Sistemi Operativi **(M. Cesati)**

Compito scritto del 17 febbraio 2015

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 multithread che simula il comportamento del sistema dei Cinque Filosofi a Cena (E. W. Dijkstra, 1965). I cinque filosofi sono seduti a cena ad un tavolo rotondo. Davanti a ciascun filosofo c'è un piatto di spaghetti, e tra un piatto e l'altro c'è una bacchetta (quindi in totale esistono cinque bacchette). Ciascun filosofo pensa per un tempo casuale, poi cerca di prendere le bacchette a destra e sinistra del proprio piatto, mangia per un tempo casuale, lascia le due bacchette, e torna a pensare, continuando così all'infinito. Ovviamente se l'una o l'altra delle due bacchette non è disponibile il filosofo dovrà aspettare prima di iniziare a mangiare.

L'applicazione deve fare uso di cinque thread, ciascuno dei quali realizza il comportamento di un filosofo. In ciascuna iterazione ogni thread scrive in standard output l'istante di tempo in cui inizia a pensare, smette di pensare, ed inizia a mangiare. Le "bacchette" condivise tra i thread devono essere realizzate con opportuni meccanismi di sincronizzazione, e si devono evitare le situazioni di stallo dovute a "deadlock".

Nota 1: per misurare il tempo si può utilizzare la API:

```
#include <time.h>
time_t time(time_t *t);
```

In pratica `time(NULL)` restituisce il numero di secondi trascorsi da una data prefissata (1 gennaio 1970).

Nota 2: per ottenere un valore casuale si può utilizzare la API:

```
#include <stdlib.h>
long random(void);
```

In pratica `random()` restituisce un numero pseudo-casuale compreso tra 0 ed il valore `RAND_MAX`.

Esercizio 2. Si descrivano gli scopi e le possibili implementazioni della memoria tampone per i dischi utilizzata nei moderni Sistemi Operativi.

Sistemi Operativi (M. Cesati)

Esempio del programma del compito scritto del 17 febbraio 2015

L'esercizio richiede di simulare il comportamento dei cinque filosofi a cena utilizzando cinque thread POSIX. È facile rendersi conto che per realizzare il programma si deve prevedere un meccanismo di sincronizzazione che consenta di tenere traccia dello stato libero/occupato di ciascuna delle cinque bacchette a disposizione dei filosofi. Facciamo la scelta di utilizzare come primitiva di sincronizzazione un vettore di cinque mutex, un mutex per ciascuna bacchetta.

L'altro problema da risolvere è quale strategia adottare per evitare le situazioni di stallo (deadlock). Ciascun filosofo deve utilizzare le bacchette poste alla propria destra ed alla propria sinistra. Se assegniamo ai filosofi indici da zero a quattro, ed alle cinque bacchette indici da zero a quattro, potremo avere:

Filosofo	Bacchetta a sinistra	Bacchetta a destra
0	4	0
1	0	1
2	1	2
3	2	3
4	3	4

Se ciascun filosofo prende le bacchette in un ordine relativo prefissato, ad esempio prende sempre prima la bacchetta a sinistra e poi quella a destra, allora è possibile che si verifichino situazione di stallo. D'altra parte, se ciascun filosofo prende sempre prima la bacchetta con indice minore (ossia se rispetta l'ordine globale indotto dagli indici delle bacchette), nessuno stallo si verificherà mai. Pertanto l'ordine con cui i filosofi prenderanno le bacchette dovrà essere il seguente:

Filosofo	Prima bacchetta	Seconda bacchetta
0	0	4
1	0	1
2	1	2
3	2	3
4	3	4

Memorizziamo l'ordine in cui i processi dovranno acquisire i mutex in due vettori costanti:

```
#define num 5
const int first_mux[num]  = { 0, 0, 1, 2, 3 };
const int second_mux[num] = { 4, 1, 2, 3, 4 };
```

Sviluppiamo il programma con un approccio "top-down". La funzione principale del programma dovrà inizializzare le strutture di dati e creare cinque thread POSIX per la simulazione di ciascun filosofo:

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

int main()
{
  initialize();
  create_phils();
  pthread_exit(0);
}
```

Il thread principale termina subito dopo aver creato i cinque thread figli. Si noti che non è possibile terminare `main()` con `return` o `exit`, altrimenti tutti i thread figli verrebbero immediatamente uccisi.

Il compito più importante della funzione `initialize()` è inizializzare il vettore dei cinque mutex (allocato staticamente come variabile globale):

```
time_t base_time;
pthread_mutex_t chopsticks[num];

void initialize(void)
{
  int i;

  for (i = 0; i < num; ++i)
    if (pthread_mutex_init(chopsticks + i, NULL) != 0) {
      fprintf(stderr, "Error in mutex initialization\n");
      exit(EXIT_FAILURE);
    }
  base_time = time(NULL);
  srandom(base_time);
}
```

La funzione `initialize()` scrive inoltre nella variabile globale `base_time` l'ora corrente (come numero di secondi trascorso dal 1 gennaio 1970). Questo valore è utilizzato per inizializzare il seme del generatore di numeri pseudo-casuali. Inoltre la funzione di servizio `now()` utilizza `base_time` per restituire il numero di secondi trascorso dall'inizio dell'esecuzione del programma:

```
time_t now(void)
{
    return time(NULL) - base_time;
}
```

La funzione `create_phils()` si occupa di creare i cinque thread POSIX e fare in modo che ciascuno di essi "impersoni" uno dei cinque filosofi:

```
struct job {
  int thread_index;
  pthread_mutex_t *first_chopstick;
  pthread_mutex_t *second_chopstick;
};

struct job job_descr[num];

void create_phils(void)
{
  int i;
  pthread_t p;

  for (i = 0; i < num; i++) {
    job_descr[i].thread_index = i;
    job_descr[i].first_chopstick=chopsticks+first_mux[i];
    job_descr[i].second_chopstick=chopsticks+second_mux[i];
    if (pthread_create(&p, NULL, play_phil,
                       (void *)(job_descr + i)) != 0) {
      fprintf(stderr, "Error in POSIX thread creation\n");
      exit(EXIT_FAILURE);
    }
  }
}
```

Il lavoro da svolgere per ciascun thread è descritto da una struttura `struct job`, contenente l'indice del thread ed i puntatori ai mutex corrispondenti alla prima ed alla seconda bacchetta utilizzata dal filosofo.

Il cuore del programma è quindi la funzione `play_phil()`, che riceve come argomento l'indirizzo della struttura `struct job`:

```
void *play_phil(void *arg)
{
  struct job *job = (struct job *)arg;
  int phn = job->thread_index;
  for (;;) {
    printf("P%d: start thinking at %ld\n", phn, now());
    random_sleep();
    printf("P%d: stop  thinking at %ld\n", phn, now());
```

```
    acquire_mux(job->first_chopstick);
    acquire_mux(job->second_chopstick);
    printf("P%d: start eating   at %ld\n", phn, now());
    random_sleep();
    release_mux(job->second_chopstick);
    release_mux(job->first_chopstick);
  }
}
```

La funzione `random_sleep()` sospende il processo per un tempo casuale:

```
void random_sleep(void)
{
    const int max_sleep = 20; /* seconds */
    int delay = random() % max_sleep + 1;
    while (delay != 0)
        delay = sleep(delay);
}
```

Il generatore di numeri pseudo-casuale ha uno stato interno che è condiviso tra tutti i thread dell'applicazione; in questa applicazione questo non costituisce un problema: tutti i thread essenzialmente utilizzano una unica sequenza di numeri pseudo-casuali. In alternativa in Linux è possibile utilizzare la funzione di libreria `random_r()`, che consente a ciascun thread di definire una sequenza pseudo-casuale privata.

Si noti anche che la funzione di libreria `sleep()` può restituire zero se l'attesa è finita, oppure un valore positivo se l'attesa è stata interrotta dalla ricezione di un segnale. In quest'ultimo caso il ciclo `while` consente di riprendere l'attesa interrotta.

Per acquisire e rilasciare un mutex si utilizzano due semplici funzioni:

```
void acquire_mux(pthread_mutex_t * mux)
{
  if (pthread_mutex_lock(mux) != 0) {
    fprintf(stderr, "Error while acquiring the mutex\n");
    exit(EXIT_FAILURE);
  }
}
void release_mux(pthread_mutex_t * mux)
{
  if (pthread_mutex_unlock(mux) != 0) {
    fprintf(stderr, "Error while releasing the mutex\n");
    exit(EXIT_FAILURE);
  }
}
```