Università di Pisa, CdL in Informatica, A.A. 2014-15

Relazione del Progetto di Laboratorio di Sistemi Operativi

Wator: un simulatore distribuito di un modello biologico

Giorgio Vinciguerra

15 giugno 2015

Indice

1. Introduzione

Nel progetto di Laboratorio di Sistemi Operativi è stato chiesto di realizzare un simulatore di vita in un pianeta acquatico popolato da squali e pesci. Il pianeta è rappresentato da una matrice $n \times m$ le cui estremità sono tra loro collegate così da formare una superficie a forma di anello. Il simulatore prevede che: i suoi abitanti si riproducano dopo un certo intervallo di tempo; che gli squali siano predatori dei pesci; che gli squali muoiano di inedia dopo un periodo impostato dall'utente.

L'evoluzione avviene per unità di tempo immaginarie, denominate "chronon", in cui gli squali cacciano i pesci ed entrambe le forme di vita si muovono. I movimenti avvengono casualmente in una delle quattro celle immediatamente adiacenti (in alto, a destra, in basso, a sinistra) libere. La fase di caccia prevede che uno squalo mangi un pesce vicino e occupi la sua posizione.

Il progetto è composto di tre componenti fondamentali sviluppate nel linguaggio C: una libreria, un processo che si occupa della simulazione e un'altro che si occupa della stampa (su video o su file) dello stato corrente del pianeta. Nelle pagine successive verrano discusse le scelte implementative per ognuno di questi componenti. Nelle ultime pagine verrano presentati anche i componenti minori di supporto e di collaudo del progetto (Par. 5.2).

2. La libreria Wator

La libreria Wator è un insieme di algoritmi che permettono la **creazione**, l'**evoluzione** e la **presentazione** dello stato corrente della simulazione. È implementata nel file `wator.c` ed esposta e documentata nel file d'interfaccia `wator.h`.

Oltre che dalla matrice del pianeta `w`, una simulazione è caratterizzata da due matrici parallele di contatori `btime` e `dtime` delle stesse dimensioni. In particolare `btime[r][c]` mantiene il numero di chronon dall'ultima riproduzione del pesce o dello squalo in `w[r][c]`; `dtime[r][c]` invece è il numero di chronon dall'ultimo pasto di uno squalo. Lo scopo di tali matrici risulterà più chiaro nel paragrafo 2.1.

Nello sviluppo della libreria è stato adottato un approccio di *programmazione difensiva* volto a ridurre al minimo la probabilità che si verifichino situazioni in cui le informazioni di simulazione sono corrotte, oppure che si effettuino operazioni non consentite dal sistema operativo ospitante che potrebbero causare la terminazione anormale del programma. Nello specifico ogni funzione (annotata se possibile da asserzioni) effettua il controllo dei parametri passati, del risultato di ogni richiesta di allocazione di memoria e del risultato di chiamate di funzioni delle librerie del linguaggio.

Per migliorare l'efficienza del programma, alcune funzioni come `cell_to_char` e `shark_rule1` ecc. sono state marcate con la keyword C `inline` per *suggerire* al compilatore di sostituire ogni invocazione a funzione con il suo codice, così da evitare il costo di chiamata-ritorno.

2.1. Funzioni per l'evoluzione del pianeta

L'algoritmo più importante della libreria è quello implementato dalle funzioni `update_wator` e `update_wator_rect`, che calcolano l'evoluzione dell'intero pianeta o di un suo rettangolo. Il risultato di tali funzioni dipende fortemente dai valori *shark death* `sd`, *shark birth* `sb` e *fish*

birth `fb` impostanti dall'utente tramite un file di configurazione `wator.conf`.
Cella per cella, se non c'è acqua, vengono applicate nell'ordine le seguenti regole:

- Se nella cella c'è uno **squalo**:
  1. e nelle quattro celle adiacenti c'è un pesce, lo squalo si sposta nella sua cella, lo mangia e il contatore `dtime` dello squalo viene azzerato, altrimenti lo squalo si sposta in modo casuale;
  2. nelle coordinate di arrivo se il contatore `btime` ha raggiunto il valore `sb`, lo squalo partorisce un figlio nella prima cella adiacente libera disponibile. Se il contatore `dtime` dello squalo è maggiore di `sd` lo squalo muore e la cella viene liberata, altrimenti il contatore `dtime` viene incrementato di uno.
- Se nella cella c'è un **pesce**:
  1. si controllano le quattro celle adiacenti: se sono tutte occupate il pesce rimane fermo, altrimenti ne viene scelta una in modo casuale in cui il pesce si sposterà;
  2. nelle coordinate di arrivo se il contatore `btime` del pesce ha raggiunto il valore `fb`, il pesce partorisce un figlio nella prima cella adiacente libera disponibile, altrimenti il contatore viene incrementato di uno.

Le quattro regole appena descritte corrispondono alle funzioni `shark_rule1`, `shark_rule2`, `fish_rule3` e `fish_rule4`.

2.2. Algoritmo per garantire l'aggiornamento singolo

L'applicazione delle regole avviene sequenzialmente scandendo la matrice dall'alto verso il basso e da sinistra a destra. Poiché i pesci e gli squali possono spostarsi nelle celle a destra e in basso, si rende necessario un algoritmo che eviti che tali movimenti non causino un ulteriore aggiornamento nelle iterazioni successive. Questa situazione è rappresentata dalla Fig. 2.1.

In `update_wator()` la soluzione adottata prevede la presenza di due array di boolean di dimensioni pari al numero di colonne della matrice del pianeta, che tengono traccia delle celle già aggiornate. Il primo array A_{curr} indica la celle da saltare nella riga corrente, mentre A_{next} indica quelle della riga successiva. Se uno squalo o un pesce in posizione (r, c) si muove a destra, ad $A_{curr}[c+1 \mod m]$ viene assegnato il valore *true*, altrimenti, se il movimento è verso il basso, è $A_{next}[c]$ a diventare *true*. All'inizio della riga successiva $A_{curr} \leftarrow A_{next}$ e il nuovo A_{next} viene azzerato.

Per `update_wator_rect()`, che verrà utilizzato intensivamente nel processo multithread Wator, non è stato possibile applicare questo tipo di ottimizzazione: è necessario passare come argomento una matrice di boolean cosicché le future chiamate alla stessa funzione, indipendentemente dal tipo di scansione adottata, possano conoscere quali celle sono state già aggiornate.

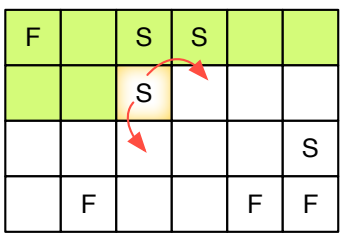


Figura 2.1: In verde sono indicate le celle già aggiornate. Se non si fanno i dovuti accorgimenti e lo squalo compie i movimenti indicati con le frecce rosse, nelle prossime iterazioni lo squalo verrebbe aggiornato di nuovo.

3. Il processo Wator

Il processo Wator si occupa principalmente dell'avvio/terminazione di una simulazione e della creazione di un processo Visualizer a cui comunicherà periodicamente lo stato corrente della simulazione. Lo stato iniziale del pianeta viene letto da un file fornito come argomento dell'eseguibile `wator`, mentre la configurazione del pianeta viene sempre letta da un file `wator.conf` posizionato nella stessa cartella dell'eseguibile. La struttura logica adottata per il processo è schematizzata nella Fig. 3.1.

La parte di simulazione avviene utilizzando un thread *dispatcher* che si occupa del riempimento di una coda di task dalla quale i k thread *worker* prelevano le porzioni di matrice su cui lavorare. Quando i worker hanno terminato la loro computazione e la coda si è svuotata, un thread *collector* conclude il calcolo del chronon e, se necessario, invia la matrice sul socket. Di default le comunicazioni tra i due processi avvengono ogni 4 chronon, ma all'utente ha la possibilità di cambiare questo parametro tramite l'opzione `-v`.

Rispetto a quanto previsto dalla specifica, nel processo Wator è presente anche un ulteriore parametro `-d`, che permette di impostare un delay in millisecondi tra il calcolo del chronon e l'altro. Questo valore, che di default è 0, rallenta dell'evoluzione del pianeta, così da apprezzarne meglio i cambiamenti.

Tra i due processi, Wator e Visualizer, al primo è stata assegnata la funzione di **server** e al secondo quella di **client**. Questa scelta è stata fatta per rendere più *scalabile* il sistema: se in futuro si rendesse necessaria la creazione di più client Visualizer, ad esempio uno che stampa su schermo, un altro che scrive su file, un altro in esecuzione su una macchina remota (ecc.), la scelta fatta renderà più agevole le modifiche al progetto.

Per rendere più modulare il codice, l'implementazione del processo è stata divisa nei file:

1. `main.c`, che legge i file e i parametri in ingresso, crea i thread, il socket, lancia il processo Visualizer, gestisce i segnali POSIX e realizza la funzionalità di checkpoint (v. 3.2);
2. `farm.h`, contenente i prototipi delle funzioni che implementano la struttura a farm e le dichiarazioni delle variabili globali condivise tra i vari moduli;
3. `farm.c`, contenente l'implementazione dei loop eseguiti dal dispatcher, dal collector e dai worker.

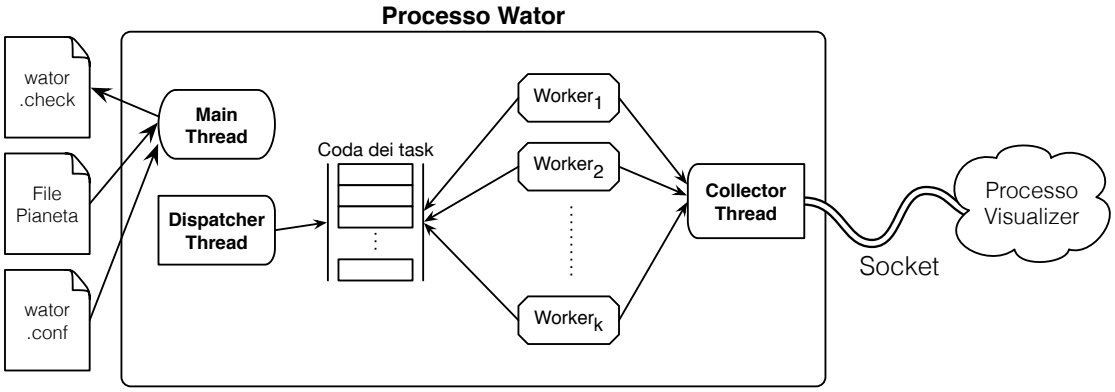


Figura 3.1: Schema del processo Wator

3.1. Algoritmo di suddivisione della matrice e sincronizzazione dei thread

Come già anticipato, la matrice di simulazione viene suddivisa in rettangoli sui quali più thread worker lavorano *parallelamente*. Gli algoritmi che si occupano di tale suddivisione e del trattamento di un rettangolo sono i più critici del programma: la soluzione deve essere *efficiente* e tale da evitare i *deadlock*, *bilanciare* il carico di lavoro ed assicurare che i thread non effettuino *accessi concorrenti* sulle stesse zone della matrice.

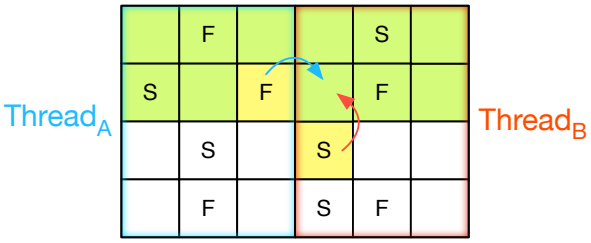
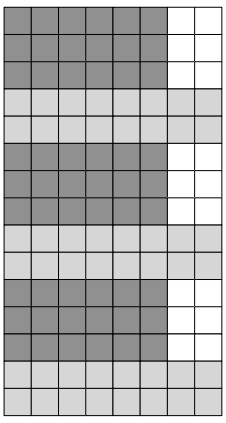


Figura 3.2: È mostrata una possibile divisione della matrice per due thread. In verde sono contrassegnate le celle già aggiornate da ognuno dei due thread, mentre le celle che stanno per essere calcolate hanno colore giallo. Lo squalo e il pesce compiendo i movimenti individuati dalle frecce, si contendono l'occupazione della stessa cella.

La necessità di evitare situazioni simili a quelle della Fig. 3.2, aveva condotto in un primo momento a prevedere l'uso di un mutex per ogni confine di un rettangolo da attivare ogniqualvolta un thread avrebbe cominciato l'aggiornamento una cella di confine. Tuttavia, ad una più attenta analisi, tale soluzione si è rivelata inadeguata, alla luce delle seguenti considerazioni:

- le operazioni di lock/unlock su un mutex possono essere molto dispendiose in termini di efficienza;
- stati in cui i pesci e i squali finiscono in poche partizioni della matrice (con il conseguente sbilanciamento del carico computazionale) sono molto rari, a causa della natura casuale degli spostamenti;
- l'organizzazione della matrice del pianeta in memoria è in *row-major order*: di conseguenza per massimizzare l'hit rate della cache, il programma deve generare indirizzi di celle consecutive nella stessa riga;
- in un sistema con n core, assumendo che il S.O. ospitante conceda al programma l'uso di tutti essi, si può ottenere un parallelismo reale lanciando la simulazione con n thread e suddividendo la matrice in base al numero di worker.

È stato quindi progettato un algoritmo nel quale la matrice del pianeta viene suddivisa in fasce orizzontali di altezza proporzionale al numero dei worker e distanziate da due celle di confine (che evitano le collisioni e di conseguenza la necessità di usare più variabili di mutua esclusione). Nella coda concorrente vengono aggiunti nell'ordine:



1. un primo gruppo di task per le fasce orizzontali (in grigio scuro nell'immagine accanto);
2. un secondo gruppo di task, aggiunto alla coda soltanto dopo che il primo è stato completato, che riguarda le fasce alte due celle che distanziano le fasce del punto precedente (in grigio chiaro);
3. un ultimo task per le ultime due colonne della matrice.

I thread worker sono sempre in attesa sulla coda dei task: appena ne arriva uno si legge il rettangolo corrispondente e lo si passa alla funzione `update_wator_rec`. Al completamento dei tre gruppi di task il thread collector viene svegliato tramite una variabile condizione e, se necessario, procede con l'invio della matrice appena aggiornata a Visualizer, secondo il protocollo che verrà descritto nella sezione 4.

3.2. Gestione dei segnali e checkpoint

È il thread principale del processo Wator che si occupa della gestione dei segnali. Di questi, `SIGINT` e `SIGTERM` provocano il cambiamento di valore della variabile globale `mustTerminateFlag` a true e l'invio del segnale `SIGUSR2` a Visualizer come notifica della terminazione del programma. L'arrivo di uno dei segnali `SIGUSR1` e `SIGALRM` indica invece che è giunto il momento di fare il *checkpointing*, cioè di salvare lo stato corrente del pianeta nel file `wator.check`. La chiamata di libreria `alarm()`, che causa la generazione di `SIGALRM`, ha permesso di implementare una funzionalità di "backup" periodico.

3.3. Terminazione della struttura a farm

La fase di *dispatch* e *collect* sono mutuamente esclusive. La terminazione della struttura a farm avviene nel seguente modo: il collector si accorge del cambiamento di valore della variabile globale `mustTerminateFlag`, invoca la funzione `destroy_queue` sulla coda, segnala tramite una variabile condizione al dispatcher di non generare più task, rilascia il mutex ed esce. Il dispatcher, dopo aver acquisito il mutex, riceve l'informazione di terminazione ed esce. Contemporaneamente i k worker in attesa sulla funzione `dequeue` si svegliano a causa della distruzione della coda, ricevono un task nullo ed interrompono il loro loop.

4. Il processo Visualizer e il protocollo di comunicazione

Il processo Visualizer (Fig. 4.1) esegue ciclicamente queste operazioni:

1. effettua diversi tentativi di connessione al socket ad intervalli di un secondo;
2. a connessione avvenuta, riceve le dimensioni della matrice ed alloca lo spazio per le celle (se e solo se le dimensioni sono cambiate dalla precedente iterazione);
3. riempie le celle con i dati ricevuti sul socket;
4. se ci sono errori esegue l'output, poi ritorna al punto 1.

La comunicazione sul socket avviene tramite lo scambio di due tipi di messaggi: il primo della dimensione di un intero senza segno, il secondo di 512 byte. Ad ogni connessione, il thread collector invia due messaggi di tipo 1 con il numero di righe e il numero di colonne della matrice. Segue una serie di messaggi di tipo 2 in cui ogni byte rappresenta il carattere contenuto in una cella della matrice.

Visualizer, a ricezione completata, può scrivere la matrice su un file oppure effettuare la stampa su schermo. L'operazione da eseguire viene stabilita una volta per tutte all'avvio del processo: se è stato invocato senza argomenti allora l'output è a schermo, altrimenti il primo argomento è il percorso del file in cui scrivere la matrice.

La stampa a schermo viene fatta con la funzione `print_planet_colored` della libreria Wator che, nei terminali che la supportano, colora i pesci di giallo, gli squali di rosso e l'acqua di blu.

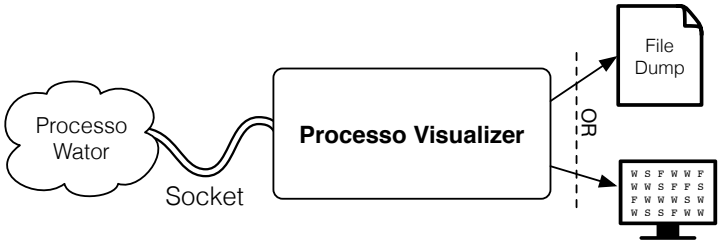


Figura 4.1: Schema del processo Visualizer

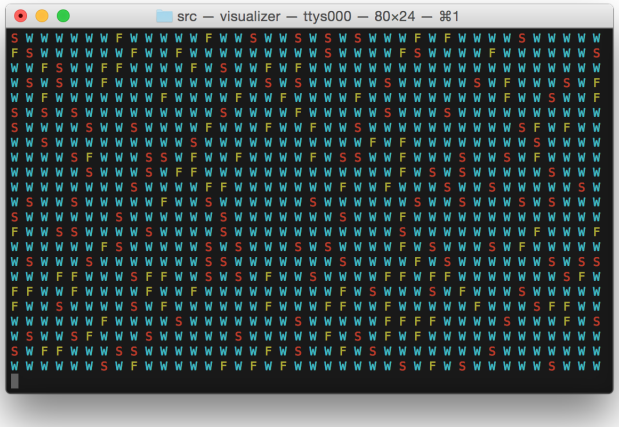


Figura 4.2: Output del Visualizer su schermo

5. Conclusioni

5.1. Problemi riscontrati e soluzioni adottate

Nel Makefile, tra i flag di compilazione di GCC, è stato aggiunto `--short-enums`, che minimizza la dimensione delle variabili di tipo enumerativo. L'effetto di questa scelta è una notevole riduzione dello spazio di memoria occupato dal programma: ad esempio `cell_t` che è il tipo **di ogni singola cella della matrice** passa da 4 a 1 byte.

Nei sistemi multiprocessore, thread in esecuzione su processori (o core) diversi potrebbero mantenere una copia nei registri di variabili globali, porzioni della matrice ecc., senza aver aggiornato le celle di memoria corrispondenti. Utilizzando la keyword `volatile` del C, è stato possibile risolvere alcuni problemi per i quali le modifiche alle informazioni condivise non venivano propagate tra i vari thread: ogni lettura/scrittura di una variabile volatile causa sempre un accesso in memoria. Nella libreria Wator, poiché non era possibile cambiare la segnatura delle funzioni, viene fatto il cast esplicito a `volatile` nel corpo della funzione (es. `*(volatile cell_t*)&planet[r][c]`).

5.2. Testing del programma

Insieme al codice sorgente è stata creata una cartella `test` per il collaudo del progetto. Le istruzioni per lanciare la test suite sulla libreria sono fornite nel file `LEGGIMI_TEST.md`.

Per testare il processo e verificare che non si presentino situazioni di stallo, è stato creato uno script Bash che genera un pianeta in modo casuale ed effettua n iterazioni della simulazione. La generazione del pianeta avviene scegliendo in modo casuale anche la densità di pesci e di squali, così da simulare vari carichi di lavoro. Si può eseguire lo script posizionandosi nella cartella `test` e lanciando il comando:

```
$ ./test_process numiterazioni durataiterazione n m
```

5.3. Sviluppi futuri e bug noti

Nelle versioni future del simulatore si potrebbe implementare il supporto a più client Visualizer in esecuzione su macchine remote. A quel punto sarebbe conveniente ottimizzare ancora di più la comunicazione sul socket: essendo tre le informazioni da spedire (W, S, F), sarebbero sufficienti $\lceil \log_2 3 \rceil = 2$ anziché 8 bit per codificare una cella. L'algoritmo di riempimento e lettura del buffer verrebbe però complicato dall'uso degli operatori bitwise del C.

Al momento della redazione della relazione non si riscontrano malfunzionamenti del programma, né si segnalano memory leak.