Relazione sull'implementazione del simulatore biologico WATOR

Alessandro Pagiaro

26 giugno 2015

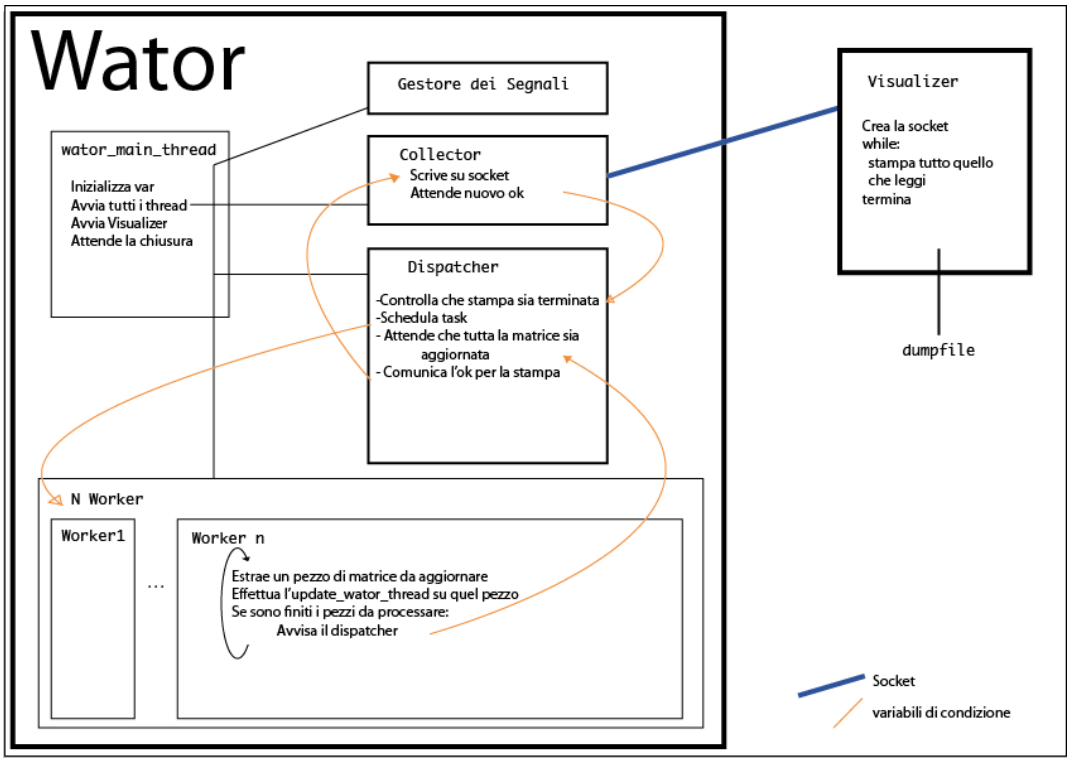
Indice

1 Il processo principale: Wator

Il processo Wator rappresenta il nucleo operativo di tutto il programma. Il processo è suddiviso in vari thread:

- Main Thread
- Dispatcher
- Collector
- Signal Handler
- Worker_1, .., Worker_n

1.1 Schema Grafico



1.2 Main Thread (wator_main_thread.c)

Il `wator_main_thread.c` è il thread che si preoccupa di avvia il processo completo. All'avvio esegue in ordine le seguenti operazioni:

- Inizializza tutte le variabili interi, le mutex e le condition

- Verifica che esiste la directory per la creazione della socket e che non contenga una precedente socket creando poi problemi di creazione della nuova
- Maschera tutti i segnali rilevanti per il processo (verrà approfondito nella sezione Signal Handler)
- Legge le opzioni passate all'avvio e ne verifica la validità
- Prova ad avviare i thread necessari all'esecuzione del processo:
  - Prova ad avviare il `Signal Handler`
  - Prova ad avviare n thread `Worker`
  - Prova ad avviare il thread `Collector`
  - Prova ad avviare il thread `Dispatcher`
  - Prova ad avviare il processo `Visualizer`
- Si mette in attesa sui thread avviati
- Al termine dei vari thread, distrugge le mutex e le variabili di condizione
- Termina

All'avvio dei vari thread viene verifica l'avvenuta creazione. In caso in cui questa fallisca si è deciso di ritentare per un numero di volte pari a `MAX_TRY`, attendendo un numero esponenziale di secondi ad ogni nuovo tentativo.

1.3 Thread Dispatcher (dispatcher.c)

Il thread `Dispatcher` è colui che si preoccupa di schedulare i vari thread e di azzerare le variabili di supporto alla simulazione ad ogni chronon.
Effettua in ordine le seguenti operazioni:

- Alloca la matrice `skip`, usata dalla funzione *update*.
- Inizia la fase di aggiornamento:
  - Azzera la matrice `skip`
  - Si assicura che il `Collector` abbia terminato la stampa prima di procedere
  - Mette a disposizione dei thread `Worker` le Sottomatrici da aggiornare
  - Attende che l'aggiornamento sia completato
  - Se sono passati `intervallo_di_stampa` chronon, comunica al `Collector` di inviare la stampa

Nel caso in cui Il `Collector` stia stampando, e quindi il `Dispatcher` non può procedere alla stampa altrimenti c'è il rischio di aggiornare la matrice durante la stampa, il `Dispatcher` si mette in attesa sulla variabile di condizione `stampa_effettuata`. Per la sincronizzazione viene quindi usata la mutex `stampa_mux`.
Per la sincronizzazione tra gli n thread `Worker`, che estraggono dalla lista delle Sottomatrici, e il thread `Dispatcher`, che invece inserisce elementi nella lista delle Sottomatrici, viene usata la mutex `lista_mux`.

Il thread `Dispatcher` prima di effettuare la stampa si assicura che:

- La lista delle Sottomatrici sia vuota, cioè che i `Worker` abbiano estratto tutti gli elementi
- Il `count_worker` sia uguale a 0, cioè che non ci sono `Worker` che stanno effettuando l'*update* della sottomatrice.

La variabile `count_worker` viene incrementata ogni qual volta la funzione *extract* chiamata da un `Worker` riesce ad estrarre un elemento dalla lista delle Sottomatrici (`lista_pezzi`) e viene decrementata dal `Worker` quando la funzione *update* termina.

1.4 Thread Collector (collector.c)

Il thread `Collector` è colui che si preoccupa di mandare le informazioni da stampare al processo `Visualizer`.
All'avvio tenta di connettersi alla socket `SOCKNAME` che viene creata dal `Visualizer`. Qualora la socket non esista ancora, e quindi la funzione *connect* restituisce l'errore ENOENT, si addormenta per 250 millisecondi e ritenta.
Connesso quindi alla socket effettua ciclicamente (finché non viene settata la variabile di chiusura dal thread `Dispatcher`) le seguenti operazioni:

- Controlla se può stampare o deve attendere il Dispatcher
- Scrive sulla socket la lettera `A`, token che per convenzione determina l'inizio di una nuova matrice, permettendo al `Visualizer` quando termina la stampa di una matrice e ne inizia una nuova
- Scrive sulla socket la matrice completa
- Setta la variabile `puoi_procedere` a **true** e risveglia eventualmente il thread `Dispatcher` in attesa sulla variabile di condizione `stampa_effettuata`.
- Si sospende sulla variabile di condizione `stampa_pronta` finché non ritorna **true** la variabile `puoi_stampare`.

Al comando di chiusura, quando cioè la variabile `close_all` viene settata a **true** dal `Gestiore Segnali`, viene chiusa la socket facendo terminare anche il processo `Visualizer` (come approfondito nella sezione `Visualizer`).

1.5 Thread Worker (worker.c)

Il thread `Worker` è il thread che effettivamente aggiorna la matrice chiamando al suo interno la funzione *update*. Questo effettua una corsa critica con gli altri $n-1$ thread `Worker` per estrarre dalla lista delle Sottomatrici `lista_pezzi` la sottomatrice da aggiornare.
Il thread `Worker` effettua ciclicamente le seguenti operazioni:

- Estrae la sottomatrice tramite la funzione *extract* (che sospende il thread in caso in cui la lista `lista_pezzi` sia vuota)
- Preleva dall'elemento estratto solo le informazioni riguardante le informazioni della sottomatrice, liberando la memoria in eccesso
- Effettua *update* sulla sottomatrice
- Verifica che non ci siano altri pezzi da elaborare e che non ci siano altri `Worker` a lavoro. Se entrambe le condizioni sono vere:
  - Controlla che non è richiesta la terminazione della simulazione
  - Sveglia, se necessario, il thread `Dispatcher` inviando un segnale alla variabile di condizione `stanno_lavorando`

I controlli finali sono effettuati accedendo in mutua esclusione sia con gli altri thread `Worker` sia con il thread `Dispatcher`.

1.6 Gestore Segnali (gestione_segnali.c)

Il thread `Gestione Segnali` è colui che si preoccupa di ricevere i segnali inviati al processo e gestirli opportunamente, settando le giuste variabili globali interpretate poi dai vari thread.
Tutti i thread, infatti, hanno i segnali relativi al processo mascherati.

1.6.1 Il segnali di `SIGINT` o di `SIGTERM`

Questi due segnali richiedono l'esecuzione gentile del processo. Registrati dal thread `Gestore Segnali`, viene settata a **true** la variabile `close_all` che farà uscire dal ciclo computazionale i thread `Dispatcher` e `Collector`.

Per effettuare la chiusura gentile, cioè per terminare l'aggiornamento della matrice e stampare l'ultimo stato, il thread `Dispatcher` controlla la variabile solo al termine di un'aggiornamento. Uscito quindi dal ciclo di aggiornamento setta a **true** la variabile `termine_elaborazione` che comunica ai thread `Worker` che il `Dispatcher` è terminato e quindi devono a loro volta terminare. I thread `Worker` non possono usare direttamente direttamente la variabile `close_all` poichè non è detto che tutta la lista delle Sottomatrici sia stata aggiornata e controllano quindi la variabile `termine_elaborazione` per terminare la loro esecuzione.

Per assicurarsi che tutti i thread `Worker` possano valutare la condizione di terminazione, viene inviata una *broadcast* per portare nella lista pronti tutti quelli in attesa. Situazione analoga viene realizzata con il thread `Collector` effettuando una *signal* sulla variabile `stampa_pronta`

Al termine dell'esecuzione il thread `Dispatcher` invia un ultimo segnale di stampa al `Collector` il quale effettuerà un'ultimo ciclo di stampa e terminerà poichè la variabile `termine_esecuzione` sarà stata messa a **true**.

1.6.2 Il segnale `SIGUSR1`

Questo segnale comunica al processo di dover stampare il wator.check. Questa funzione, a differenza della classica stampa della matrice sul `dumpfile`, viene effettuata dal thread `Dispatcher`. Questo poichè il thread `Collector` viene risvegliato dall'attesa solo ogni `intervallo_di_stampa` chronon, non potendo quindi assicurare una stampa del `wator.check` nel chronon in cui viene richiesta.

2 Il processo di stampa: Visualizer

Questo processo non fa altro che creare la socket `SOCKNAME`, valutare il parametro che gli è stato passato `dumpfile` e quindi leggere finchè non viene chiusa la socket dallo scrittore `Collector` stampando quel che legge sul `dumpfile`.
Qualora `dumpfile` è lo `stdout` la stampa avviene come stream, altrimenti il file `dumpfile` contiene solamente l'ultimo aggiornamento della matrice pianeta.

3 Funzioni Rilevanti

3.1 split_matrice

Questa funzione, dichiarata nel file `wator_thread_funzioni`, si occupa di dividere la matrice del pianeta in varie sottomatrici che saranno poi aggiunte (sottoforma di puntatori) alla lista `lista_pezzi` dal `Dispatcher` ed estratte dagli n `Worker`. Questa funzione divide la matrice in tanti quadranti di dimensione `K` e `N`, settati da opportuna macro. Qualora la matrice ha un avanzo crea dei rettangoli di dimensioni degli elementi rimanenti.

3.2 update_wator_thread

È la funzione di aggiornamento della sottomatrice di competenza. Viene invocata dal thread `Worker`.
La funzione esegue un'aggiornamento lineare sulla sottomatrice accedendo alle aree critiche (le prime/ultime due righe/colonne) tramite mutex unica. Le celle critiche sono chiaramente quelle che confinano (che definiamo critiche$_1$) con altre sottomatrici per ovvie ragione ma anche quelle più interne, cioè quelle

sulla seconda riga, sulla seconda colonna, sulla penultima riga e penultima colonna (critiche$_2$), poichè un eventuale animale potrebbe spostarsi capitando su su una cella critica$_1$ che in quel momento è acceduta anche dal thread `Worker` adiacente.

3.3 add

La funzione *add* è quella che gestisce l'inserimento delle sottomatrici in una lista. Accede alla lista in mutua esclusione. Dopo aver inserito un elemento effettua una *signal* sulla variabile di condizione `lista_vuota` in modo da mettere nello stato di pronto un eventuale thread `Worker` in attesa.

3.4 extract

La funzione *extract* è quella che gestisce l'estrazione degli elementi dalla lista delle sottomatrici. È invocata dai thread `Worker`. Nel caso in cui la lista sia vuota mette il thread in attesa sulla variabile di condizione `lista_vuota`.

4 Lo script bash Watorscript

Lo script `Watorscript` è uno script di supporto al simulatore. Verifica la correttezza sintattica del pianeta, ne conta il numero di squali e di pesci.
Il *parsing* delle opzioni avviene mediante l'utility `getopt`.

5 Installazione

Per l'installazione si può ricorrere all'utility `Make`. Basterà invocare il comando `make install` o alternativamente `make wator; make visualizer`.

6 Mappa dei file del progetto

7 File Sorgenti

- wator_thread_funzioni.h
  - wator_thread_funzioni.c
  - dispatcher.c
  - collector.c
  - worker.c
  - gestore_segnali.c
  - wator_main_thread.c
- valutazione_argomenti.h

  - valutazione_argomenti.c
- wator.h
  - wator.c
- Visualizer.c