Corso di Sistemi Operativi e Reti

Corso di Sistemi Operativi

Prova scritta di Febbraio 2017

ISTRUZIONI

1. **Rinomina** la cartella chiamata "CognomeNomeMatricola" che hai trovato sul Desktop e in cui hai trovato questa traccia, sostituendo "Cognome" "Nome" e "Matricola" con i tuoi dati personali;
2. **Carica** tutto il materiale didattico che vorrai usare sul Desktop; puoi farlo solo nei primi 5 minuti della prova;
3. **Svolgi** il compito; lascia tutto il sorgente che hai prodotto nella cartella di cui al punto 1;
4. Quando hai finito lascia la postazione facendo logout, **senza spegnere il PC**.

SALVA SPESSO il tuo lavoro

PER GLI STUDENTI DI SISTEMI OPERATIVI: si può sostenere solo uno dei due esercizi se si è già superata in un appello precedente la corrispondente prova. Il tempo a disposizione in questo caso è di 2 ORE.

ESERCIZIO 1 (Programmazione multithread. Punti: 0-20)

Bisogna implementare una struttura dati che chiameremo `VersionedArray`.
Un `VersionedArray` è una collezione di oggetti di tipo `T` che si comporta similmente a un comune array, ma consente anche di accedere a precedenti valori storici dell'array stesso; in particolare l'array viene memorizzato in più "versioni" ognuna delle quali numerata a partire da 0.
Un `VersionedArray` tiene traccia di una `versioneCorrente`, che è l'ultima in ordine temporale ed è quella a cui si accede di default. Le operazioni prescritte di seguito su un `VersionedArray` devono essere thread-safe secondo le regole prescritte.

`startRead(int v)`, `endRead(int v)`, `startRead()`, `endRead()`: prende/rilascia il lock in lettura sulla versione `v`. Laddove `v` non sia specificata si prende il lock sulla `versioneCorrente`. Più thread possono leggere in contemporanea sul `VersionedArray`. Le operazioni generano un'eccezione se si prova ad accedere a una versione non esistente.

`T get(int i)`: restituisce il valore dell'elemento `i`-esimo nel `VersionedArray`, prelevato dalla versione precedentemente selezionata con il metodo `startRead`. Genera un'eccezione se non si possiede già un read lock.

`startWrite()`: **CREA UNA NUOVA VERSIONE TEMPORANEA DELL'ARRAY A PARTIRE DA QUELLA CORRENTE** e ne prende il lock in scrittura. Solo un thread per volta può effettuare questa operazione, la quale deve poter essere effettuata anche se dovessero esserci in quel momento altri lettori sulla `versioneCorrente` o sulle precedenti.

`set(int i, T val)`: imposta nella versione temporanea corrente il valore `i`-esimo a `val`. Genera un'eccezione se non si possiede il lock in scrittura.

`endWrite()`: termina le operazioni di scrittura sulla nuova versione temporanea, la quale deve diventare l'ultima `versioneCorrente`. Non è necessario attendere che eventuali lettori terminino le operazioni sulle versioni già esistenti, inclusa l'ultima versione (che però diventa penultima).

Le problematiche di starvation possono essere trascurate, ma non quelle di Thread safety e potenziale deadlock. Si noti che non è prevista la scrittura su vecchie versioni del VersionedArray.

BONUS: si provi a implementare il `VersionedArray` cercando di risparmiare memoria, ad esempio memorizzando per ogni versione solo le differenze rispetto alle versioni precedenti.

CI SONO DEI PUNTI AMBIGUI NELLA TRACCIA? COMPLETA TU

È parte integrante di questo esercizio completare le specifiche date nei punti non esplicitamente definiti, introducendo o estendendo tutte le strutture dati laddove si ritenga necessario, e risolvendo eventuali ambiguità.

POSSO CAMBIARE IL PROTOTIPO DEI METODI RICHIESTI? NO

Non è consentito modificare il prototipo dei metodi se questo è stato fornito. Potete aggiungere qualsivoglia campo e metodo di servizio, e qualsivoglia classe ausiliaria

CHE LINGUAGGIO DEVO USARE? JAVA 7 O SUCCESSIVO

Il linguaggio da utilizzare per l'implementazione è Java. *È* consentito usare qualsiasi funzione di libreria di Java 7 o successivi.

MA IL MAIN() LO DEVO SCRIVERE? E I THREAD DI PROVA? SOLO PER FARE IL TUO DEBUG

Non è esplicitamente richiesto di scrivere un `main()` o di implementare esplicitamente del codice di prova, anche se lo si suggerisce per testare il proprio codice prima della consegna.

ESERCIZIO 2 (Linguaggi di scripting. Punti 0-10)

Il comando `history` mostra la lista di tutti i comandi eseguiti su una command line ed è uno dei comandi più utilizzati sia dagli utenti, per ricordare e ri-utilizzare un comando eseguito in precedenza, sia dagli amministratori di sistema, per avere informazioni riguardo i comandi che sono stati eseguiti prima di un determinato evento.

Esempio 1

Output del comando `history`

```
1  ping google.com
2  cd /var/tmp
3  ls
4  ls -lrt
5  vi robots.txt
6  ls -lrt
7  cd ..
8  tail -f /var/log/messages
```

In questo esercizio bisogna realizzare uno script Perl che prende un parametro da riga di comando (che chiameremo `NAME`); esegua il comando `history`, calcoli alcune informazioni riguardo `NAME` e stampi su Standard Output queste informazioni secondo il formato riportato di seguito.

Nota: per motivi tecnici si suggerisce di usare il comando "`cat ~/.bash_history | nl`" che produce lo stesso risultato di `history`, quando quest'ultimo viene eseguito su un terminale appena aperto.

Lo script deve tenere traccia di quante volte il comando `NAME` è presente nell'output del comando e di quali sono stati i parametri con cui è stato eseguito e deve visualizzare queste informazioni su Standard Output nel formato riportato di seguito. Sono da intendersi come "parametri con cui è stato eseguito il comando" tutto ciò che segue lo spazio successivo al nome del comando.

Formato Output

L'output deve contenere il valore di `NAME` seguito da uno spazio e dal numero di occorrenze (**n**) e poi, nelle **n** linee successive, dovrà riportare tutti i parametri con cui il comando è stato usato, nell'ordine in cui appaiono nell'output del comando, ognuno preceduto da un carattere di tabulazione (TAB). A riguardo si veda l'***Esempio 2***.

Esempio 2

Si assuma che l'output del comando sia quello riportato nell'***Esempio 1***.

Se lo script Perl (`script.pl`) viene invocato con `cd` come valore di `NAME`, cioè così:

```
./script.pl cd
```

l'output dovrà essere:

```
cd 2
    /var/tmp
    ..
```

Se lo script Perl (`script.pl`) viene invocato con `ls` come valore di `NAME`, cioè così:

```
./script.pl ls
```

l'output dovrà essere:

```
ls 3

    -lrt
    -lrt
```

Se lo script Perl (`script.pl`) viene invocato con `vi` come valore di `NAME`, cioè così:

```
./script.pl vi
```

l'output dovrà essere:

```
vi 1
    robots.txt
```

Se lo script Perl (`script.pl`) viene invocato con `cp` come valore di `NAME`, cioè così:

```
./script.pl cp
```

l'output dovrà essere:

```
cp 0
```

Si noti che se un comando non è seguito da nulla bisogna stampare comunque uno rigo bianco. Se invece un comando non è presente nella history, bisogna semplicemente stampare il nome del comando seguito da 0 (senza lasciare nessuna riga).