Sistemi Operativi **(M. Cesati)**

Compito scritto del 4 luglio 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 multithread C/POSIX facente uso di 100 thread. L'applicazione esegue il seguente algoritmo:

1. Tutti insieme i 100 thread cominciano a dormire per un tempo casuale (diverso da thread a thread) compreso tra 1 e 60 secondi.
2. Si misura il numero di secondi necessario affinché si risveglino 50 thread, e lo si stampa in standard output.
3. Dopo aver atteso il risveglio di tutti i 100 thread, si ricomincia dal passo 1.

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 in modo sintetico ed esauriente i principali meccanismi adottati da un moderno sistema operativo per ridurre il rischio di esaurimento della memoria centrale (RAM).

Sistemi Operativi (M. Cesati)

Esempio del programma del compito scritto del 4 luglio 2014

Si intende implementare il seguente algoritmo:

1. Tutti i thread attendono su una barriera con soglia pari a 100 (iniziano il passo successivo tutti insieme)
2. Tutti i thread salvano in memoria locale il numero di secondi attuale
3. Tutti i thread si auto-sospendono per un numero di secondi tra 1 e 60
4. Quando ciascun thread si risveglia:
   (a) incrementa una variabile globale `count` (inizializzata preventivamente a zero)
   (b) il thread che, dopo l'incremento, trova il valore di `count` uguale a 50 è il 50-esimo thread che si è risvegliato: scrive in standard output per quanto tempo ha dormito
   (c) il thread che, dopo l'incremento, trova il valore di `count` uguale a 100 è l'ultimo thread a risvegliarsi: riazzera la variabile `count` per il prossimo ciclo
   (d) il thread torna ad eseguire il passo 1

Svolgiamo l'esercizio seguendo un approccio "top-down". La funzione `main()` inizializza le strutture di dati, crea i 100 thread, e poi termina il thread originale del processo:

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

int main()
{
  initialize();
  create_threads(100);
  pthread_exit(0);
}
```

La funzione `initialize()` si occupa di inizializzare le strutture di dati utilizzate dal programma. È necessario definire una barriera, una variabile globale `count`, ed un mutex che eviti i problemi di race condition causati dall'accesso contemporaneo di più thread alla stessa variabile condivisa `count`.

```
pthread_barrier_t barrier;
int count = 0;
pthread_mutex_t mutex;

void initialize(void)
{
  srandom(time(NULL));
  if (pthread_barrier_init(&barrier, NULL, 100)) {
    fprintf(stderr, "Error in pthread_barrier_init()\n");
    exit(EXIT_FAILURE);
  }
  if (pthread_mutex_init(&mutex, NULL)) {
    fprintf(stderr, "Error in pthred_mutex_init()\n");
    exit(EXIT_FAILURE);
  }
}
```

La funzione `srandom()` inizializza il seme del generatore di numeri pseudo-casuali. Il seme iniziale è il numero di secondi corrispondente all'ora attuale di sistema. Se non venisse invocata `srandom()` la sequenza di numeri generata da `random()` sarebbe sempre la stessa.

La funzione `create_threads()` crea i 100 thread richiesti. Ciascuno di essi esegue la funzione `thread_loop()` e non termina mai.

```
void create_threads(int n)
{
  int i;
  pthread_t tid;
  for (i=0; i<n; ++i)
    if (pthread_create(&tid, NULL, thread_loop, NULL)!=0) {
      fprintf(stderr, "Error creating thread #%d\n", i);
      exit(EXIT_FAILURE);
    }
}
```

La funzione `thread_loop()` esegue l'algoritmo sopra descritto:

```
void *thread_loop(void *arg)
{
  unsigned long t0;
  arg = arg; /* unused */
  for (;;) {
    wait_threads(&barrier);
    t0 = time(NULL);
    random_sleep();
    lock();
    ++count;
```

```
    if (count == 50)
      printf("%lu\n", time(NULL)-t0);
    else if (count == 100)
      count = 0;
    unlock();
  }
}
```

La funzione `wait_threads()` forza il thread ad attendere sulla barriera:

```
void wait_threads(pthread_barrier_t *barrier)
{
  int rc = pthread_barrier_wait(barrier);
  if (rc != 0 && rc != PTHREAD_BARRIER_SERIAL_THREAD) {
    fprintf(stderr, "Error in pthread_barrier_wait()\n");
    exit(EXIT_FAILURE);
  }
}
```

La funzione `random_sleep()` sospende il thread per un numero casuale di secondi:

```
void random_sleep(void)
{
  long delay = (random() * 59)/RAND_MAX + 1;
  sleep(delay);
}
```

Per ottenere un numero casuale tra 1 e 60 è necessario scalare il valore ottenuto da `random()`. Poiché `random()` restituisce un valore tra 0 e `RAND_MAX`, l'espressione `(random() * 59)/RAND_MAX` assume un valore compreso tra 0 e 59. Si noti al contrario che l'espressione `(random()/RAND_MAX)*59` non da' il valore desiderato a causa dell'operazione di divisione *tra numeri interi* il cui risultato è quasi sempre uguale a zero.

Al posto della funzione `sleep()` sarebbe più corretto utilizzare la funzione `nanosleep()`. Infatti 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. In Linux comunque la funzione `sleep()` è basata su `nanosleep()` e non utilizza segnali. Un'altra alternativa è usare la funzione `usleep()`, in cui è necessario specificare il numero di microsecondi invece che di secondi. Su alcuni sistemi

POSIX, però, non è possibile utilizzare questa funzione per una attesa superiore ad un secondo. In Linux comunque questa limitazione non esiste.

Infine, le funzioni `lock()` e `unlock()` acquisiscono e rilasciano il mutex che protegge la variabile globale `count`:

```
void lock(void)
{
  if (pthread_mutex_lock(&mutex) != 0) {
    fprintf(stderr, "Error in pthread_mutex_lock()\n");
    exit(EXIT_FAILURE);
  }
}

void unlock(void)
{
  if (pthread_mutex_unlock(&mutex) != 0) {
    fprintf(stderr, "Error in pthread_mutex_unlock()\n");
    exit(EXIT_FAILURE);
  }
}
```