PCD Assignment #01
Report

Simone Ceredi

13 aprile 2023

Indice

Capitolo 1

Analisi

L'obiettivo del progetto è la realizzazione di un programma concorrente che esplori una directory, e sottodirectory, presente sul file system locale contenente un insieme di sorgenti Java, ed in seguito visualizzare alcune statistiche relative all'esplorazione effettuata.

1.1 Requisiti

Primo punto

- Il software dovrà effettuare l'esplorazione di una directory D presente sul file system locale
- Tutte le sottodirectory dovranno essere esplorate ricorsivamente
- Dovranno essere letti tutti i file Java per stabilirne il numero di linee di codice
- Sarà necessario mostrare in standard output:
  - Gli N sorgenti con il maggior numero di linee di codice
  - La distribuzione complessiva relativa a quanti sorgenti hanno un numero di linee di codice che ricade in un certo intervallo, considerando un certo numero di intervalli NI e un numero massimo MAXL di linee di codice per delimitare l'estremo sinistro dell'ultimo intervallo.

Secondo punto

Sarà necessario estendere il programma al punto primo includendo una GUI con:

- Input box per specificare i parametri
- Pulsanti start/stop per avviare/fermare l'elaborazione
- Una view ove visualizzare interattivamente l'output aggiornato, mediante 2 frame:
  - un frame relativo ai file con il maggior numero di linee di codice
  - un frame relativo alla distribuzione

1.2 Analisi e modello del dominio

La soluzione dovrà essere basata su programmazione multi-threaded, adottando sia principi e metodi di programmazione utili per favorire tutte le proprietà fondamentali della programmazione ad oggetti, che a massimizzare performance e reattività. Il programma dovrà quindi:

1. Analizzare la directory data ed esplorarla assieme a tutte le sottodirectory, andando nel frattempo a memorizzare i vari file incontrati.
2. Leggere tutti i file e memorizzarne il numero di linee di codice.
3. Elaborare la lunghezza di ogni file per estrarne le 2 metriche di interesse.
4. Visualizzare le 2 metriche.

Per massimizzare le performance del programma sarà necessario parallelizzare il più possibile i task descritti sopra. Si osserva come questi task siano indipendenti l'uno dall'altro, l'unico ordinamento necessario è ovvio, ovvero:

- Se il primo task non ha ancora trovato alcun file Java il secondo non può partire.
- Se il secondo non ha ancora letto il numero di linee di codice di alcun file Java il terzo non può partire.
- Fino a quando l'elaborazione del numero di linee di codice di un file non è stata completata non è possibile visualizzarne il risultato.

Il progetto avrà quindi in esecuzione contemporanea i 4 task descritti sopra. Ogni task potrà essere svolto da uno o più thread, tranne quello di visualizzazione delle statistiche che sarà svolto dall'EDT nel secondo punto, mentre nel primo gli sarà assegnato un singolo thread in quanto la scrittura dei dati in Standard Output richiede tempo e andrebbe quindi a rallentare la computazione degli altri thread.

La difficoltà del progetto sta nel permettere l'interazione tra i vari thread nel modo corretto andando ad evitare situazioni di deadlock, starvation e livelock.

Capitolo 2

Design

I task da eseguire sono quindi:

Task 1: Esplorazione di una directory

Task 2: Lettura delle linee di codice di un file Java

Task 3: Elaborazione della lunghezza di un file

Task 4: Visualizzazione dei dati elaborati

Siccome ogni task va a produrre dei dati che i task successivi vanno ad elaborare il sistema è stato implementato in modo simile al problema dei produttori-consumatori.

- Il Task 1 consuma una directory presente nel suo buffer andando a leggere ciò che contiene e producendo quindi un insieme di file Java e sottodirectory. I file Java vengono poi aggiunti al buffer del Task 2, mentre le sottodirectory vengono aggiunte al buffer del Task 1 stesso.
- Il Task 2 consuma un file Java presente nel buffer e produce come output dell'elaborazione il numero di linee di codice al suo interno, questo valore va poi inserito nel buffer del Task 3.
- Il Task 3 elabora il numero di righe aggiornando il relativo intervallo e in caso la lista di file più lunghi.
- Il Task 4 viene notificato di una modifica effettuata dal Task 3 e va quindi a visualizzare i valori aggiornati a video.

Il pattern architetturale scelto è MVC.

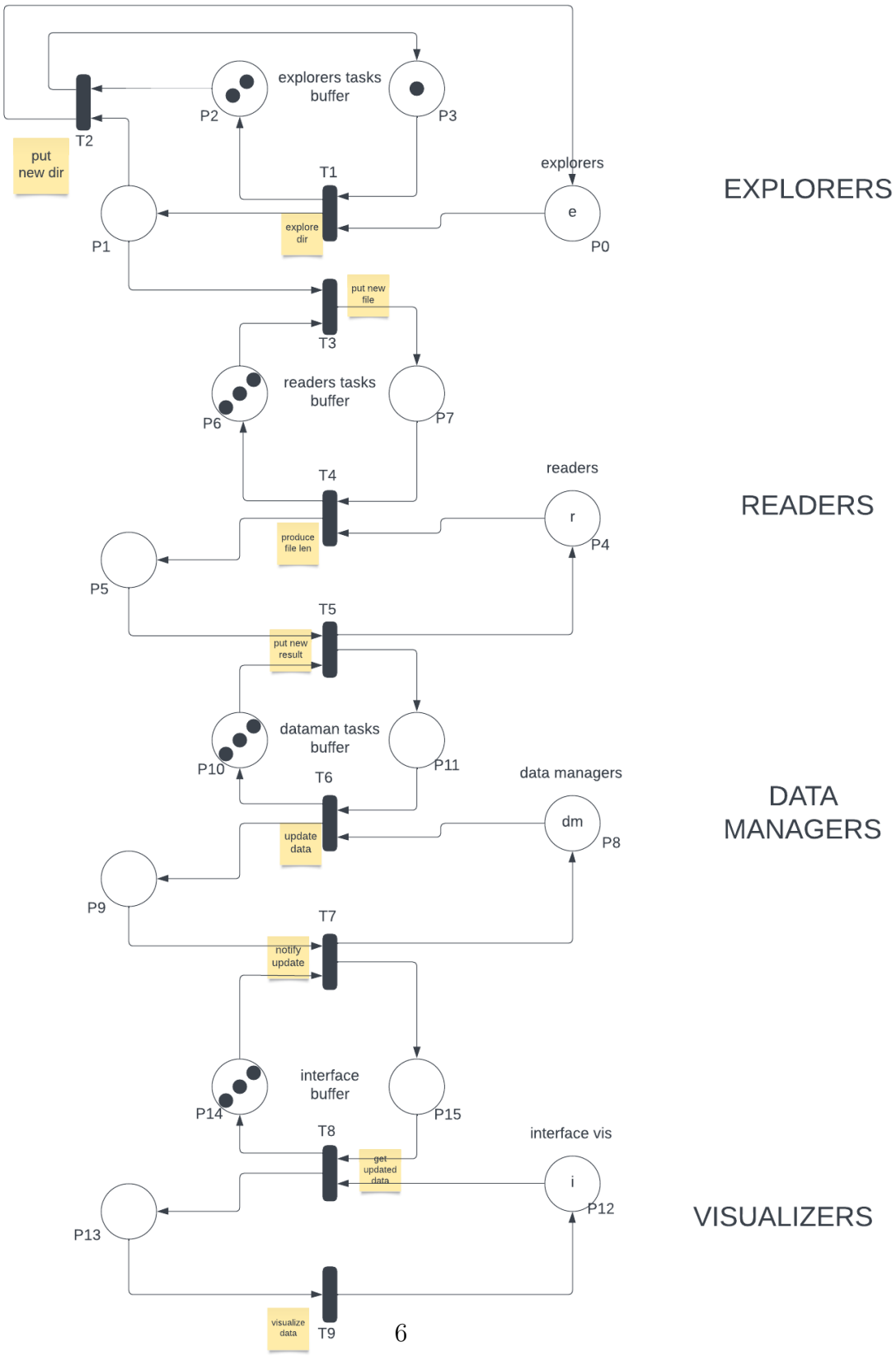

Figura 2.1: Rappresentazione tramite Reti di Petri dei task in esecuzione sul programma

2.1 Architettura

Unica nota riguardante il pattern MVC è la presenza all'interno del Model di una lista di observer, questo sarà utile per notificare le modifiche che avvengono nel model alle 2 view presenti nel progetto, quella testuale, riguardante il primo punto, e la GUI.

Per l'implementazione dei Task è stata realizzata un implementazione di una Thread Pool, permettendo di incapsulare il funzionamento e la gestione dei vari thread all'interno di un oggetto specifico. Il pool si compone quindi di una coda di Task, ovvero tutti i task di una delle tipologie descritte sopra, da mandare in esecuzione. L'esecuzione di un task viene delegata ad uno dei thread presenti all'interno di una lista. Fino a quando la coda di task non sarà vuota uno dei thread andrà a rimuovere un task, per poi eseguirlo. Il pool dei task permette di aggiungere un nuovo task alla coda, di pulire tutti i task presenti, permettendo ad esempio per cambiare la directory da esplorare, di interrompere l'esecuzione da parte dei vari thread ed espone inoltre un metodo onFinish che permette di eseguire una callback quando i vari thread hanno terminato l'esecuzione.

Capitolo 3

Implementazione

Uniche note di implementazione sul pattern MVC riguardano:

- Nel caso di view testuale le stampe in console vengono delegate ad un thread a parte per non appesantire il lavoro dei thread che eseguono il Task 3, inoltre, siccome la scrittura in console risulta lenta le stampe vengono effettuate solamente una volta ogni "UPDATES_BEFORE_PRINT" aggiornamenti.

- Nel caso della GUI, implementata utilizzando Swing, la stampa a video dei valori viene delegata all'EDT, in quanto l'accesso di più thread ai componenti della GUI potrebbe causare situazioni di deadlock.

Per quanto riguarda l'implementazione delle Pool di Thread, la lista dei task è stata implementata utilizzando una BlockingQueue, mentre i task sono dei Runnable. Questa implementazione permette ai vari thread di ricevere il task da mandare in esecuzione in mutua esclusione, evitando corse critiche.

Per quanto riguarda invece la gestione dei dati gli N file di lunghezza maggiore sono stati gestiti tramite un monitor.
Gli intervalli sono stati gestiti diversamente per avere performance migliori. Si è utilizzata una Map che ha come chiave la coppia lower e upper bound dell'intervallo, mentre il valore è di tipo Counter, un monitor, in questo modo l'accesso non risulta in mutua esclusione sull'intero insieme degli intervalli ma solamente sul singolo intervallo.

Capitolo 4

Commenti finali

4.1 Prove di performance

Le prove di performance sono state effettuate su un PC portatile con intel core i7-1065G7 4 core 8 thread, 16GB di ram DDR4 ed una ssd m.2 ed inoltre su una macchina virtuale lubuntu con 2 core 2 thread e 2GB di ram. La directory analizzata contiene 148000 file Java compresi tra 0 e 999 righe di codice, per una dimensione totale di 1.1GB circa.

Utilizzando il PC portatile:

- Nel caso di lettura sequenziale la media è stata di circa 8115ms.
- Utilizzando l'implementazione multithreaded il tempo si è abbassato ad una media di 1834ms sfruttando 8 thread "explorers", 14 "readers", 2 "data managers" e solamente 1 per visualizzare i dati a schermo.

Per uno speedup risultante di 4.42, ovviamente trattandosi di operazioni IO intensive non è pensabile poter raggiungere livelli di speedup pari al numero di thread utilizzati, in quanto viene raggiunto un limite per il quale la velocità del bus di IO e il costo del context switching non permettono di migliorare ulteriormente le performance.

Utilizzando invece la macchina virtuale le performance raggiunte sono:

- Nel caso di lettura sequenziale la media è stata di 24904ms
- Utilizzando l'implementazione multithreaded il tempo si è abbassato ad una media di 18505ms sfruttando 2 thread "explorers", 4 "readers", 1 "data managers" e solamente 1 per visualizzare i dati a schermo.

Lo speedup risultante è di 1.35 che risulta comunque molto lento, probabilmente il problema si trova però nell'accesso lento in memoria, in quanto anche nel caso sequenziale risulta molto più lenta l'esecuzione.

4.2 Correttezza e verifica

Per la verifica della correttezza del sistema realizzato sono state implementate 2 versioni semplificate del software.

La prima sfruttando PlusCal descrive i processi attivi contemporaneamente nel sistema, e come questi interagiscano tra loro, astraendo dalla lettura da disco dei dati e le strutture dati utilizzate per memorizzare i dati statistici che vengono mostrati a video, e dimostrando che le interazioni tra i processi non causano situazioni di starvation, deadlock e livelock.

La seconda semplificazione del software è stata scritta in Java, sfruttando JPF, e mostra la correttezza nell'implementazione delle pool di thread, ancora una volta astraendo dalla memorizzazione dei dati, che sfruttano monitor, e che sono invece stati testati tramite unit testing.