PCAD a.a. 2017/18 - Scritto del 15 giugno 2018

L'esame è composto da 12 domande a risposta multipla e 1 a risposta libera.
Nelle prime 12 domande bisogna indicare se le affermazioni sono vere o false.
Se avete sulla domanda aggiungete una breve spiegazione per giustificare la risposta.
Nella stessa domanda ci possono essere zero o più affermazioni vere.

D1 (1 punto) Nei modelli della concorrenza con consistenza forte (strong consistency):
1. Vale sempre la single-thread rule
2. È possibile che istruzioni all'interno dello stesso thread vengano eseguite out-of-order
3. Non viene garantita la write-atomicity nelle operazioni di update
4. Thread differenti non possono modificare le stesse aree di memoria

Risposte 1: Falso, 2: Falso, 3: Falso, 4: Falso

D2 (1 punto) L'assenza di race-condition
1. si ottiene dichiarando thread che non allocano dati sullo heap
2. si ottiene creando thread che allocano memoria solo attraverso variabili locali
3. si ottiene garantendo che i dati siano letti o modificati solo all'interno di singoli thread
4. si ottiene acquisendo lock su dati condivisi tra diversi thread

Risposte 1: Falso, 2: Falso, 3: Vero, 4: Vero

D3 (1 punto) Quando si utilizza un monitor alla Hoare in Java
1. i thread non condividono memoria ma vengono usati solo per parallelizzare il calcolo
2. i context-switch avvengono solo all'uscita dal monitor
3. se un thread sospende l'esecuzione con "wait" entrerà nel monitor alla prima chiamata di "notify"
4. un thread prima di chiamare "wait" rilascia il lock sul monitor

Risposte 1: Falso, 2: Falso, 3: Falso, 4: Falso (è la chiamata stessa di wait che deve rilasciare il lock)

D4 (1 punto) Un ReadWrite lock
1. è un semaforo binario che viene usato su oggetti con metodi getter e setter
2. viene usato per garantire la mutua esclusione tra thread che aggiornano una variabile condivisa
3. viene usato per garantire la mutua esclusione tra thread che leggono una variabile condivisa
4. garantisce starvation-freedom se usato per controllare una risorsa condivisa

Risposte 1: Falso, 2: Vero, 3: Falso (posso leggere simultaneamente),
4: Vero (proprietà delle soluzioni al problema della sezione critica)

D5 (1 punto) Un Array CopyOnWrite in Java
1. viene usato per ottimizzare le operazioni di update di celle di un array
2. implementa un'istanza del produttore-consumatore
3. fornisce solo operazioni thread-safe per modificare un array concorrente
4. implementa la tecnica dello snapshot per strutture dati concorrenti

Risposte 1: Falso, 2: Falso, 3: Falso, 4: Vero

D6 (1 punto) La tecnica di programmazione chiamata lock-splitting
1. viene usata per includere in un singolo record diversi semafori binari
2. viene usata per massimizzare la concorrenza mantenendo consistenza dei dati
3. viene usata per minimizzare il numero di lock in un programma concorrente
4. viene usata nell'implementazione di liste concorrenti

Risposte 1: Falso, 2: Vero (si cerca di parallelizzare accesso a strutture dati complesse come liste ecc), 3: Falso, 4: Vero (abbiamo visto esempi a lezione)

D7 (2 punti) Le soluzioni al problema della sezione critica viste a lezione
1. possono essere composte da thread che non usano mai la risorsa condivisa da proteggere
2. si possono implementare senza usare lock o altre istruzioni atomiche specifiche dell'hw
3. sono sempre corrette indipendentemente dall'architettura hw sottostante
4. dipendono dalla velocità nell'esecuzione dei diversi thread

Risposte 1: Falso (avremmo casi di starvation e/o deadlock!), 2: Vero (visto all'inizio del corso es Alg. Peterson, Alg. Lamport), 3: Falso (primo esempio corso), 4: Falso (non dipendono da vel. relativa)

D8 (2 punti) In un programma concorrente
1. un errore su un certo input si può riprodurre ripetendo l'esecuzione una sola volta
2. la funzione calcolata dal programma associa ad un certo input un insieme di output
3. con lo stesso input posso avere un'esecuzione che termina e una che non termina
4. con lo stesso input posso avere un'esecuzione che termina e una che si blocca

Risposte 1: Falso, 2: Vero, 3: Vero, 4: Vero

D9 (2 punti) Nell'esecuzione di un programma concorrente
1. i thread vengono sempre eseguiti su core diversi
2. i thread vengono sempre eseguiti in parallelo
3. non avvengono mai context-switch sullo stesso core
4. il numero di context-switch dipende dal numero di lock usati nel programma

Risposte 1: Falso, 2: Falso, 3: Falso, 4: Falso

D10 (2 punti) Consideriamo un programma concorrente con due thread T1 e T2
1. servono almeno due lock per generare un deadlock nell'esecuzione di T1 e T2
2. con un solo lock è possibile fare in modo che T1 venga eseguito tutto insieme prima o dopo T2
3. servono almeno due lock per fare in modo che T1 venga sempre eseguito prima di T2
4. il numero di possibili esecuzioni dipende solo dal numero di istruzioni che operano su dati condivisi

Risposte 1: Falso, 2: Vero, 3: Falso, 4: Falso

D11 (2 punti) Nella libreria RMI
1. L'accesso ad un oggetto remoto è sempre thread-safe
2. Il registry viene gestito dallo stesso server che gestisce un oggetto remoto
3. Il registry serializza le chiamate dei metodi verso un oggetto remoto
4. L'implementazione di un'interfaccia remota deve essere la stessa su server e client

Risposte 1: Falso, 2: Falso, 3: Falso, 4: Falso

D12 (6 punti) Considerate il seguente programma multithreaded MT

array A[3]={0,0,0}; x=0; // gli indici di A partono da 0
THREAD P: while (x<3) { A[x]=1; x++ }
THREAD Q: while (x<3) { A[x]=2; }
THREAD R: while (x<2) { A[x]=3; x=0 }

1. Alla fine dell'esecuzione A può contenere {1,1,1} (spiegare risposta)

Risposta: Vero, P può mettere tutto a 1 e incrementare x fino a 3

2. Alla fine dell'esecuzione A può contenere {1,2,3} (spiegare risposta)

Risposta: Falso, è possibile scrivere 3 in ultima posizione tramite R ma in tal caso x viene resettata e il programma non termina

3. Alla fine dell'esecuzione A può contenere {2,3,1} (spiegare risposta)

Risposta: Falso, per scrivere 3 e far terminare R occorre resettare x e quindi rischedulare P ecc (il programma non termina)

4. Il programma potrebbe non terminare (spiegare risposta)

Risposta: Vero, basta non schedulare mai P o schedulare infinite volte R fino al reset

Esercizio (10 punti)

Supponiamo di avere a disposizione una funzione
int HTTPconnect(String Address)
che restituisce
(1) il tempo in millisec richiesto per aprire una connessione HTTP
all'URL Address passato come parametro;
(2) -1 in caso di fallimento della richiesta.

Sia A un array di N posizioni inizializzato con stringhe che rappresentano URL.

Scrivere un algoritmo concorrente per
(1) parallelizzare, attraverso chiamate asincrone, la connessione ad ogni URL in A
(2) calcolare quindi il tempo medio di connessione scartando le connessioni non riuscite.

Possibile soluzione:

SHARED DATA STRUCTURES

```
int array URL[N] = {…..};  // inizializzato
DIM=N/K //dimenzione delle partizioni assumiamo N sia divisibile per K e K<=N
int array TIME[N];  tempi connessioni//
COUNTER=0; //assumiamo sia intero con operazioni atomiche
```

THREAD DEFINITIONS

```
THREAD MONITOR {
    while (COUNTER < N) {};
    var AUX=0,C=0;
    for i=0 to N  if TIME[i]>=0  then AUX=AUX+TIME[i]; C++;
    if (AUX>0) print(AUX/C);  else print(0);
 }

THREAD T[INDEX] { // con INDEX: 0,…,K
   // ogni thread esegue sequenzialmente N/K connessioni
   var AUX=DIM*INDEX;
   var LIMIT=DIM*(INDEX+1)

   for i=AUX to LIMIT  do TIME[i]=HTTPconnect(A[i]);   // bloccante
   COUNTER++;
}
```