Esame di Programmazione Concorrente - appello del 21 giugno 2006

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 consideri una soluzione al problema dei cinque filosofi mangiatori in cui i filosofi richiedono incrementalmente le 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:

- ◦ la soluzione è immune al problema dello stallo
- ◦ la soluzione presenta pericolo di stallo
- ◦ la soluzione gode della proprietà di fairness
- ◦ la soluzione non può comportare starvation
- ◦ nessuna delle precedenti

• A quale delle seguenti affermazioni è equivalente l'assunzione di progresso finito con riferimento alla correttezza di un programma concorrente?

- ◦ è possibile costruire soluzioni concorrenti corrette basandosi sull'uso di pause di lunghezza predeterminata
- ◦ bisogna evitare ogni forma di attesa attiva
- ◦ anche le sequenze di interleaving rare possono compromettere la correttezza
- ◦ i fenomeni di interferenza non si devono manifestare solo per le sequenze di interleaving più frequenti
- ◦ nessuna delle precedenti

• La tecnica di gestione dell'interferenza tramite confinamento per thread prevede:

- ◦ che un solo thread alla volta acceda alla risorsa condivisa
- ◦ che nessun thread acceda alla risorsa condivisa
- ◦ che tutti i thread possono accedere alla risorsa sincronizzandosi
- ◦ che può accedere alla risorsa un unico thread a cui devono rivolgersi tutti gli altri
- ◦ nessuna delle precedenti

• Quale tecnica di gestione dell'interferenza attua il metodo `java.util.Collections.synchronizedList(List list)`?

- ◦ Confinamento per metodo
- ◦ Confinamento per oggetto
- ◦ Confinamento per thread
- ◦ Confinamento per sessione
- ◦ nessuna delle precedenti

• È possibile che si manifestino fenomeni di interferenza durante l'esecuzione di un costruttore di una classe java?

- ◦ no mai, purché non si facciano operazioni di lettura nel costruttore
- ◦ si, se un riferimento all'oggetto creato è già posseduto da un altro thread prima della terminazione del costruttore
- ◦ no mai, perché thread diversi da quello che ha invocato il costruttore non possono ottenere un riferimento all'oggetto creato prima della terminazione del costruttore
- ◦ si, se un riferimento all'oggetto creato è già posseduto da un altro thread prima dell'invocazione del costruttore
- ◦ nessuna delle precedenti

Esercizio Java

11 punti Si è deciso un intervento teso a migliorare le prestazioni di un programma 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 `getAvg()` per il calcolo della media di numeri interi associabili agli oggetti contenuti in un array di dimensioni enormi.

Scrivere una classe che realizzi l'intervento implementando l'interfaccia:

```
public interface ComputeAvg {
  public int computeAvg(Element[]); //calcola la media per l'array
}
```

Il calcolo del numero intero da associare a ciascun elemento è svolto applicando un metodo statico della classe `Computer` computazionalmente non oneroso agli oggetti contenuti nell'array

```
public static int Computer.compute(Element e); //non oneroso
```

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 `computeAvg()` pur essendo stati creati al suo interno.

```
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 *provider* fornisce ad intervalli di tempo regolari (ad es. una volta ogni 10 millisecondi) un valore numerico intero positivo scaturito da una misurazione tramite sensori. 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 devono leggere i valori restituiti dal *provider* ad una velocità sufficiente, in quanto la politica del *provider* è quella di interrompere la connessione con un suo client se si accorge che non ha ancora letto l'ultima misura quando deve inviarne una nuova.

È nata l'esigenza di nascondere il *provider* dietro un servizio di *buffering* che memorizzi le ultime `M` misurazioni (dove `M` può essere un numero molto grande) e sia capace di gestire concorrentemente un numero di client anche molto rilevante.

In questo modo i client possono connettersi al nuovo servizio che rispetto all'originale impone dei tempi di lettura meno stringenti da parte dei client. In particolare i termini di utilizzo del *server di buffering* prevedono almeno questi punti:

- non appena il client si connette, il server si impegna a fornire i valori a cominciare dall'ultimo disponibile (relativo alla misurazione più recente)
- il server restituisce le misure rispettando rigidamente l'ordine progressivo con il quale le riceve dal *provider* e quindi non ci saranno misure invertite di ordine e/o non inviate e/o invii doppi
- 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
- se un client è talmente in ritardo da essere rimasto indietro per più di `M` misurazioni, la connessione verso il client viene interrotta, ed il server perde ogni traccia di quel client
- se ci sono problemi di comunicazione con un client, la connessione verso tale client viene interrotta, ed il server perde ogni traccia di quel client

Il server *provider* che fornisce il flusso continuo di misure è raggiungibile all'indirizzo `P_ADDRESS` con porta `P_PORT`, mentre il *server di buffering* ha indirizzo `B_ADDRESS` con porta `B_PORT`.

Impostare una soluzione concorrente per realizzare il servizio di buffering con il massimo grado di parallelismo possibile ed evitando ogni forma di attesa attiva o di interferenza come segue:
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 buffering* trascurando sia il codice dei client che del *provider*.

È possibile fare le seguenti ipotesi semplificative: i valori delle misurazioni sono rappresentabili con il tipo primitivo `int`; è già disponibile un libreria `pc.h` per la comunicazione client/server delle misurazioni, che fornisce le seguenti funzioni:

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

Si osservi che entrambe 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 situazioni anomale e 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);
```