PCD-Assignment1

Davide Bucchieri 0001075283

Presentazione quesito

L'assignment1 richiedeva la realizzazione di un programma concorrente che dato un certo path D:

1. Provvedesse a individuare tutti i files .java accessibili da esso;
2. Trovasse gli N sorgenti con il maggior numero di linee;
3. Classificasse in NI intervalli i files in base al numero di righe che li compongono. In particolare, i primi NI-1 intervalli presentano una dimensione uniforme, mentre nel NI-esimo ricadono tutti i file più lunghi di MAXL righe.

Sia D, che N, NI e MAXL sono indicati dall'utente attraverso una GUI che dovrà anche mostrare interattivamente l'output aggiornato.

Soluzione proposta

Per la risoluzione del problema si è pensato ad un'organizzazione gerarchica che prevede un thread master e x threads slaves. A questi si aggiungono due observes che si occuperanno di aggiornare i due frame interattivi della GUI (uno per mostrare gli N sorgenti più lunghi e uno per gli intervalli).

Per individuare tutti i file si userà una lista di stringhe (fileToReadList) che sarà via via riempita da tutti i file .java e le directory (valide e non vuote) che popolano la directory D.

I file .java, una volta che siano stati individuati e che sia stato contato il loro numero di righe, saranno codificati nel seguente modo:

prefix n nome.java

Dove:

- Prefix è una stringa di dieci caratteri in cui viene salvato il numero di righe (che essendo registrato in un int non può avere più di dieci cifre) preceduto da tanti 0 quanti ne servono per raggiungere la lunghezza sopracitata.
- N è un numero da 0 a 9 (un char in realtà) che indica la prima posizione di prefix in cui non ci siano 0. In questo modo al momento della stampa si può accedere direttamente al numero di righe evitando gli zeri iniziali.
- Per concludere si avrà il nome del file. Non si usa il path per non avere stringhe troppo lunghe (anche se così facendo potrebbero apparire file codificati nello stesso modo).

Così facendo per trovare gli N file più lunghi non serviranno mappe o strutture complesse, ma semplicemente una seconda lista di stringhe, che chiameremo rankingList, che sarà ordinata seguendo l'ordine alfabetico e sfruttando proprio prefix.

Per gli intervalli, infine, si è scelto di usare un array di contatori seguendo la filosofia di usare strutture dati il più leggere e semplici possibili.

Non sono stati applicati vincoli nel valore di N e MAXL, mentre NI è stato limitato ai soli divisori di MAXL + 1. Ad esempio, se MAXL è pari a 10, gli unici valori di NI ammissibili sono 2, 3, 6, 11 (1+1, 2+1, 5+1, 10+1). In

questo modo si potranno avere NI – 1 intervalli di pari dimensione per i files con numero di righe inferiore a MAXL e un NI-esimo intervallo per tutti gli altri. Nel caso in cui NI sia pari a MAXL + 1 si avrà una visualizzazione ad hoc nella GUI e si potranno individuare il numero di files con lo stesso numero di righe (i files con 0 righe, con 1, 2 etc fino ad arrivare a quelli con più di MAXL righe).

Monitors

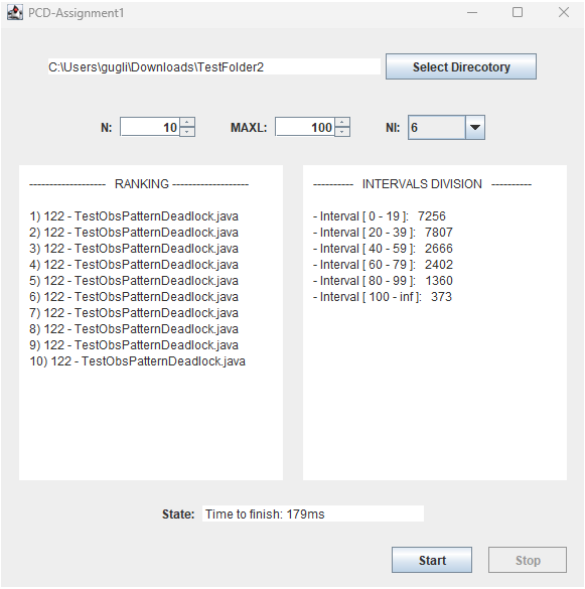
Avendo più threads che lavorano contemporaneamente sono necessari una serie di monitors per garantire il corretto uso delle risorse comuni (rankingList, l'array di contatori e fileToReadList) e la sincronizzazione. Per massimizzare le performance si è deciso di avere un monitor per risorsa critica, così da poter lavorare in parallelo su più di una. In questo modo si ha come contro che la struttura del programma si fa più complessa, ma in compenso le singole classi si fanno più piccole e semplici.

In particolare, durante l'esecuzione del programma si avranno i seguenti monitors:

- RankingMonitor (Interfaccia): Come dice il nome serve a gestire l'accesso alla rankingList. Per questioni di ottimizzazione si avrà un RankingListMonitor nel caso di N > 1 o un RankingStringMonitor se N = 1. In entrambi i casi, grazie alla sopracitata interfaccia, si avranno gli stessi metodi, mentre cambierà solo il tipo di risorsa gestita (un arraylist nel primo caso e una stringa nel secondo). La logica di aggiornamento della risorsa è incapsulata dentro il metodo put. Risulta banale nel caso di una semplice stringa, mentre è da evidenziarne il funzionamento nel caso dell'arrayList: in un primo momento la lista sarà riempita casualmente, poi al momento del raggiungimento del size() N si farà un primo ordinamento con Collection.sort(). Da quel momento in poi le nuove stringhe saranno inserite solo se il files ad esse associate presenteranno un numero di righe superiori a quelle già presenti e saranno posizionate direttamente nell'indice corretto (andando via via ad eliminare la stringa in testa). Nota: l'ordinamento in ordine alfabetico è inverso rispetto a quello per numero di righe decrescente, quindi bisogna lavorare al contrario. Inoltre fino al raggiungimento del size() N, la lista risulterà disordinata e sarà necessario che l'observer riordini l'output (non la rankingList) prima di farlo stampare a schermo.
- CounterMonitor: gestisce un contatore che parte da 0 e può essere solo incrementato o letto. Non si avrà un monitor per tutto l'array di contatori degli intervalli, ma un array di CounterMonitor (countersMonitor) di dimensioni N cosi da permettere ai thread di aggiornare in contemporanea più contatori (senza doversi dividere un unico lock).
- PathToReadListMonitor: ha una duplice funzione occupandosi sia della gestione della risorsa pathToReadList sia del coordinamento Master-Slaves. Il Master, infatti, da inizio al processo attraverso l'inserimento del path D nella suddetta lista, mentre attraverso il contatore nActivedSlaves è possibile sapere se l'elaborazione della richiesta è finita o no. Inoltre, essendo pathToReadList una lista di stringhe, il master può inserire stringhe particolari per disattivare temporaneamente o addirittura spegnere gli slaves.
- StateMonitor: mentre PathtoReadListMonitor è usato per la coordinazione dei threads slaves, per gestire i due observer e il master si usano degli stateEnum che possono assumere vari valori: start, stop, continue, wait e off. In particolare, riveste un ruolo fondamentale il wait, che permettere di mettere in wait (appunto) il thread associato in attesa di un nuovo cambio di stato. A differenza degli altri valori, wait non è fissato da agenti esterni, ma viene assunto dalla variabile stateEnum alla fine di ogni read.
- DataMonitor: per finire, sia i parametri, che i monitors sono salvati in una superclasse passiva che racchiude anche i metodi per la definizione delle stringhe da inviare all'Event Dispatch Thread e l'aggiornamento di rankingList e counters. Il suo ruolo è evitare di avere gli stessi campi sparsi in più classi e permettere l'inizializzazione degli stessi. Fanno eccezione i monitors che hanno una

funzione sincronizzate, ovvero i tre StateMonitor (uno per il rankingListObserver, uno per il countersObserver e uno per il master) e FileToReadList, che essendo salvati in un campo final non verranno più modificati e per questo possono essere passati anche i threads(cosi da migliorare le performance). I parametri D, N, NI e MAXL e i monitor rankingListMonitor e counters (che dipendono dai primi), invece, saranno inizializzati ogni volta che l'utente premerà start attraverso un metodo synchronized presente nella superclasse stessa. Il synchronized è necessario per impedire ai due observers di realizzare nuove stringhe per la GUI basandosi su strutture dati che stanno venendo inizializzate. Questo è l'unico caso in cui si usa il paradigma readers-writer. A differenza del master e gli slaves, gli observers, infatti, saranno legati all'Event Dispatch Thread e di conseguenza non è garantito che al momento in cui l'utente clicchi nuovamente il pulsante start, dopo una precedente run, essi abbiano già finito di lavorare.

GUI



La GUI è stata pensata per essere prevalentemente funzionale più che bella. Presenta i due frame richiesti e in più un terzo frame per visionare lo stato del processo. Ci sono poi i campi di input per inserire i parametri e i pulsanti di start e stop. Il valore di NI è scelto attraverso un JCombo i cui items dipendono dal valore di MAXL.

Premendo il tasto start, i campi di input e il tasto stesso saranno disabilitati, sarà avviata l'inizializzazione dei parametri e delle strutture dati e sarà dato l'ok al thread master per iniziare. Il tasto stop, però, non sarà attivato in automatico, ma sarà lo stesso thread master a farlo abilitare ad inizializzazione ultimata. Così, si impedisce all'utente di avviare più volte il processo. Stessa cosa avviene con il pulsante stop: i campi di input e il tasto start non saranno abilitati subito, ma solo quando i thread si saranno effettivamente fermati. Questo permette di evitare di inserire altre lock e di poter lavorare comunque in sicurezza con i parametri inseriti dall'utente e le risorse condivise che dipendono da essi.

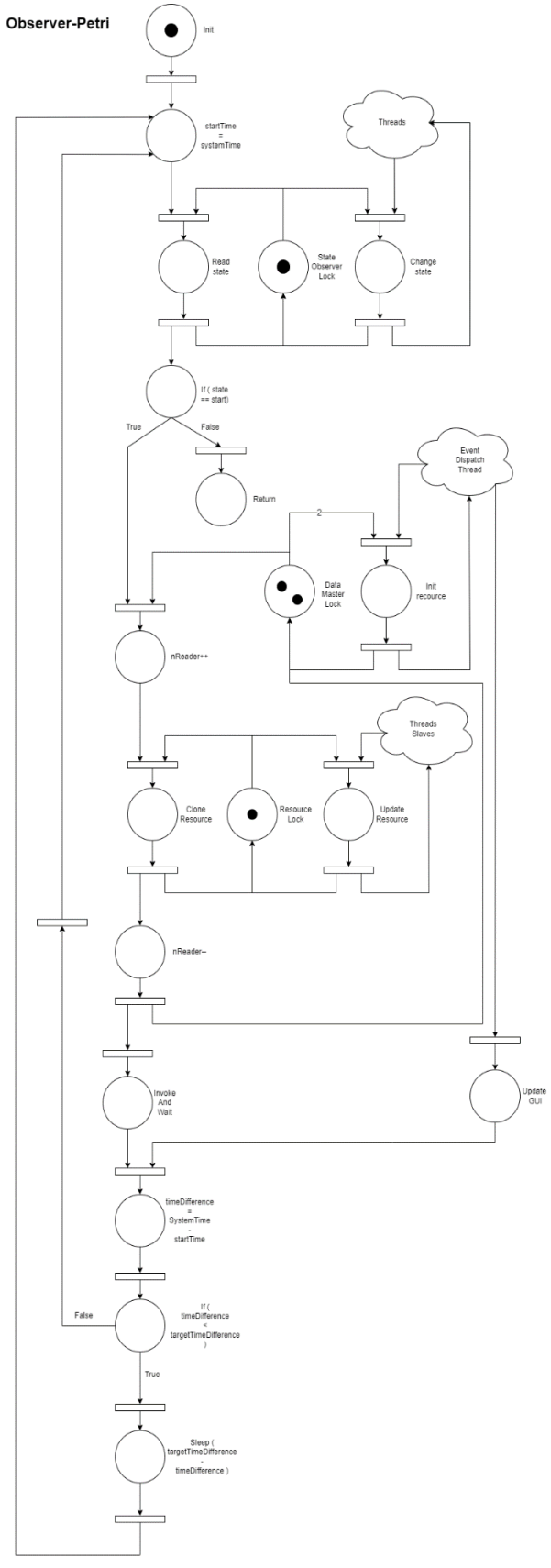
I due frame ranking e intervals division saranno aggiornati dall'event dispatch thread su comando degli observers, mentre State dipenderà ancora una volta dal thread master e dalla GUI stessa che lo modificheranno via via per informare l'utente sull'andamento dell'elaborazione.

Per finire quando sarà chiusa la finestra verrà automaticamente cambiato lo stato del master thread in off così che sarà poi avviato il codice per la terminazione pulita dei vari threads.

Threads

Come già detto ci sono fondamentalmente tre tipi di threads: gli observers, il master e gli slaves. Pur avendo compiti diversi seguono tutti uno stesso schema: richiamano il metodo start in automatico in fase di inizializzazione e si inseriscono in un loop infinito che sarà rotto solo in fase di chiusura del programma. In questo modo si userà sempre lo stesso pull di threads, non si perderà tempo per generare di nuovi e si avranno complessivamente performance migliori. Per permettere tutto questo risultano essenziali i meccanismi che permettono ai singoli threads di rimanere in attesa di nuove task da svolgere.

Di seguito saranno riportati le reti di petri associate alle singole tipologie di threads. Sono state realizzate con un certo livello di astrazione, così da poter rendere più chiara la struttura della soluzione adottata (di per se complessa).



Observers

Gli observers hanno il compito fondamentale di aggiornare la GUI durante il processamento dei file della directory. È fondamentale che interferiscano il meno possibile con il lavoro di slaves e master.

Dopo la prima inizializzazione viene salvato il tempo attuale e ci si mette in attesa di un cambiamento di stato. In pratica si aspetta che o la ranking list o i contatori degli intervalli siano cambiati e che tale cambiamento sia segnalato con l'assegnazione del valore start al rispettivo stateEnum.

Se, invece, si è chiusa la GUI il master setterà tale variabile in off, provocando il return.

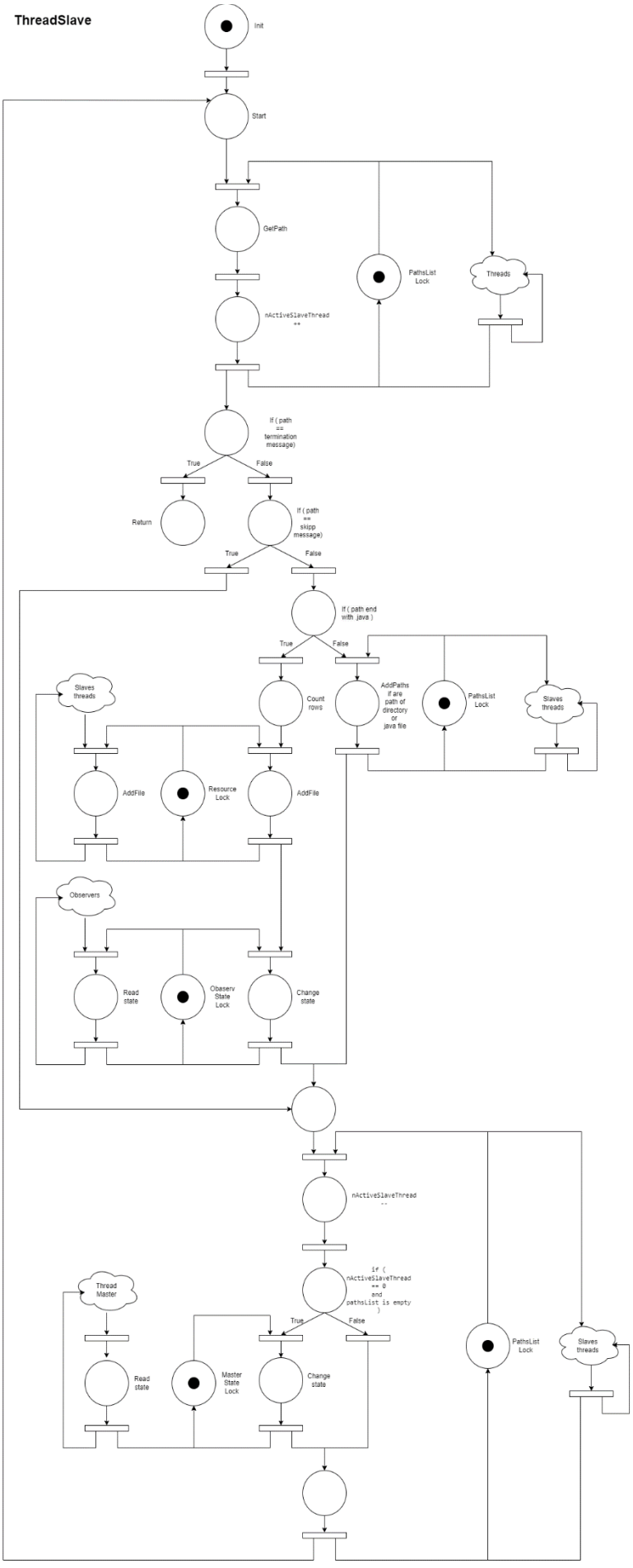
Quindi, dopo aver verificato che lo stato è uguale a Start, si richiede il lock per acceder alla risorsa (che ancora una volta può essere o un contatore o la rankingList) e la si clona, così da poterla leggere piano piano senza che nel frattempo sia modificata (è richiesta l'atomicità). Prima di fare questo, però, è essenziale verificare che le risorse non stiano venendo inizializzate dell'Event Dispatch Thread. Si può vedere come sia stato adottato il pattern readers-writer (i due readers non solo altro che i due observers).

Il contatore nReaders dovrà, quindi, essere incrementato all'inizio e decrementato alla fine (con una successiva notify per risvegliare un eventuale E.D.T in wait).

A questo punto, dopo aver costruito la stringa da inserire nel frame della GUI, viene fatta una invokeAndWait sempre all' E.D.T e si attende cha abbia finito. Infatti, non ha senso fare ulteriori richieste di aggiornamento della GUI se prima non sia stata ultimato quello precedente.

In ogni caso, se alla fine di tutto questo è passato meno tempo di un certo intervallo di tempo (30 fps), l'observer va in sleep così da accedere meno volte possibile alle risorse condivise senza che l'utente se ne accorga.

Visto che si guarda il valore dello stato e non si tiene conto dei singoli cambiamenti, anche se le risorse verranno modificate più volte in un intervallo di tempo la GUI sarà aggiornata solo una volta (con i valori finali).



Slaves

Gli slaves si occupano del lavoro bruto. Per prima cosa provano a prendere una stringa dalla fileToReadList. Se questa è vuota attendono in wait che o un altro slave o il master inserisca un nuovo item.

Una volta presa la stringa, incrementano il contatore dei thread attivi e iniziano il processo vero e proprio.

Si controlla, quindi, se la stringa è relativa ad un path o se è uguale al messaggio di terminazione o skip così da agire di conseguenza.

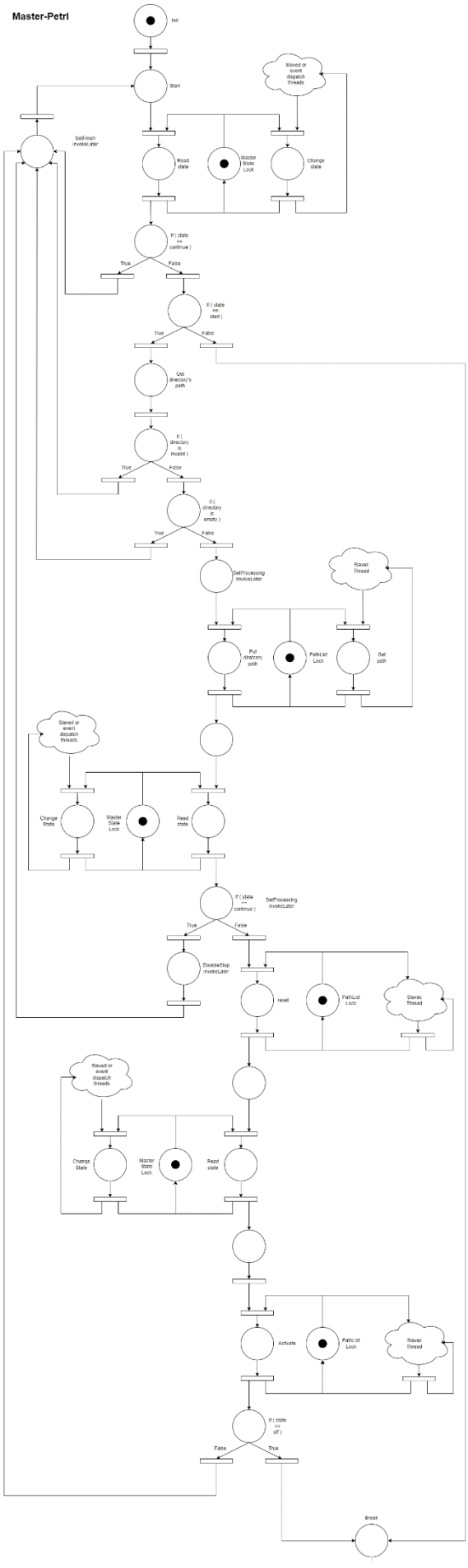
In particolare, la stringa di skip serve solo a saltare l'elaborazione della stringa (si vedrà meglio dopo il perché), mentre quella di terminazione porta al return.

Nel caso la stringa contenga il path di un file si controlla inanzitutto se si tratta di una directory o di un .java. Nel primo caso ci si limita ad inserire in fileToReadList tutti i path associati a .java o a directory valide e non vuote (con valide si intende che File.listFiles non ritoni null), mentre nel secondo caso prima si aggiornano le risorse e poi si invia la segnalazione agli observers.

Nello schema risulta semplificato, ma nella pratica prima si accede alla rankingList (con un lock) e poi al counter (con un secondo lock) relativo all'intervallo appropriato. Inoltre, mentre il countersObserver è sempre risvegliato, il rankingListObserver è attivato solo se il file è stato realmente aggiunto alla lista (cosa non certa, come si è visto, dopo che questa ha raggiunto il size() N).

A questo punto il lavoro risulta ultimato e dopo aver preso la lock, si decrementa il numero di slaves attivi. Nel caso in cui si verifica contemporaneamente che il contatore è uguale a 0 e la fileToReadList è vuota, il processo può dirsi ultimato.

Alla fine si ricomincia da capo.



Master

Il master si occupa della gestione di tutto il processo. Nella fase di init inizializzerà la stessu GUI, il dataMonitor e tutti gli altri thread. In questo modo nel main basterà inizializzare un nuovo thread master.

Come i precedenti tipi di threads per prima cosa si metterà in wait, in attesa, questa volta, che l'utente faccia qualcosa.

Quindi controlla se lo stato letto è pari a continue, in caso affermativo ricomincia da capo. Se, invece, è uguale a start inizia l'elaborazione. In tutti gli altri casi esce dal while.

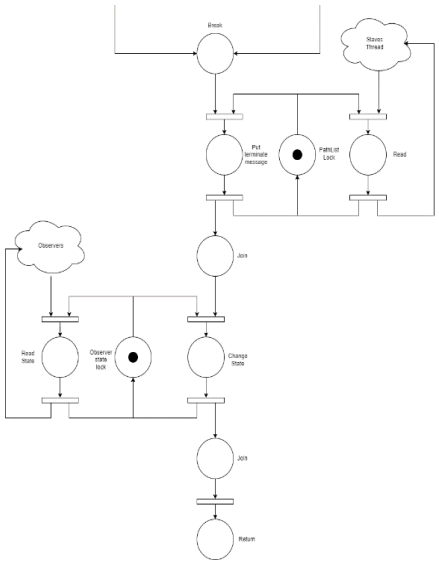
Ad elaborazione iniziata, prende d, e verifica se si tratta di una direcotory non valida o vuota. In caso negativo, segnala l'inizio del processo alla GUI, facendo aggiornare asincronicamente la casella dello stato e abilitando il bottone stop (sempre con un invokeLater). Quindi inserisce il path nella fileToReadList e si mette in attesa di un nuovo cambiamento del MainState.

A questo punto possono avvenire due cose: o l'utente clicca il bottone stop o il processamento dei file termina.

In quest'ultimo caso, lo stato sarà uguale a "continue", lo stop sarà disabilitato e si segnalerà la terminazione della procedura. Non rappresentando un'azione atomica potrebbe capitare che al momento della terminazione prima che lo stato sia stato settato in continue, l'utente possa premere il pulsante di stop portando il master a comportarsi di conseguenza. Per evitare problemi è stato posto quell'if iniziale che nel caso in cui ad una successiva lettura dovrebbe essere letto un valore della stateEnum pari a continue si leggerà una seconda volta (ignorandolo di fatto). In questo modo anche se la terminazione del processo dovesse essere segnalata dopo lo stop il master non ne risentirebbe.

Nel caso, invece, in cui lo stop viene premuto prima della terminazione del processo o che si chiuda la finestra della GUI, per prima cosa si richiama il metodo resed di pathToReadListMonitor. In questo modo non solo si ripulisce la lista, ma si fanno si anche altre due cose:

- Sarà inserita una stringa di skip, cosi che almeno un thread resti attivo.



- Sarà disattivata la possibilità di aggiungere nuovi path alla lista, cosi da impedire il proseguimento dell'elaborazione nel caso in cui alcuni slaves stiano lavorando su duelle directory

A questo punto il master attende che tutti gli slaves abbiano finito (visto che la lista è vuota e non possono essere aggiunte nuove stringhe, sarà questione di istanti), riattiva la possibilita di aggiornare fileToReadList e nel caso abbia prima ricevuto uno stop ricomincia da capo, se no rompe il while loop.

Dopo il break per prima cosa si inseriscono tanti messaggi di terminazione quanti sono gli slaves, quindi si fa la join ad ognuno di essi (con due for diversi per evitare la starvation). Quindi si cambiano gli state degli observers ad off e si attende anche la loro terminazione.

Test performance

Per il test delle performance si è preso come punto di riferimento quello che nel progetto prende il nome di mainSequenziale, che ricorsivamente e senza concorrenza svolge tutto quello che è richiesto nel punto 1 dell'assignment. Impostando un certo path d con migliaia di file java, N = 10, MAXL = 100 e NI = 6 si avranno stampati a schermo 10 file con 122 righe, e una suddivisione in intervalli pari a:

- 7256 files con un numero di righe compreso fra 0 e 19;
- 7807 files con un numero di righe compreso fra 20 e 39;
- 2666 files con un numero di righe compreso fra 40 e 59;
- 2402 files con un numero di righe compreso fra 60 e 79;
- 1360 files con un numero di righe compreso fra 80 e 99;
- 373 files con almeno 100 righe.

Partendo da questi dati si sono fatti una serie di test con gli stessi parametri e un numero di threads diverso:

n Thread	Tempo 1	Tempo 2	Tempo 3	Correttezza
1	1065	1078	1090	ok
2	624	632	61	ok
4	339	408	365	ok
8	73	261	259	ok
12	177	197	203	ok
16	143	179	176	ok
32	138	166	164	ok
64	154	172	167	ok
128	206	189	197	ok

Con numero di threads si intende il numero di slaves, visto che master e observers restano in attesa per buona parte del tempo di esecuzione e di conseguenza hanno un impatto relativamente piccolo. I tempi sono espressi in millisecondi e ne sono riportati tre per caso di studio. L'ok indica che in tutte le run si è ottenuto lo stesso output che si è avuto nel caso sequenziale.

Come si nota il programma risulta robusto e scalabile. Arrivati a 32 threads (che corrispondono al doppio dei core logici del pc usato per le simulazioni) si ha il massimo delle performance, mentre con un numero di core superiore si fa sentire il collo di bottiglia rappresentato dalla lettura delle memorie. Inoltre, si può notare che il caso con un solo thread non si discosta molto come tempi da quello sequenziale (anche se questo usa meccanismi leggermente diversi per il processamento), il che fa capire come tutta la struttura creata intorno per la gestione delle risorse e la sincronizzazione (e la GUI stessa) non impatta più di tanto.