Sistemi Operativi (M. Cesati)

Compito scritto del 8 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 un programma che continuamente legge dallo standard input righe di testo terminate da caratteri di fine riga (`\n`). Ciascuna riga rappresenta un comando esterno da eseguire, quindi contiene il nome del comando e gli eventuali argomenti:

comando *argomento1* *argomento2* ...

Il programma deve eseguire il comando (lanciato con gli eventuali argomenti indicati) in modo da salvare lo standard output del comando stesso su di un file chiamato "`out.`***n***", ove ***n*** è il numero progressivo del comando eseguito.
Ad esempio, se il programa inizialmente legge dallo standard input le seguenti tre righe:

```
ls -l -S /
date
cat /etc/passwd
```

allora il file "`out.1`" conterrà l'elenco dei file della directory radice ordinati per dimensione, il file "`out.2`" conterrà la data e l'ora al momento dell'esecuzione, ed il file "`out.3`" sarà una copia del file "`/etc/passwd`".

Esercizio 2. Si descrivano in modo esauriente i principali algoritmi di schedulazione dei trasferimenti da/verso i dischi magnetici, discutendone i rispettivi meriti e svantaggi.

Sistemi Operativi (M. Cesati)

Compito scritto del 8 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 continuamente legge dallo standard input linee con il seguente formato:

numero_decimale spazio stringa_di_testo

Il numero decimale è considerato un *ritardo in secondi*. Per ciascuna linea con il formato corretto l'applicazione crea un thread POSIX che attende il numero di secondi indicato e poi scrive in standard output la stringa di testo, *evitando di sovrapporre la scrittura a quella di altri thread.*
Ad esempio, passando contemporaneamente sullo standard input le tre linee seguenti:

```
10 ritardo massimo
1 piccolo ritardo
5 ritardo medio
```

verranno scritti sullo standard output da tre thread differenti:

- `> piccolo ritardo` (dopo circa 1 secondo)
- `> ritardo medio` (dopo circa 5 secondi)
- `> ritardo massimo` (dopo circa 10 secondi)

Si noti che l'applicazione deve continuare a leggere linee dallo standard input anche durante la stampa ritardata delle stringhe (sono attività da svolgere contemporaneamente). Si noti anche che le righe sullo standard input possono arrivare ad istanti di tempo non predicibili, pertanto non si possono in generale fare assunzioni sugli istanti di tempo in cui i vari thread dovranno scrivere sullo standard output.

Esercizio 2. Si descrivano in modo esauriente il ruolo e le principali varianti della memoria tampone dei sistemi operativi.

Sistemi Operativi (M. Cesati)

Esempio di programmi del compito scritto del 8 luglio 2013

Esercizio 1 — Turno 1

Svolgiamo l'esercizio seguendo un approccio "top-down". La funzione `main()` esegue un ciclo in cui vengono lette e processate tutte le linee dello standard input.

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

int main(int argc, char *argv[])
{
    int n;
    if (argc != 1) {
        fprintf(stderr,
                "Usage: %s (no arguments)\n", argv[0]);
        return EXIT_FAILURE;
    }
    n = 0;
    while (!feof(stdin))
        read_and_process_line(++n);
    return EXIT_SUCCESS;
}
```

La funzione `read_and_process_line()` riceve come argomento il numero progressivo della riga da leggere. Essa deve effettivamente leggere la linea, analizzare il suo contenuto, ed eseguire il comando corrispondente.

```
#define max_line_length 256
void read_and_process_line(int n)
{
    char line[max_line_length];
    if (read_line(line))
        process_line(line, n);
}
```

È stata prefissata una dimensione massima per la linea pari al valore della macro `max_line_length`. Il buffer `line` è destinato a contenere i dati letti dallo standard input. La funzione `read_line()` si occupa di leggere una linea dallo standard input; essa restituisce il valore 0 in caso di EOF:

```
int read_line(char line[])
{
    if (fgets(line, max_line_length, stdin) == NULL) {
        if (!feof(stdin)) {
            perror("stdin");
            exit(EXIT_FAILURE);
        }
        return 0;
    }
    /* remove the final '\n' character */
    if (line[strlen(line) - 1] == '\n')
        line[strlen(line) - 1] = '\0';
    return 1;
}
```

Si noti che la funzione di libreria `fgets()` può memorizzare anche il carattere di fine riga, che però va eliminato per evitare che sia considerato parte del nome del comando da eseguire o di un argomento.

La funzione `process_line()` riceve come argomenti il buffer contenente la linea letta dallo standard input ed il numero progressivo del comando:

```
void process_line(char line[], int n)
{
    char *argv[max_line_length / 2];
    parse_line(line, argv);
    execute_command(argv[0], argv, n);
}
```

La funzione deve svolgere due compiti: (1) analizzare la linea letta dallo standard input per costruire un vettore di puntatori a stringa, ciascuno corrispondente all'inizio del comando o di uno degli argomenti, e (2) eseguire il comando stesso. Si noti che il vettore dei puntatori a stringa è allocato staticamente sulla base della dimensione massima della linea (nel caso peggiore ogni comando o argomento è costituito da un singolo carattere, ed è seguito/preceduto da un carattere separatore).

La funzione `parse_line()` riceve come argomenti il buffer con la linea e l'indirizzo del vettore di puntatori a stringa:

```
void parse_line(char *line, char *argv[])
{
    int len = strlen(line);
    int i = 0, k = 0;
    while (i < len) {
        i = skip_blanks(line, i);
        if (i >= len)
            break;
        argv[k++] = line + i;
        i = skip_non_blanks(line, i + 1);
    }
    argv[k] = NULL;
}
```

Le funzioni `skip_blanks()` e `skip_non_blanks()` determinano la posizione del successivo carattere separatore e non separatore, rispettivamente:

```
int skip_blanks(char *line, int i)
{
    while (i < max_line_length && (line[i] == ' ' ||
            line[i] == '\t')) {
        line[i] = '\0';
        ++i;
    }
    return i;
}
int skip_non_blanks(char *line, int i)
{
    while (i < max_line_length && line[i] != ' ' &&
        line[i] != '\t')
        ++i;
    return i;
}
```

Si noti come la funzione `skip_blanks()` sostituisca tutti i caratteri separatore con terminatori di stringa, così che alla fine ciascun elemento nel vettore `argv` punta ad una stringa "indipendente".

Per eseguire il comando esterno si invoca la funzione `execute_command()`, che riceve una stringa con il nome del comando, il vettore di puntatori di stringa corrispondente agli argomenti, ed il numero progressivo del comando:

```
void execute_command(char *cmd, char *argv[], int n)
{
    pid_t pid;

    if (!cmd) /* nothing to do (blank line) */
        return;
```

```
    if ((pid = fork()) == -1) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    if (pid) /* the parent returns */
        return;
    redirect_output(n);
    execvp(cmd, argv);
    perror(cmd);
    exit(EXIT_FAILURE);
}
```

La funzione crea un processo: il padre ritorna subito, mentre il figlio prima redirige lo standard output e poi esegue la funzione di libreria `execvp()` per eseguire il comando.

La funzione `redirect_output()` apre un file in scrittura con un nome dipendente dal numero progressivo del comando e ridirige lo standard output su tale file:

```
void redirect_output(int n)
{
    char filename[32];
    int fd;
    sprintf(filename, "out.%d", n);
    fd = open(filename, O_WRONLY | O_CREAT, 0644);
    if (fd == 1) {
        perror(filename);
        exit(EXIT_FAILURE);
    }
    if (dup2(fd, STDOUT_FILENO) == -1) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
    if (close(fd) == -1)
        perror(filename);
}
```

Esercizio 1 — Turno 2

Svolgiamo l'esercizio seguendo un approccio "top-down", dunque iniziamo a descrivere la funzione `main()`:

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

pthread_mutex_t mutex;

int main(int argc, char *argv[])
{
    if (argc != 1) {
        fprintf(stderr,
                "Usage: %s (no arguments)\n", argv[0]);
        return EXIT_FAILURE;
    }

    if (pthread_mutex_init(&mutex, NULL)) {
        fprintf(stderr, "Error in pthread_mutex_init()\n");
        return EXIT_FAILURE;
    }

    while (!feof(stdin))
        process_input_line();

    pthread_exit(0);
}
```

La funzione `main()` inizializza un mutex globale utilizzato dai thread POSIX per sincronizzare le scritture in standard output, come richiesto dall'esercizio.

Il cuore del programma è un ciclo che legge dallo standard input. La funzione `process_input_line()` alloca spazio sullo stack per una linea di testo (vettore `buf`). Assumiamo che la linea sia lunga al massimo 128 caratteri:

```
void process_input_line(void)
{
    char buf[128];
    char *msg;
    unsigned long delay;

    msg = parse_input_line(buf, 128, &delay);
    if (msg != NULL)
        create_thread(delay, msg);
```

```
}
```

La funzione `parse_input_line()` legge la linea dallo standard input e la analizza. Il ritardo in secondi viene memorizzato in `delay`, mentre `msg` punta entro `buf` al primo carattere della stringa di testo. In caso di linea malformata `parse_input_line()` restituisce `NULL`.

```
char * parse_input_line(char *line, int maxsize, unsigned
    long *v)
{
    char *msg;

    /* read a line from standard input */
    if (fgets(line, maxsize, stdin) == NULL) {
        if (!feof(stdin))
            fprintf(stderr,
                    "Error reading from standard input\n");
        return NULL;
    }

    /* parse the line */
    errno = 0;
    *v = strtoul(line, &msg, 10);
    if (errno != 0) {
        fprintf(stderr,
                "Invalid number in standard input line\n");
        return NULL;
    }
    if (*msg++ != ' ') {
        fprintf(stderr,
             "Invalid separator in standard input line\n");
        return NULL;
    }

    /* remove the trailing new line, if any */
    if (msg[strlen(msg)-1] == '\n')
        msg[strlen(msg)-1] = '\0';
    return msg;
}
```

Per ciascuna riga ben formata letta dallo standard input viene eseguita la funzione `create_thread()`:

```
struct thread_data {
    unsigned long delay;
    char *message;
    pthread_t tid;
};
```

```
void create_thread(unsigned long delay, char *message)
{
    struct thread_data *ptd;
    ptd = malloc(sizeof(struct thread_data));
    if (ptd == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    ptd->delay = delay;
    ptd->message = strdup(message);
    if (ptd->message == NULL) {
        fprintf(stderr, "Memory allocation error\n");
        exit(EXIT_FAILURE);
    }
    if (pthread_create(&ptd->tid, NULL, t_delay, ptd)) {
        perror(NULL);
        exit(EXIT_FAILURE);
    }
}
```

La funzione alloca una nuova istanza di una struttura `thread_data` contenente i dati da passare alla funzione eseguita dal nuovo thread. Per copiare la stringa di testo del messaggio si è fatto uso della funzione di libreria `strdup()`, che sostanzialmente esegue `malloc()` per allocare memoria sufficiente per la stringa e poi esegue `strcpy()` per copiare i caratteri della stringa. Infine la funzione `create_thread()` invoca `pthread_create()` per create un nuovo thread, passandogli l'indirizzo della funzione `t_delay()`.

```
void *t_delay(void *p)
{
    struct thread_data *ptd = (struct thread_data *) p;
    tsleep(ptd->delay);
    if (pthread_mutex_lock(&mutex)) {
        fprintf(stderr,
                "Error in by pthread_mutex_lock()\n");
        exit(EXIT_FAILURE);
    }
    printf("> %s\n", ptd->message);
    if (pthread_mutex_unlock(&mutex)) {
        fprintf(stderr,
                "Error in by pthread_mutex_unlock()\n");
        exit(EXIT_FAILURE);
    }
    free(ptd->message);
    free(ptd);
    pthread_exit(0);
}
```

La funzione `t_delay()` sospende il thread per il numero prefissato di secondi. Poi acquisisce il mutex, scrive il messaggio sullo standard output, e rilascia il mutex. Infine la funzione rilascia la memoria occupata dalla stringa del messaggio e dalla struttura di dati `thread_data`, e termina l'esecuzione del thread.

La funzione `tsleep()` ha il compito di sospendere il thread per un numero prefissato di secondi passato come argomento, ed è realizzata utilizzando la funzione di libreria `nanosleep()`:

```
void tsleep(unsigned long delay)
{
    struct timespec ts;
    ts.tv_sec = delay;
    ts.tv_nsec = 0;
    nanosleep(&ts, NULL); /* assume no signal received */
}
```

Si osservi che lo standard POSIX specifica che la funzione `nanosleep()` non deve fare uso di segnali, pertanto è possibile utilizzarla senza problemi in una applicazione multithread.

Al contrario la funzione `sleep()` *potrebbe* far uso del segnale `SIGALRM`, pertanto in generale non può essere utilizzata in modo affidabile in una applicazione multithread che sia portabile su sistemi diversi. Per inciso, in Linux la funzione `sleep()` è basata su `nanosleep()` e non utilizza segnali.