Esame di Programmazione Concorrente - appello del 24 giugno 2008

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

• Si considerino n flussi di esecuzione che su una architettura multi-processore che eseguono indefinitivamente e ripetutamente la stessa identica sezione critica protetta da un semaforo binario di Dijkstra S implementato tramite spin-lock basato su una istruzione atomica di tipo *TestAndSet*:
loop P(S) <<*sezione critica*>> V(S) **end loop**
All'aumentare progressivo di n:

- non aumenta né il numero di flussi che sono sospesi in attesa passiva né il numero di quelli sono bloccati in attesa passiva di entrare in sezione critica
- non aumenta il numero di flussi che sono sospesi in attesa passiva mentre aumenta il numero di quelli sono bloccati in attesa passiva di entrare in sezione critica
- aumenta il numero di flussi che sono sospesi in attesa passiva ma non il numero di quelli sono bloccati in attesa attiva di entrare in sezione critica
- aumenta il numero di flussi che sono sospesi in attesa passiva ed il numero di quelli sono bloccati in attesa attiva di entrare in sezione critica
- nessuna delle precedenti

• Si consideri un algoritmo concorrente che riesce a parallelizzare una frazione $0 \leq P \leq 1$ del proprio lavoro. All'aumentare del numero n di processori fisici disponibili, come si comporta lo *speed-up*, ovvero il rapporto tra il tempo di esecuzione di tale algoritmo concorrente su una architettura mono-processore ed il suo tempo di esecuzione su una architettura multi-processore?

- aumenta indefinitivamente come n
- aumenta indefinitivamente come nP
- tende asintoticamente verso $1/(1-P)$
- tende asintoticamente verso $1/P$
- nessuna delle precedenti

• Una soluzione al problema dei cinque filosofi mangiatori che non presenta pericolo di stallo prevede la richiesta incrementale delle due forchette; quattro filosofi richiedono sistematicamente prima la forchetta di sx e poi quella di dx, mentre l'ultimo, mancino, richiede prima la forchetta di dx e poi quella di sx. A quale tecnica di gestione dello stallo è assimilabile questa soluzione?

- invalidazione della incrementalità delle richieste

- invalidazione della attesa circolare
- invalidazione della mutua esclusione
- invalidazione della non-prerilasciabilità
- nessuna delle precedenti

• Si considerino questi due programmi che violano le condizioni di Bernstein. I due programmi sono formati da istruzioni indivisibili. Quali delle possibili sequenze di interleaving risultanti da una loro esecuzione su di una architettura mono-processore evidenzieranno interferenza:

Primo Prg	*Secondo Prg*	*Sequenze di interleaving*
$r)A \leftarrow B;$	$x)E \leftarrow D;$	$1)xyrstz$
$s)P \leftarrow P-1;$	$y)F \leftarrow B+1;$	$2)rxsytz$
$t)C \leftarrow D;$	$z)P \leftarrow P+1;$	$3)xyrzst$

- nessuna
- la 1
- la 2
- la 3
- tutte
- nessuna delle precedenti

• Con riferimento alla proprietà di fairness di un algoritmo concorrente eseguito indefinitivamente (ad es. i *cinque filosofi pensatori*):

- per confutarla è sufficiente trovare una sequenza di esecuzione ammissibile infinitamente lunga in cui un flusso di esecuzione non avanza mai
- per confutarla è sufficiente trovare una sequenza di esecuzione ammissibile di lunghezza finita in cui un flusso di esecuzione non avanza mai
- per dimostrarla è sufficiente trovare una sequenza di esecuzione ammissibile infinitamente lunga in cui tutti i flussi avanzano
- per dimostrarla è sufficiente trovare una sequenza di esecuzione ammissibile di lunghezza finita in cui tutti i flussi avanzano
- nessuna delle precedenti

Esercizio Java

11 punti Si è deciso un intervento teso a migliorare le prestazioni di una libreria java mono-thread. In particolare si è deciso di migrare ad una piattaforma multiprocessore dotata di N CPU e di riscrivere in versione multi-thread un metodo `Finder.search(List<Element> list)` molto oneroso. Tale metodo permette di trovare uno qualsiasi degli elementi tra tutti quelli che all'interno di una lista di enormi dimensioni soddisfano un dato predicato booleano.

Scrivere una classe che realizzi l'intervento implementando l'unico metodo di questa interfaccia:

```
public interface Finder {
  public Element search(List<Element> list); // trova un *solo* risultato
}                              // se esiste; restituisce null altrimenti
```

dove `Element` è una interfaccia utilizzata dalla libreria per modellare gli elementi della lista che `search()` riceve come parametro, e `GoalTest` modella il predicato booleano da valutare su tali elementi:

```
public interface Element {
  //altri metodi omessi
}
```

```
public interface GoalTest {
  boolean test(Element element); // true sse
}                                // soddisfatto
```

Si richiede esplicitamente di evitare: (*i*) ogni forma di attesa attiva ed ogni forma di interferenza (*ii*) l'elaborazione multipla degli stessi elementi della lista da parte di thread diversi (*iii*) thread che sopravvivano all'esecuzione del metodo `Finder.search()` pur essendo stati creati al suo interno (*iv*) la continuazione della ricerca quando è già stato trovato un elemento da restituire come risultato. Di seguito sono riportate alcune delle interfacce e delle classi del package `java.util.concurrent` che potrebbero risultare utili allo scopo:

```
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();
  boolean cancel(boolean mayInterruptIfRunning);
}
```

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

public class ExecutorCompletionService<V>
             implements ExecutorService<V> {
  //Costruttore:
  public ExecutorCompletionService(Executor executor);

  Future<V>  poll();
  Future<V>  submit(Callable<V> task);
  Future<V>  submit(Runnable task, V result);
  Future<V>  take();
}
```

Esercizio C

11 punti Si vuole realizzare un server per un servizio di *alerting* sui valori rilevati da un singolo sensore. Il servizio di alerting consente a tutti gli interessati di specificare un valore di soglia e di venire tempestivamente informati ogni qualvolta il sensore la supera.

I valori sensoriali sono forniti al server di *alerting* da un server *provider* al quale il sensore è fisicamente connesso, e vengono inoltrati ad intervalli di tempo regolari (ad es. una volta ogni 10 millisecondi) sotto forma di valore numerico intero positivo. Tutti i client che vi si connettono ricevono una seq. di valori numerici corrispondenti alle misurazioni effettuate a partire dal momento della connessione.

I client del server *provider*, e quindi anche il server di *alerting* che in effetti si configura come un suo client, sono tenuti a leggere i valori che restituisce ad una velocità sufficiente, in quanto la sua politica è quella di interrompere la connessione se si accorge che un suo client non ha ancora letto l'ultima misura quando deve inviarne una nuova.

In particolare i termini di utilizzo del server di alerting prevedono almeno questi punti:

- non appena il client si connette, è tenuto a specificare un valore di soglia; il client è interessato a ricevere notifiche del superamento di tale valore di soglia da parte delle misurazioni ricevute dal provider
- i client possono specificare un valore di soglia diverso da client a client
- le notifiche che il server di alerting si impegna ad inviare ai client riportano questi dati: l'istante temporale in cui la soglia è stata superata, un numero progressivo delle notifiche inviate a quel determinato client e, ovviamente, il valore osservato

- fa fede, come istante di riferimento del superamento di una soglia, l'istante di ricezione all'interno del server di alerting della relativa misurazione proveniente dal provider; tale istante viene assegnato autonomamente dal server di alerting
- il numero progressivo delle notifiche comincia da zero ed è relativo alle sole notifiche spedite ad un determinato client
- durante una sessione i client non possono cambiare in nessun modo il valore di soglia specificato all'inizio della loro sessione
- il server si impegna a fornire le notifiche relative a *tutti* i superamenti del valore di soglia specificato avvenute durante la sessione; le verifiche sono effettuate a cominciare dall'ultimo valore ricevuto dal provider (relativo alla misurazione più recente)
- un valore di soglia si considera superato ogni qualvolta si susseguono due misurazione consecutive, una minore od uguale alla soglia ed una strettamente maggiore
- i client restano connessi sino a quando non richiedono al server di alerting esplicitamente la fine della loro sessione, ed in questo caso, le eventuali notifiche pendenti al momento della richiesta sono comunque evase
- il server smista le notifiche il più celermente possibile ma è tenuto a mantenere il loro naturale sequenziamento temporale

Gli utenti interessati ad usufruire del servizio possono utilizzare un apposito client che consente di specificare un solo valore di soglia, di produrre messaggi informativi in seguito alla ricezione di notifiche e di richiedere esplicitamente la fine della sessione e la conseguente disconnessione dal server.

Impostare una soluzione concorrente per realizzare il servizio di alerting 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 e della frequenza di ricezione delle misurazioni dal provider. Argomentare i vantaggi ed i limiti della propria soluzione in tal senso.
a) 3 punti su 11
descrivere quanti e quali tipologie di flussi di esecuzione sono creati e, per ciascun flusso di esecuzione, le sue interazioni con gli altri flussi; chiarire il protocollo (sintassi e semantica) alla base dei messaggi scambiati tra server e client; chiarire la natura e la funzione delle strutture dati a supporto della propria soluzione.
b) 8 punti su 11
implementare un'applicazione client/server scritta in C sotto Linux; dettagliare il codice del solo *server di alerting* trascurando sia il codice del client che del provider; indicare esplicitamente gli strumenti implementativi e gli eventuali meccanismi IPC utilizzati per realizzare i flussi di esecuzione, disciplinare le loro interazioni ed i loro accessi alle strutture dati condivise.

È possibile fare le seguenti ipotesi: il server *alerter* fornisce ai client interessati i suoi servizi all'indirizzo `AL_ADDRESS` con porta `AL_PORT`; il *provider* è raggiungibile

all'indirizzo `PR_ADDRESS` sulla porta `PR_PORT`; i valori delle misurazioni, le soglie ed i numeri progressivi delle notifiche sono interi positivi agevolmente rappresentabili con il tipo primitivo `int`; gli istanti temporali sono rappresentabili con il tipo `time_t`; è già disponibile un libreria `pc.h` per la comunicazione client/server:

```
#include <pc.h>
int sendInt(int socket, int measure); // spedisce misura/progressivo sulla socket
int receiveInt(int socket);           // riceve misura/progressivo dalla socket
int sendTime(int socket, time_t time);     //spedisce un istante temporale
int receiveTime(int socket, time_t *time); //riceve un istante temporale
```

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_t*  allocList();                      //alloca una lista
void freeList(list_t *list);               //dealloca una lista
void add(list_t *list, void *element);     //aggiunge in fondo un elemento
void remove(list_t *list, void *element); //rimuove un elemento
iterator_t* createIterator(list_t *list); //crea un iteratore sulla lista
void freeIterator(iterator_t *it);         //dealloca un iteratore
int hasNext(iterator_t *it);               //iterazione finita?
void *next(iterator_t *it);                //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 comp. */

#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 f);
int recv(int s, void *buf, int len, unsigned int flags);

int bind(int sockfd, struct sockaddr *my_addr, int adlen);
int connect(int sockfd, struct sockaddr *serv_a, int adlen);
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; // addr. family, e.g. AF_INET
  char sa_data[14];         // 14 bytes of protocol addr.
};
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
```

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

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);
```

Time, I/O , Miscellanea

```
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <time.h>

time_t time(time_t *t); //restituisce tempo corrente
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);
```