_on_error(rc == -1, "close");
        child_task(fds[0]);
        /* never returns here */
    }
    rc = close(fds[0]);
    exit_on_error(rc == -1, "close");
}
```

Nell'esecuzione del processo figlio la funzione chiude il descrittore di uscita della pipe ed invoca la funzione `child_task()`. Quando eseguita dal processo padre, invece, la funzione chiude il descrittore di ingresso della pipe.

La funzione `read_file()`, eseguita dal padre, si occupa di leggere il file di ingresso il cui nome è passato sulla linea comando:

```
void read_file(const char *filename, int pipefd)
{
    FILE *inf = fopen(filename, "r");
    exit_on_error(inf == NULL, filename);
    while (!feof(inf))
        read_and_process_line(inf, pipefd);
}
```

Il file viene aperto in sola lettura, poi viene letta e processata una riga di testo alla volta per mezzo della funzione `read_and_process_line()`:

```
#define MAXLINE 256
void read_and_process_line(FILE *inf, int pfd)
{
    char linebuf[MAXLINE], *p;
    p = fgets(linebuf, MAXLINE, inf);
    if (p == NULL)
        exit_on_error(ferror(inf), "input file");
    else
        process_line(linebuf, pfd);
}
```

Per semplicità scegliamo di limitare la dimensione possibile di ciascuna riga al valore codificato nella macro `MAXLINE`. Dopo aver letto la riga di testo, il controllo della sua correttezza formale è affidato alla funzione `process_line()`:

```
#define MAXNUMBERS (MAXLINE/2)
void process_line(const char *line, int pipefd)
{
    int values[MAXNUMBERS];
    char opers[MAXNUMBERS];
    int ci = 0;
    const char *p = line;

    while (*p != '\n' && *p != '\0') {
        p = read_number(p, values+ci);
        if (p != NULL)
            p = read_operation(p, opers+ci);
        if (p == NULL) {
            fprintf(stderr, "Invalid format in line: %s\n",
                    line);
            return;
        }
        ++ci;
    }
    send_to_pipe(ci, values, opers, pipefd);
}
```

La dimensione massima della riga di testo ci permette di definire anche un limite superiore al numero di valori e di operazioni presenti sulla riga: poiché ogni numero ed ogni operazione consuma almeno un carattere, ci potranno essere soltanto `MAXLINE/2` valori e `MAXLINE/2` operazioni su ciascuna riga.

La funzione legge un numero con `read_number()`; se non vi sono errori, la funzione legge l'operazione seguente con `read_operation()`. Il procedimento è ripetuto finché non si legge il carattere di fine riga o fine stringa.

La funzione `read_number()` si occupa di convertire un numero decimale nel buffer di caratteri in un valore in formato nativo del calcolatore:

```
const char *read_number(const char *buf, int *value)
{
    char *p;

    errno = 0;
    *value = strtol(buf, &p, 10);
    if (errno != 0)
        return NULL;
    return p;
}
```

Si noti che la funzione restituisce il puntatore nel buffer al carattere seguente il numero. La funzione di libreria `strtol()`, tra l'altro, ignora automaticamente ogni sequenza di caratteri bianchi eventualmente presenti prima del numero.

La funzione `read_operation()` si occupa di leggere il successivo carattere che identifica una operazione:

```
const char *read_operation(const char *buf, char *op)
{
    buf = skip_blanks(buf);
    switch (*buf) {
        case '+':
        case '-':
        case '*':
        case '/':
            *op = *buf;
            break;
        case '\n':
        case '\0':
            *op = '=';
            return buf;
        default:
            return NULL;
    }
    return buf+1;
}
```

Si noti che la terminazione della riga (con il carattere di fine stringa o quello di fine riga) viene gestita come una operazione di tipo "=", in accordo a quanto previsto nel testo dell'esercizio. In questo caso inoltre la funzione restituisce l'indirizzo del carattere di terminazione, mentre in ogni altro caso restituisce l'indirizzo del carattere successivo nel buffer.

La funzione `skip_blanks()` ha il compito di passare oltre ogni eventuale carattere bianco presente tra il numero e l'operazione:

```
const char *skip_blanks(const char *buf)
{
    while (*buf == ' ')
        ++buf;
    return buf;
}
```

Per inviare la codifica della espressione aritmetica sulla pipe viene utilizzata la funzione `send_to_pipe()`:

```
void send_to_pipe(int n, int *val, char *ops, int pfd)
{
    int i, rc;
    for (i=0; i<n; ++i) {
        rc = write(pfd, val+i, sizeof(int));
        exit_on_error(rc != sizeof(int), "write");
        rc = write(pfd, ops+i, 1);
        exit_on_error(rc != 1, "write");
    }
}
```

Per semplicità si assume che le scritture siano sempre eseguite in modo completo o non eseguite affatto (ovvero che non vengano eseguite parzialmente). Sebbene in assoluto ciò non sia corretto, è ammissibile in questo caso considerato il numero ridotto di byte trasferito in ciascuna `write()`.

Il cuore delle operazioni eseguite dal processo figlio sono implementate dalla funzione `child_task()`:

```
void child_task(int pfd)
{
    for (;;) {
        int res = 0;
        char o = '+';
        do {
            int v = get_number(pfd);
            switch (o) {
                case '+':
                    res += v;
                    break;
                case '-':
                    res -= v;
                    break;
                case '*':
                    res *= v;
                    break;
                case '/':
                    res /= v;
                    break;
```

```
                default:
                    fprintf(stderr, "Invalid operation (%c)
                        \n", o);
                    exit(EXIT_FAILURE);
            }
            o = get_operation(pfd);
        } while (o != '=');
        printf("%d\n", res);
    }
}
```

Poiché la variabile `o` è inizializzata con `'+'`, il primo numero letto è sommato alla variabile `res`, inizializzata a zero. Non appena viene letto il carattere `'='` il ciclo interno termina, viene stampato su standard output il risultato dell'espressione, e si ricomincia a leggere dalla pipe una nuova espressione.

La funzione legge dalla pipe l'esatto numero di byte che codificano il numero in formato nativo con `get_number()`:

```
int get_number(int fd)
{
    int rc, v;
    rc = read(fd, &v, sizeof(int));
    if (rc == 0)
        exit(EXIT_SUCCESS);
    exit_on_error(rc != sizeof(int), "read");
    return v;
}
```

Quando il processo padre chiude il descrittore di scrittura della pipe ed il figlio ha terminato di leggere tutti i caratteri inviati, la `read()` restituisce il valore zero: in questo caso il processo figlio termina l'esecuzione.

Subito dopo il numero, sulla pipe è presente il carattere che rappresenta l'operazione da svolgere; questo viene letto con `get_operation()`:

```
char get_operation(int fd)
{
    int rc;
    char v;
    rc = read(fd, &v, 1);
    if (rc == 0) {
        fprintf(stderr, "Truncated data in pipe\n");
        exit(EXIT_FAILURE);
    }
    exit_on_error(rc != 1, "read");
    return v;
}
```