Sistemi Operativi (M. Cesati)

Compito scritto del 7 settembre 2016 (Traccia A)

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 multiprocesso che legge da un file di input il cui nome è specificato come primo argomento della linea comando. Il file è costituito da righe di testo con il seguente formato:

numero_decimale spazio stringa di testo

Il numero in formato testo decimale è da considerare come un *ritardo periodico in secondi*. Per ciascuna riga del file con il formato corretto l'applicazione crea un processo che attende il numero di secondi indicato e poi esegue il comando (con eventuali argomenti) specificato dalla stringa di testo. Dopo la conclusione del comando, il processo attende ancora il numero di secondi indicato e poi torna ad eseguire il comando, e così via.
Ad esempio, se il file contiene le seguenti due righe:

```
100 ls -l /tmp
60 date
```

dovranno essere creati due nuovi processi: un processo eseguirà `ls`, che elencherà in standard output il contenuto della directory `/tmp`, ad intervalli di 100 secondi; l'altro processo eseguirà `date`, che scriverà in standard output data e ora corrente, ad intervalli di 60 secondi.

Esercizio 2

Si descriva in modo esauriente la differenza tra un sistema operativo monolitico ed uno basato su microkernel.

Sistemi Operativi (M. Cesati)

Compito scritto del 7 settembre 2016 (Traccia B)

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 multiprocesso che legge da un file di input il cui nome è specificato come primo argomento della linea comando. Il file è costituito da righe di testo con il seguente formato:

numero_decimale spazio stringa di testo

Il numero in formato testo decimale è da considerare come un *timeout in secondi*. Per ciascuna riga del file con il formato corretto l'applicazione crea un processo che esegue il comando (con eventuali argomenti) specificato dalla stringa di testo. Se però l'esecuzione del comando non si conclude entro il numero di secondi indicato, al processo che esegue il comando dovrà essere inviato il segnale di terminazione `SIGTERM`. Qualora dopo 10 secondi dall'invio di `SIGTERM` il processo non sia ancora terminato, dovrà essergli inviato il segnale di terminazione `SIGKILL`. Ad esempio, se il file contiene le righe:

```
3600 nano /tmp/mail
60 ls -lR /
```

dovranno essere eseguiti due comandi, il primo con un timeout di 1 ora, il secondo con un timeout di 1 minuto.
Si badi che l'esecuzione dei vari comandi indicati nel file deve essere concorrente, ossia l'applicazione deve lanciare tutti i comandi il più rapidamente possibile.

Esercizio 2

In riferimento ad un generico sistema operativo moderno, si descrivano in modo esauriente i possibili stati di un processo e gli eventi che causano le transizioni tra tali stati.

Sistemi Operativi (M. Cesati)

Esempi dei programmi del compito scritto del 7 settembre 2016

Esercizio 1, traccia A

Descriviamo il programma utilizzando un approccio "top-down". La funzione `main()` è molto semplice: controlla che sia stato passato il nome di un file sulla linea comando ed invoca la funzione che processa tale file.

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

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

int main(int argc, char *argv[])
{
    abort_on_error(argc < 2,
        "Missing file name in command line");
    read_input_file(argv[1]);
    return EXIT_SUCCESS;
}
```

La funzione `read_input_file()` apre il file di ingresso ed avvia l'elaborazione del suo contenuto:

```
void read_input_file(const char *filename)
{
    int rc;
    FILE *f;
    abort_on_error(!filename || *filename=='\0',
            "Invalid file name in command line");
    f = fopen(filename, "r");
    abort_on_error(f == NULL, "Cannot open input file");
    read_all_lines(f);
```

```
    rc = fclose(f);
    abort_on_error(rc!=0,"Error while closing input file");
}
```

Il file viene effettivamente letto dalla funzione `read_all_lines()`, una riga per volta:

```
#define MAX_LINE_LENGTH 4096

void read_all_lines(FILE *f)
{
    char lbuf[MAX_LINE_LENGTH], *line;
    while (!feof(f)) {
        line = get_next_line(f, lbuf);
        if (line != NULL)
            process_line(line);
    }
}
```

La lettura di una singola riga del file è realizzata da `get_next_line()`:

```
char *get_next_line(FILE *f, char *buf)
{
    char *p;
    p = fgets(buf, MAX_LINE_LENGTH, f);
    abort_on_error(p == NULL && ferror(f),
            "Error reading from input file");
    return p; /* p == NULL in case of EOF */
}
```

Si noti che dopo aver letto l'ultima riga del file, `fgets()` restituisce il valore `NULL`, e `read_all_lines()` appropriatamente controlla tale valore prima di provare a decodificare la riga.

La funzione `process_line()` analizza ogni singola riga del file:

```
void process_line(char *line)
{
    unsigned long delay;
    char *cmdstr;
    line[strlen(line)-1]='\0';
    delay = read_value(line, &cmdstr);
    if (cmdstr)
        spawn_periodic_command(delay, cmdstr);
}
```

Innanzi tutto la funzione rimuove il carattere `'\n'` alla fine della riga di testo. Poi il valore in formato testo decimale che si dovrebbe trovare all'inizio della riga viene convertito in formato nativo e memorizzato in `delay` dalla funzione `read_value()`:

```
unsigned long read_value(char *line, char **nextch)
{
    unsigned long delay;
    char *p;

    errno = 0;
    delay = strtoul(line, &p, 10);
    if (errno != 0 || *p != ' ') {
        fprintf(stderr, "Invalid line: %s\n", line);
        *nextch = NULL;
        return 0;
    }
    *nextch = ++p;
    return delay;
}
```

Se le cifre decimali all'inizio della riga non sono seguite rigorosamente da uno spazio bianco, l'intera riga viene considerata avente formato non valido, e ciò viene segnalato a `process_line()` impostando la variabile puntatore `cmdstr` a `NULL`. Altrimenti, in `cmdstr` viene scritto l'indirizzo del primo carattere dopo lo spazio bianco.

La durata del ritardo e la stringa di testo contenente il comando da eseguire vengono passati alla funzione `spawn_periodic_command()`:

```
void spawn_periodic_command(unsigned long delay,
                            char *cmdstr)
{
    int rc, status;
    pid_t pid = fork();
    abort_on_error(pid == -1, "Error in fork()");
    if (pid != 0)
        return; /* parent, go out */
    sleep(delay);
    pid = spawn_command(cmdstr);
    for (;;) {
        rc = wait(&status);
        abort_on_error(rc == -1, "Error in wait()");
        if (pid == rc && !WIFSTOPPED(status)
                && !WIFCONTINUED(status)) {
            sleep(delay);
            pid = spawn_command(cmdstr);
        }
    }
```

```
}
```

La funzione crea un processo figlio. Il genitore, ossia il processo principale dell'applicazione, poi ritorna immediatamente. Il processo figlio al contrario invoca `sleep()` per attendere il numero di secondi specificato, poi invoca `spawn_command()` per lanciare il comando. Si entra poi in un ciclo senza fine, nel quale prima si attende che il comando lanciato termini l'esecuzione (con la chiamata di sistema `wait()`), poi si torna ad attendere il numero di secondi specificato e si rilancia il comando. I controlli effettuati dopo la chiamata di sistema `wait()` sono volti ad assicurare che l'evento segnalato da `wait()` sia effettivamente la terminazione del processo che esegue il comando: il valore restituito da `wait()` deve coincidere con il PID del processo, e `status` non deve indicare che l'evento in questione è legato al blocco (STOP) o continuazione del processo.

Per lanciare il comando viene utilizzata la funzione `spawn_command()`:

```
pid_t spawn_command(char *cmdstr)
{
    char **argv;
    pid_t pid = fork();
    abort_on_error(pid == -1, "Error in fork()");
    if (pid != 0)
        return pid; /* parent, go out */
    argv = convert_string_to_argv(cmdstr);
    execvp(argv[0], argv);
    abort_on_error(1, "Cannot execute command");
    return 0;
}
```

Essenzialmente la funzione esegue `fork()`, poi il processo figlio esegue da `execvp()`. L'unica complicazione risiede nel fatto che tutte le chiamate di sistema della famiglia `exec` richiedono un vettore di puntatori a stringa, ciascuno elemento del quale identifica il nome del programma da eseguire oppure un singolo argomento. Nel nostro caso, invece, queste informazioni sono codificate nell'unica stringa `cmdstr`, ed i vari elementi sono separati da spazi bianchi.

Per effettuare la conversione tra la stringa unica ed il formato "`argv`" si utilizza la funzione `convert_string_to_argv()`:

```
char **convert_string_to_argv(char *cmdstr)
{
    char **argv, *p;
    int i, argc;
    argc = compute_argc(cmdstr);
    argv = malloc(sizeof(argv[0])*(argc+1));
    abort_on_error(argv==NULL, "Memory allocation error");
```

```
    for (i=0, p=cmdstr; i<argc; ++i) {
        argv[i] = p;
        p = strchr(p, ' ');
        for (; p != NULL && *p == ' '; *p++ = '\0')
            ;
    }
    argv[argc] = NULL;
    return argv;
}
```

Innanzi tutto la funzione conta il numero di parole separate da spazi bianchi contenute all'interno della stringa `cmdstr` utilizzando la funzione `compute_argc()`. Poi alloca spazio per il vettore `argv()` (allocando un elemento in più, che conterrà un valore `NULL` finale). Infine gli indirizzi di tutte le parole della stringa `cmdstr()` vengono scritti entro `argv`. Per trovare la posizione del successivo spazio bianco nella stringa viene utilizzata la funzione di libreria `strchr()`. Si considera inoltre il caso in cui a separare due parole sia una sequenza di spazi bianchi al posto di un singolo spazio. Infine, la funzione sostituisce tutti gli spazi bianchi con il terminatore di stringa `'\0'`, in modo che tutte le stringhe referenziate da `argv` siano correttamente terminate.

La funzione `compute_argc()` ha in effetti una logica molto simile a quella appena vista, ma si limita a contare le parole nella stringa:

```
int compute_argc(char *p)
{
    int argc = 1;
    if (p == NULL)
        return 0;
    for (;;) {
        p = strchr(p, ' ');
        if (p == NULL)
            return argc;
        while (*p == ' ')
            ++p;
        ++argc;
    }
}
```

Esercizio 1, traccia B

Il programma è molto simile a quello della traccia A. Cominciamo dalla funzione `main()`, che è assolutamente identica:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <string.h>
#include <signal.h>
#include <errno.h>
#include <sys/types.h>
#include <sys/wait.h>

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

int main(int argc, char *argv[])
{
    abort_on_error(argc < 2,
            "Missing file name in command line");
    read_input_file(argv[1]);
    return EXIT_SUCCESS;
}
```

Anche la funzione `read_input_file()` è identica a quella mostrata nel programma della traccia A, così come le funzioni `read_all_lines()` e `get_next_line()`, poiché il formato del file di ingresso è il medesimo.

La funzione che analizza ciascuna riga del file è ancora chiamata `process_line()`:

```
void process_line(char *line)
{
    unsigned long timeout;
    char *cmdstr;
    line[strlen(line)-1]='\0';
    timeout = read_value(line, &cmdstr);
    if (cmdstr)
        spawn_command_with_timeout(timeout, cmdstr);
}
```

Anche questa volta viene rimosso il carattere terminatore di linea e viene letto il valore decimale all'inizio della riga (utilizzando la funzione `read_value()` identica a quella già mostrata per la traccia A).

Questa volta per invocare il comando viene utilizzata la funzione `spawn_command_with_timeout()`:

```
void spawn_command_with_timeout(unsigned long timeout,
                                char *cmdstr)
{
    pid_t pid = fork();
    abort_on_error(pid == -1, "Error in fork()");
    if (pid != 0)
        return; /* parent, go out */
    pid = spawn_command(cmdstr);
    if (pid)
        pid=send_signal_on_timeout(pid, timeout, SIGTERM);
    if (pid)
        send_signal_on_timeout(pid, 10, SIGKILL);
}
```

La funzione è analoga a quanto già visto, ed utilizza la stessa funzione `spawn_command()` per lanciare un nuovo processo che esegue il comando. La differenza è che dopo aver lanciato il comando si controlla che esso termini entro i tempi indicati. In particolare, dopo un intervallo di tempo lungo `timeout` secondi viene inviato un segnale `SIGTERM`. In caso di necessità, dopo ulteriori 10 secondi viene inviato il segnale `SIGKILL`.

Per l'attesa e conseguente invio del segnale viene utilizzata la funzione `send_signal_on_timeout()`, che riceve il PID del processo che esegue il comando, la durata dell'attesa, ed il codice del segnale da inviare.

A titolo esemplificativo, consideriamo due possibili implementazioni alternative per questa funzione. La prima variante utilizza la funzione `sleep()` per attendere in modo incondizionato, poi invoca `waitpid()` in modalità non bloccante per controllare se il processo in questione è terminato o meno:

```
pid_t send_signal_on_timeout(pid_t pid,
                    unsigned long timeout, int signal)
{
    int rc, status;
    sleep(timeout);
    rc = waitpid(pid, &status, WNOHANG);
    abort_on_error(rc == -1, "Error in wait()");
    if (pid == rc && !WIFSTOPPED(status)
            && !WIFCONTINUED(status))
        return 0;
    send_signal(pid, signal);
    return pid;
```

```
}
```

Al solito si controlla che l'evento segnalato da `waitpid()` sia effettivamente la terminazione del processo. L'opzione `WNOHANG` è essenziale, perché fa in modo che `waitpid()` non blocchi l'esecuzione ma ritorni subito nel caso in cui il comando non sia ancora terminato.

Per inviare il segnale si utlizza la funzione `send_signal()`, basata sulla chiamata di sistema `kill()`:

```
void send_signal(pid_t pid, int signal)
{
    int rc = kill(pid, signal);
    abort_on_error(rc == -1, "Error in kill()");
}
```

La seconda variante della funzione `send_signal_on_timeout()` utilizza la chiamata di sistema bloccante `wait()`, ma fa in modo che essa sia interrotta dal segnale `SIGALRM` alla fine dell'intervallo di tempo d'attesa:

```
pid_t send_signal_on_timeout(pid_t pid,
                    unsigned long timeout, int signal)
{
    int rc, status;
    register_sigalrm();
    alarm(timeout);
    rc = wait(&status);
    abort_on_error(rc == -1 && errno != EINTR,
                    "Error in wait()");
    if (pid == rc && !WIFSTOPPED(status)
            && !WIFCONTINUED(status))
        return 0;
    send_signal(pid, signal);
    return pid;
}
```

Si noti che ora si deve controllare esplicitamente il motivo per il quale `wait()` termina con un codice d'errore: se `errno` ha il valore `EINTR`, la chiamata di sistema è stata effettivamente interrotta dal segnale ricevuto, quindi si deve procedere ad inviare il segnale di terminazione all'altro processo.

È inoltre necessario registrare un gestore del segnale `SIGALRM` (in effetti questo potrebbe essere fatto anche prima, ad esempio in `main()`):

```c
void alarm_hndl(int sig)
{
    sig = sig;
}

void register_sigalrm(void)
{
    int rc;
    struct sigaction sa;
    sa.sa_handler = alarm_hndl;
    sigemptyset(&sa.sa_mask);
    sa.sa_flags = 0; /* no SA_RESTART ! */
    rc = sigaction(SIGALRM, &sa, NULL);
    abort_on_error(rc == -1, "Error in sigaction()");
}
```

Il gestore del segnale in effetti non compie alcuna azione utile, ma deve essere registrato altrimenti la ricezione del segnale provocherebbe la terminazione immediata del processo, che quindi non potrebbe inviare il segnale di terminazione all'altro processo.

Si noti che non è possibile, in questo caso, registrare il gestore del segnale con la chiamata di sistema `signal()` invece che `sigaction()`. Infatti, quando si registra un gestore di segnali con `signal()` si utilizza implicitamente (almeno in Linux) un meccanismo che riavvia automaticamente alcune chiamate di sistema interrotte dal segnale. In pratica, la soluzione che vogliamo utilizzare non funzionerebbe perché `wait()` non sarebbe realmente interrotta dal segnale `SIGALRM`. Utilizzando invece `sigaction()` possiamo escludere esplicitamente il meccanismo di riavvio automatico delle chiamate di sistema, perché tra le opzioni non indichiamo il flag `SA_RESTART`.