Sistemi Operativi (M. Cesati)

Compito scritto del 25 settembre 2014

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 diversi processi interagenti. L'applicazione legge dalla linea comando una serie di nomi di file ed avvia un processo per ciascun file specificato. Ciascun processo dovrà leggere il proprio file una riga di testo alla volta, e scrivere tale riga sullo standard output dell'applicazione, rispettando il seguente ordinamento:

- Prima riga del file del primo processo
- Prima riga del file del secondo processo

⋮

- Prima riga del file dell'ultimo processo
- Seconda riga del file del primo processo
- Seconda riga del file del secondo processo

⋮

L'applicazione deve gestire correttamente il caso in cui i file non sono costituiti dallo stesso numero di righe di testo (nel turno corrispondente il processo relativo non scriverà nulla in standard output). Prestare attenzione agli aspetti della sincronizzazione tra i processi e della efficienza dell'applicazione, evitando di introdurre ritardi artificiali per forzare l'ordinamento delle righe scritte sullo standard output.

Esercizio 2. Si descrivano lo scopo ed il funzionamento della "tabella dei file aperti" utilizzata nei kernel dei Sistemi Operativi della famiglia Unix.

Sistemi Operativi (M. Cesati)

Esempio di programma del compito scritto del 25 settembre 2014

Esercizio 1

Per sincronizzare i processi tra loro verranno uilizzati due segnali, `SIGUSR1` e `SIGUSR2`. In effetti `SIGUSR1` è sufficiente per gestire la scrittura coordinata sullo standard output. Il segnale `SIGUSR2` è invece utilizzato per implementare la chiusura dell'applicazione quando tutti i processi hanno terminato di copiare il proprio file. La scelta di utilizzare due segnali, al posto di altri meccanismi di sincronizzazione più flessibili come ad esempio memoria condivisa o code di messaggi, ha lo svantaggio di richiedere una accurata progettazione del protocollo di comunicazione tra i processi per coprire tutti i possibili casi ed evitare le "race condition". Svolgiamo l'esercizio seguendo un approccio "top-down":

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

int main(int argc, char *argv[])
{
    if (argc < 2) {
        fprintf(stderr, "Usage: %s <file> <file> ...\n", argv[0]);
        return EXIT_FAILURE;
    }
    setlinebuf(stdout);
    register_sighandlers();
    fork_processes(argc, argv);
    return EXIT_SUCCESS;
}
```

Poiché i vari processi dovranno scrivere linee di testo sullo standard output dell'applicazione in modo sincronizzato, è necessario evitare che le scritture possano sovrapporsi tra loro a causa dei buffer della libreria C di ciascun processo. A tale scopo, impostiamo la modalità "line buffered" per lo standard output con la funzione `setlinebuf()`, il che equivale in pratica ad invocare `fflush(stdout)` dopo ogni scrittura di riga.

La funzione `register_sighandlers()` si occupa di registrare i gestori dei segnali `SIGUSR1` e `SIGUSR2`:

```
void register_sighandlers(void)
{
    if (signal(SIGUSR1, usr_handler) == SIG_ERR ||
        signal(SIGUSR2, usr_handler) == SIG_ERR) {
        fprintf(stderr, "Error in signal()\n");
        exit(EXIT_FAILURE);
    }
}
```

Il gestore dei due segnali è unico: si limita ad incrementare una variabile globale diversa a seconda del segnale ricevuto:

```
int turno = 0;
int prec_ended = 0;

void usr_handler(int sig)
{
    if (sig == SIGUSR1)
        turno++;
    else
        prec_ended++;
}
```

La funzione `main()` termina invocando la funzione `fork_processes()`:

```
void fork_processes(int argc, char *argv[])
{
    int i;
    int next_pid = getpid();

    for (i=argc-1; i>1; --i) {
        pid_t pid = fork();
        if (pid == -1)
            exit(EXIT_FAILURE);
        if (pid == 0)
            do_job(argv[i], next_pid); /* never returns */
        next_pid = pid;
    }
    turno = 1;
    prec_ended = 1;
    do_job(argv[i], next_pid); /* never returns */
}
```

I processi vengono creati in ordine inverso rispetto ai nomi di file posti sulla linea comando. Come primo processo viene creato quello che gestisce l'ultimo nome di file, e riceve l'identificatore del processo principale dell'applicazione (che ha eseguito `main()`). Come secondo

processo viene creato quello che gestisce il penultimo nome di file, e riceve l'identificatore del processo creato in precedenza. L'ultimo processo creato gestirà il secondo nome di file sulla linea comando. Infine il processo originale gestirà il primo nome di file sulla linea comando.

Questo algoritmo garantisce che ciascun processo possiede l'identificatore del successivo processo nell'ordine definito dai nomi di file sulla linea comando. L'ultimo di tali processi tuttavia avrà come identificatore quello del processo principale associato al primo nome di file. In breve, si è definito un ciclo tra i processi.

In ciascun momento solo un processo potrà avere la propria variabile `turno` diversa da zero: questa variabile stabilisce chi ha il diritto a scrivere sullo standard output. Il processo originale associato al primo nome di file dovrà cominciare a scrivere per primo, quindi la sua variabile `turno` è impostata a 1 prima di iniziare.

La variabile `end_prec` vale 1 se tutti i processi precedenti nell'ordine definito sopra hanno terminato di leggere il proprio file. Il processo originale associato al primo nome di file non ha alcun processo che lo precede, quindi per esso `end_proc` è impostato subito ad uno.

Il lavoro svolto da ciascun processo è implementato dalla funzione `do_job()`:

```
void do_job(char *file, pid_t next)
{
    process_file(file, next);
    wait_for_others(next);
}
```

Il lavoro di ciascun processo è suddiviso in due fasi distinte: la prima si occupa di leggere e scrivere le linee del proprio file rispettando il protocollo per il turno con gli altri processi; la seconda viene avviata quando il proprio file è stato interamente copiato, e si occupa di continuare a far circolare i segnali `SIGUSR1` per il turno e di gestire la chiusura dell'applicazione.

La prima fase è implementata dalla funzione `process_file()`:

```
void process_file(char *file, pid_t next)
{
#   define max_line_len 1024
    char linebuf[max_line_len];
    FILE *f = fopen(name, "r");
    if (f == NULL) {
        fprintf(stderr, "Error: cannot open file \"%s\"\n", file);
        exit(EXIT_FAILURE);
    }

    while (!feof(f)) {
        if (fgets(linebuf, max_line_len, f) != NULL) {
            while (turno == 0)
                pause();
            fputs(linebuf, stdout);
            turno = 0;
            send_signal(SIGUSR1, next);
```

```
        }
        if (ferror(f)) {
            fprintf(stderr, "Error reading from file %s\n", file);
            exit(EXIT_FAILURE);
        }
    }
    if (fclose(f)) {
        fprintf(stderr, "Error closing file %s\n", file);
        exit(EXIT_FAILURE);
    }
}
```

Finché il file non èterminato, il processo legge una riga con `fgets()`, poi aspetta il proprio turno, ossia che la propria variabile `turno` assuma il valore 1. Per evitare di occupare inutilmente la CPU si invoca la chiamata di sistema `pause()`, che sospende il processo finché non viene ricevuto un segnale non ignorato. Quando `turno` vale 1, la riga viene scritta sullo standard output, poi la variabile `turno` viene azzerata e si invia il segnale `SIGUSR1` al processo successivo. (Si noti che invertire l'ordine delle due operazioni dà luogo ad una race condition.)

La funzione `send_signal()` è utilizzata per inviare il segnale controllando il successo dell'operazione:

```
void send_signal(int sig, pid_t pid)
{
    if (kill(pid, sig) == -1) {
        fprintf(stderr, "Error sending signal SIGUSR1\n");
        exit(EXIT_FAILURE);
    }
}
```

La seconda fase di lavoro del processo è implementata dalla funzione `wait_for_others()`:

```
void wait_for_others(pid_t next)
{
    int sig_end_sent = 0;
    for (;;) {
        while (turno == 0)
            pause();
        if (prec_ended == 1 && !sig_end_sent) {
            send_signal(SIGUSR2, next);
            sig_end_sent = 1;
        }
        if (prec_ended == 2) {
            kill(next, SIGUSR2);
            kill(next, SIGUSR1);
            exit(EXIT_SUCCESS);
        }
        turno = 0;
```

```
            send_signal(SIGUSR1, next);
        }
    }
```

La funzione esegue un ciclo in cui si continua ad attendere la ricezione del segnale `SIGUSR1` e a rispedirlo al processo successivo, in modo da assicurare la continuazione del protocollo per i turni di scrittura. Ad ogni ricezione del segnale, comunque, si controlla anche se l'intera applicazione deve essere chiusa in quanto tutti i file sono stati interamente copiati.

A tale scopo, la variabile `prec_ended` vale 0 se uno o più dei processi precedenti non ha ancora terminato la fase 1, vale 1 se tutti i processi preedenti hanno terminato la fase 1, e vale 2 se tutti i processi del ciclo hanno terminato la fase 1, e quindi è possibile chiudere l'applicazione.

In ciascuna iterazione del ciclo, al proprio turno, il processo controlla se la variabile `prec_ended` vale 1: in questo caso, poiché il processo stesso si trova nella fase 2, invia il segnale `SIGUSR2` al processo successivo nel ciclo. Tale segnale avrà l'effetto di incrementare la variabile `prec_ended` del processo ricevente. La spedizione di `SIGUSR2` deve però essere fatta una sola volta, e di questo si occupa il flag `sig_end_sent`.

Quando l'ultimo processo termina di copiare il proprio file e vede la variabile `prec_ended` uguale ad uno, esso invia il segnale `SIGUSR2` al primo processo, quello che aveva `prec_ended` già uguale ad 1 fin dall'inizio. Quando un processo osserva il valore 2 in `prec_ended` sa che tutti i processi successivi hanno terminato di copiare il file, e quindi l'applicazione può essere terminata. Pertanto, invia i segnali `SIGUSR2` (per incrementare a 2 la variabile `prec_ended` del processo successivo) e `SIGUSR1` (per passare il turno), e termina l'esecuzione.

Si noti che l'ultimo processo tenta di inviare i segnali finali al primo processo, che presumibilmente è già terminato. Per semplicità evitiamo di controllare il codice d'errore restituito dalle funzioni `kill`, in quanto già sappiamo che in questo caso particolare esse falliscono. Infine, si noti che invertendo l'ordine dell'invio dei due segnali finali (prima `SIGUSR1` e poi `SIGUSR2`) si viene a creare una "race condition" che può impedire la terminazione di qualche processo.