Esame di Programmazione Concorrente
appello del 3 luglio 2009

Nome: Cognome: Matricola:

Durata: 2 ore e 30 minuti. Non si può consultare documentazione. Usare solo ed esclusivamente i fogli che sono stati consegnati.

Quiz

Una sola risposta esatta a domanda. Non sono ammesse correzioni.
+1 risposta esatta; 0 nessuna risposta o risposta multipla; -1 risposta sbagliata

• Quale tecnica di gestione dell'interferenza attua il metodo `java.util.Collections.synchronizedMap(Map map)`?

- confinamento per metodo
- confinamento per oggetto
- confinamento per thread
- confinamento per sessione
- nessuna delle precedenti

• Si considerino degli strumenti di sincronizzazione implementati mediante attesa attiva oppure mediante attesa passiva e context-switching. Con riferimento alle prestazioni?

- quelli basati su attesa attiva sono sempre più efficienti
- quelli basati su attesa passiva e context-switching sono sempre più efficienti
- all'aumentare della frequenza delle sezioni critiche ed al diminuire della loro durata, quelli basati su attese passive e context-switching diventano via via più convenienti
- all'aumentare della frequenza delle sezioni critiche ed al diminuire della loro durata, quelli basati su attese attive diventano via via più convenienti
- nessuna delle precedenti

• Per quale motivo la tecnica di scrittura di codice thread-safe tramite confinamento per thread risulta particolarmente utilizzata nella progettazione di librerie grafiche concorrenti come `javax.swing`?

- perché consentono agevolmente la scrittura di codice thread safe anche da parte di programmatori che non conoscono la teoria della programmazione concorrente
- perché il numero totale di thread creati non supera le due unità
- perché si è in presenza di oggetti con uno stato di dimensioni rilevanti
- per contenere il numero totale di oggetti creati
- nessuna delle precedenti

• Con riferimento a `java.util.concurrent.ConcurrentHashMap`, quale delle seguenti affermazioni è corretta?

- Si tratta di una implementazione thread safe di `java.util.Map`

- Si tratta di una implementazione di `java.util.Map` *completamente sincronizzata*
- Tutti i metodi atomici che fornisce fanno parte dell'interfaccia `java.util.Map`
- Può presentare una forma "controllata" di interferenza
- nessuna delle precedenti

• Tra `java.lang.StringBuffer`, `String` e `StringBuilder` quale classi risultano thread safe grazie alla tecnica degli oggetti immutabili?

- `StringBuffer` e `String`
- solo `String`
- `StringBuffer` e `StringBuilder`
- solo `StringBuffer`
- nessuna delle precedenti

Esercizio Java

11 punti Si è deciso un intervento teso a migliorare le prestazioni di una libreria java mono-thread per il calcolo numerico operante su matrici *sparse*, ovvero per la maggior parte composte da valori nulli. In particolare si è deciso di migrare ad una piattaforma multiprocessore dotata di numerose CPU ($N > 32$) e di riscrivere in versione multi-thread un metodo `findNonZero()` per la ricerca di un qualsiasi elemento non nullo in una matrice sparsa di dimensioni enormi. Scrivere una classe che realizzi l'intervento implementando l'unico metodo di questa interfaccia:

```
public interface NonNullFinder {
  public Coord findNonZero(int m[][]);
}
```

dove `Coord` è una interfaccia utilizzata dalla libreria per restituire il risultato della ricerca, ovvero le coordinate dell'elemento trovato non nullo all'interno della matrice scandita:

```
public interface Coord {
  public int getRow(); // restituisce indice di riga
  public int getCol(); // restituisce indice di colonna
}
```

Si richiede esplicitamente di evitare: (*i*) ogni forma di attesa attiva ed ogni forma di interferenza (*ii*) l'elaborazione multipla degli stessi elementi dell'array da parte di thread diversi (*iii*) thread che sopravvivano all'esecuzione del metodo `NonNullFinder.findNonZero()` pur essendo stati creati al suo interno; (*iv*) la continuazione della ricerca da parte dei thread creati quando uno di loro ha già individuato un elemento non nullo.

```
import java.util.concurrent.*;
public class Executors {
  static ExecutorService newFixedThreadPool(int n);
  static ExecutorService newCachedThreadPool();
  static ExecutorService newSingleThreadExecutor()
}
public interface Executor {
  void execute(Runnable command);
}
public interface Callable {
  V call() throws Exception;
}
public interface Future<V> {
  V get();
  boolean isDone();
}
```

```
public interface ExecutorService
                implements Executor {
  Future<?> submit(Runnable task);
  Future<T> submit(Callable<T> task);
  void execute(Runnable command);
  void shutdown();
}

public class FutureTask<V>
             implements Runnable, Future<V>{
  FutureTask(Callable<V> callable);

  V get();
  boolean isDone();
}
```

Esercizio C

11 punti Un server *merger* riceve regolarmente da dei client chiamati *provider* delle misurazioni sensoriali. Ciascun *provider* è collegato ad un solo sensore, è univocamente identificato da un codice numerico e spedisce al server merger misurazioni effettuate ad intervalli di tempo regolari (ad es. una volta ogni 10 ms.) ma con frequenze variabili da sensore a sensore. Oltre ai client *provider* esiste una seconda tipologia di client, detti *reader*, che sono interessati a ricevere una sequenza di misurazioni ottenuta per "fusione" delle sequenze fornite da uno o più *provider*. In particolare, al momento della loro connessione specificano a quali *provider* sono interessati, e quindi cominciano a ricevere un'unica sequenza di coppie *(id, v)* risultante dalla fusione delle sequenze di misurazioni ricevute dai diversi client *provider* associati ai sensori di interesse; le coppie contengono rispettivamente il codice (*id*) identificativo del client *provider* da cui proviene la misura, ed il valore (*v*) della misurazione rilevata.

I termini di utilizzo del *server di merging* da parte dei client *provider* prevedono almeno questi punti:

- i client *provider* possono fornire misurazioni con frequenza diversa da client a client
- non appena un client *provider* si connette è tenuto a specificare il codice identificativo del suo sensore e subito dopo la sequenza di misurazioni rilevate a partire dal momento della connessione
- il codice identificativo del sensore è predeterminato e già noto al client *provider* prima della sua richiesta di connessione
- i client *provider* restano connessi sino a quando non richiedono esplicitamente al server merger la fine della loro sessione, ed in questo caso, il server provvede ad inviare tutti i valori ricevuti prima della disconnessione verso i client *reader* che si erano detti interessati al suo sensore
- se ci sono problemi di comunicazione con un client *provider*, la connessione verso tale client viene interrotta, il server perde traccia di quel *provider* ma si preoccupa di evadere tutte le misurazioni già pervenute

Il server merger è raggiungibile dai client *provider* all'indirizzo `M_ADDRESS` con porta `M_PORT_PROVIDER`.

I termini di utilizzo del *server di merging* da parte dei client *reader* prevedono almeno questi punti:

- non appena il client *reader* si connette, è tenuto a specificare il codice identificativo di tutti i *provider* ai quali si dichiara interessato
- non appena il client *reader* si connette, è tenuto a specificare il massimo ritardo che chiede di tollerare prima di essere disconnesso
- non appena il client *reader* si connette, il server si impegna a fornirgli i valori delle misurazioni a cominciare dalla prima resa disponibile successivamente alla connessione di tale client da uno qualsiasi dei client *provider* di interesse correntemente collegato

- il server restituisce le misure rispettando rigidamente l'ordine di ricezione da ciascun *provider* e quindi nel flusso di coppie che spedisce ai client non ci saranno misure che risultano invertite di ordine e/o non inviate e/o inviate più di una volta
- il server smista le nuove misure il più celermente possibile verso ogni client ma compatibilmente con le letture che questo ha già effettuato per non alternarne il naturale sequenziamento
- il server si impegna a fondere le misurazioni che riceve dai diversi *provider* cercando di minimizzare il ritardo tra la ricezione di una misurazione da un *provider* e la spedizione della stessa verso i client *reader* interessati
- se ci sono problemi di comunicazione con un client *reader* la connessione verso tale client viene interrotta, ed il server perde ogni traccia di quel client
- se un client *reader* è in ritardo per più del numero di misurazioni che ha specificato all'atto della sua connessione, viene disconnesso ed il server perde ogni traccia di quel client

Il server *merger* che fornisce il flusso continuo di coppie è raggiungibile dai client *reader* all'indirizzo `M_ADDRESS` con porta `M_PORT_READER`.

Impostare una soluzione concorrente per realizzare il servizio di merging su una piattaforma SMP dedicata con `NCPU` processori disponibili (dove `NCPU`$\geq$4) con il massimo grado di parallelismo possibile ed evitando ogni forma di attesa attiva e di interferenza. Prestare particolare attenzione alla robustezza delle propria soluzione all'aumentare del numero di client (sia *reader* che *provider*) e della frequenza di ricezione delle misurazioni dai client *provider*. Viene inoltre esplicitamente richiesto che eventuali rallentamenti nella lettura dei valori da parte di un client *reader* (rispett. nella spedizione di un valore da parte di un client *provider*) non causino ritardi nella spedizione dei valori verso gli altri client *reader* (rispett. nella ricezione dei valori dagli altri client *provider*).

Argomentare i vantaggi ed i limiti della propria soluzione rispetto a tutte queste problematiche.

a) 3 punti su 11

Descrivere quanti flussi di esecuzione sono creati e, per ciascun flusso di esecuzione, le sue interazioni con gli altri flussi. Indicare esplicitamente gli strumenti implementativi e gli eventuali meccanismi IPC utilizzati per realizzare tali flussi e loro interazioni (processi/thread; mutex/semafori/var.cond...; pipe/shm...).

b) 8 punti su 11

Implementare un'applicazione client/server scritta in C sotto Linux. Dettagliare il codice del solo *server di merging* trascurando il codice dei due tipi di client.

È possibile fare le seguenti ipotesi semplificative: i valori delle misurazioni ed i codici identificativi dei *provider* sono interi positivi agevolmente rappresentabili con il tipo primitivo `int`; ciascun client *reader* può essere interessato a ricevere le misurazioni risultanti dalla fusione di misurazioni provenienti da non più di `MAX_NUM_P` client *provider*; i codici identificativi dei sensori sono già noti ai client *provider* e *reader*, e comunque sono interi non negativi che non superano la costante `MAX_NUM_S`; è già disponibile un libreria `pc.h` per la comunicazione client/server delle misurazioni e dei codici identificativi dei *provider*:

```
#include <pc.h>
int sendInt(int socket, int measure); // spedisce misura/codice sulla socket
int receiveInt(int socket);           // riceve misura/codice dalla socket
```

Si osservi che tutte le funzioni sono bloccanti, ma sono state implementate in modo da garantire il ritorno e la restituzione del controllo al flusso di esecuzione invocante in un tempo finito anche in presenza di errori (in tal caso restituiscono `-1`).

È infine possibile, ma non necessario, utilizzare una semplice ed efficiente libreria per la gestione di liste di elementi. Si precisa che la libreria non è thread-safe.

```
#include <list.h>
//Gestione lista di elementi tipati puntatori a void (void *)
//Gli elementi non possono essere NULL
List*  allocList();                       //alloca una lista
void freeList(List *listPtr);             //dealloca una lista
void add(List *ptr, void *elementPtr);    //aggiunge in fondo un elemento
void remove(List *ptr, void *elementPtr);//rimuove un elemento
Iterator* createIterator(List *listPtr); //crea un iteratore sulla lista
void freeIterator(Iterator *itPtr);       //dealloca un iteratore
int hasNext(Iterator *itPtr);             //iterazione finita?
void *next(Iterator *itPtr);              //prossimo elemento, NULL se finiti
```

Segnature Si riportano di seguito le segnature di alcune chiamate di sistema che potrebbero risultare utili per lo svolgimento dell'esercizio. Si noti che si dovrà scegliere quali utilizzare ed eventualmente aggiungerne altre simili.

Socket

```
#include <netdb.h>
struct hostent *gethostbyname(const char *name);
struct hostent {
    char    *h_name;        /* official name of host */
    char    **h_aliases;    /* alias list */
    int     h_addrtype;     /* host address type */
    int     h_length;       /* length of address */
    char    **h_addr_list;  /* list of addresses */
}
#define h_addr  h_addr_list[0]  /* for backward compatibility */

#include <sys/types.h>
#include <sys/socket.h>
int socket(int domain, int type, int protocol);
int send(int s, const  void *msg, int len, unsigned int flags);
int sendto(int s, const void *msg, int len, unsigned int flags, const struct sockaddr *to, int tolen);
int recv(int s, void *buf, int len, unsigned int flags);
int recvfrom(int s, void *buf, int len, unsigned int flags, struct sockaddr *from, int *fromlen);

int bind(int sockfd, struct sockaddr *my_addr, int addrlen);
int connect(int sockfd, struct sockaddr *serv_addr, int addrlen);
int listen(int s, int backlog);
int accept(int s, struct sockaddr *addr, int *addrlen);

#include <netinet/in.h>
#include <arpa/inet.h>
char *inet_ntoa(struct in_addr in);

struct sockaddr {
    unsigned short sa_family; // address family, AF_xxx (AF_INET)
    char sa_data[14];         // 14 bytes of protocol address
};
struct sockaddr_in {
    short int sin_family;            // Address family
    unsigned short int sin_port;     // Port number
    struct in_ in_addr addr sin_addr; // Internet address
    unsigned char sin_zero[8];       // Same size as struct sockaddr
};
struct in_addr {
    unsigned long s_addr; // 32-bit long, 4 byte
};

unsigned long int htonl(unsigned long int hostlong); //htons(), htohl(), ntohs() sono analoghe
```

Segnali

```
#include <sys/types.h>
#include <signal.h>
typedef void (*sighandler_t)(int);
sighandler_t signal(int signum, sighandler_t handler);
int kill(pid_t pid, int sig); //sig: SIGUSR1, SIGUSR2, SIGALRM, SIGINT...
unsigned int alarm(unsigned int seconds);
```

Segmenti di Memoria Condivisa

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
int shmget(key_t key, int size, int shmflg);
key_t ftok(const char *pathname, int proj_id);
int shmctl(int shmid, int cmd, struct shmid_ds *buf); // cmd: IPC_STAT, IPC_SET, IPC_RMID
void *shmat(int shmid, const void *shmaddr, int shmflg);
int shmdt(const void *shmaddr);
```

Semafori

```
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/sem.h>
int semop(int semid, struct sembuf *sops, unsigned nsops);
int semget(key_t key, int nsems, int semflg);
int semctl(int semid, int semnum, int cmd, ...); // cmd: SET_VAL, IPC_RMID...
struct sembuf {...
   unsigned short sem_num;  /* semaphore number */
   short sem_op;            /* semaphore operation */
   short sem_flg;           /* operation flags: IPC_NOWAIT, SEM_UNDO */
...}
union semun {
  int val;                  /* value for SETVAL */
  struct semid_ds *buf;     /* buffer for IPC_STAT, IPC_SET */
  unsigned short *array;    /* array for GETALL, SETALL */
  /* Linux specific part: */
  struct seminfo *__buf;    /* buffer for IPC_INFO */
};
```

Thread

```
#include <pthread.h>
int pthread_create(pthread_t *thread, pthread_attr_t *attr, void * (*start_routine)(void *), void *arg);
void pthread_exit(void *retval);
int pthread_join(pthread_t th, void **thread_return);

pthread_mutex_t fastmutex = PTHREAD_MUTEX_INITIALIZER;
int pthread_mutex_init(pthread_mutex_t *mutex, const pthread_mutex-attr_t mutexattr);
int pthread_mutex_lock(pthread_mutex_t *mutex));
int pthread_mutex_trylock(pthread_mutex_t *mutex);
int pthread_mutex_unlock(pthread_mutex_t *mutex);
int pthread_mutex_destroy(pthread_mutex_t *mutex);

pthread_cond_t cond = PTHREAD_COND_INITIALIZER;
int pthread_cond_init(pthread_cond_t *cond, pthread_condattr_t *cond_attr);
int pthread_cond_signal(pthread_cond_t *cond);
int pthread_cond_broadcast(pthread_cond_t *cond);
int pthread_cond_wait(pthread_cond_t *cond, pthread_mutex_t *mutex);
int pthread_cond_timedwait(pthread_cond_t *cond, pthread_mutex_t *mutex, const struct timespec *abstime);
int pthread_cond_destroy(pthread_cond_t *cond);
```

```
#include <semaphore.h>
int sem_init(sem_t *sem, int pshared,
             unsigned int value);
int sem_wait(sem_t * sem);
int sem_trywait(sem_t * sem);
int sem_post(sem_t * sem);
int sem_getvalue(sem_t * sem, int * sval);
int sem_destroy(sem_t * sem);
```

Pipe

```
#include <unistd.h>
int pipe(int filedes[2]);
int dup(int oldfd);
int dup2(int oldfd, int newfd)

#include <stdio.h>
FILE *popen(const char *command, const char *type);
int pclose(FILE *stream);
```

I/O , Miscellanea

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
int open(const char *pathname, int flags);
int open(const char *pathname, int flags, mode_t mode);
int creat(const char *pathname, mode_t mode);
int close(int fd);
ssize_t read(int fd, void *buf, size_t count);
ssize_t write(int fd, const void *buf, size_t count);

#include <string.h>
void *memset(void *s, int c, size_t n);
```