Wator

Progetto di Sistemi Operativi 2015

Nicoló Tonci
Corso B
503919

30 giugno 2015

Indice

1 Progetto Wator

Il progetto si propone la realizzazione di un simulatore parallelo del pianeta acquatico toroidale *Wator*[1]. L'architettura di riferimento per l'implementazione è quella della replicazione funzionale (Farm).
Sono stati sviluppati quindi algoritmi per risolvere le situazioni di *race condition* e di mutua esclusione che rappresentano il cuore dell'implementazione e la parte più significativa del progetto, andando a mettere in pratica ed usare buona parte delle tecniche viste a lezione e funzionalità della libreria POSIX pthread.
La realizzazione è divisa in due processi:

- `wator`: processo principale in cui avviene la simulazione (source: `wator_main.c` - `wator_main.h`);
- `visualizer`: stampa su uno specificato descrittore di file il pianeta ricevuto dal processo wator (source: `visulizer.c` - `visualizer.h`);

All'interno del processo principale coesistono essenzialmente 3 tipi di thread:

- Dispatcher: organizza e assegna il lavoro ai workers;
- Workers: eseguono la computazione nella loro porzione di dati;
- Collector: verifica e ricompone i risultati delle computazioni dei workers;

Inoltre vi è il thread principale del processo che si occupa della gestione dei segnali.
Le funzionalità specifiche e i dettagli implementativi sono descritti successivamente in questo documento.
Si fa inoltre uso della libreria, implementata nel primo frammento del progetto, nella quale vi sono le funzioni base del pianeta Wator, quindi l'allocazione del pianeta e delle regole che lo caratterizzano; si rimanda alla documentazione fornita dal docente e al file `wator.h` per le *signature* delle funzioni adottate.

1.1 Algoritmo di mutua esclusione

L'obbiettivo di un implementazione parallela del pianeta Wator è quella di aggiornare (spostare) più specie viventi contemporaneamente. Questo molto spesso comporta problemi di race condition, ad esempio due due threads che operano su due celle adiacenti ed entrambi tentano di scrivere sulla cella dell'altro. Prendiamo un'ulteriore esempio: la Figura 1 presenta una situazione ancora più complessa: le celle verdi sono squali, quelle blu pesci e le

[1] Wa-Tor è un automa cellulare ideato da Alexander Keewatin Dewdney e presentato sul numero di dicembre 1984 della rivista Scientific American.

Figura 1: Race Condition

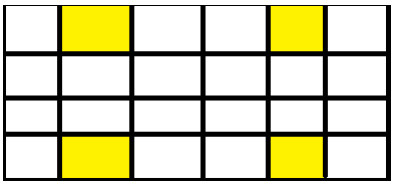

Figura 2: Possibile configurazione contemporanea di aggiornamento

frecce rappresentano i spostamenti che vorrebbero fare contemporaneamente. È chiaro che qualcosa verrà perso o vi si verificheranno dei conflitti.
Analizzando i possibili spostamenti che un pesce o uno squalo può fare notiamo che tra due celle aggiornate nello stesso momento devono esserci altre due celle per ovviare a problemi di conflitti o di race condition. Ad esempio nella Figura 2 le celle evidenziate in giallo sono quelle che possono essere aggiornate contemporaneamente.
Dunque l'idea dell'algoritmo implementato è processare righe in parallelo raggruppate per il loro indice modulo 3 e sincronizzarsi (esempio: aggiornare le righe di indice 0 mod 3 e sincronizzare, aggiornare poi le righe di indice 1 mod 3 ed infine le righe di indice 2 mod 3). Stessa idea per le colonne: processare le colonne in parallelo raggruppate per il loro indice mod 3. Le dimensioni delle sotto-matrici sono fissate (come nella specifica da opportune macro) ma è stato aggiunto il vincolo di essere (entrambe le dimensioni) multiple di 3. Le eventuali colonne e righe che avanzano dalla divisione vengono processate sequenzialmente dal thread collector.

1.2 Meccanismo di aggiornamento cella

L'aggiornamento (applicazione delle regole di Wator) di ogni singola cella avviene chiamando la funzione `update_cell(i,j)` dove `i` e `j` sono gli indici di riga e di colonna della cella da aggiornare.
Per assicurare che ogni pesce squalo sia aggiornato (mosso) una singola volta per chronon[2] è stata creata una matrice di controllo parallela con dimensioni uguali a quelle del pianeta nella quale viene salvato il numero dell'attuale chronon (type: int) quando una delle due specie si sposta in quella cella, esempio: la cella di coordinate i,j deve essere aggiornata, se essa contiene un pesce che si è spostato nella cella i,j+1, nella cella i,j+1

[2]unità temporale del pianeta Wator

della matrice parallela scriveremo il chronon attuale; stessa cosa vale per le nascite (esempio: nasce uno squalo in cella i,j, verra aggiornata la cella i,j della matrice di controllo). Questi valori servono per controllare, all'interno della funzione `update_check`, che la cella che si sta tentando di processare non sia stata scritta nel chronon attuale.

```
if (check\_up[i][j] < GetNchron(0)) ...
```

Se si supera il check: la cella viene aggiornata con le regole di Wator a secondo che vi sia un pesce o uno squalo, aggiornando la matrice di controllo se vi è una nascita o uno spostamento in una cella adiacente.
La funzione `update_cell(i,j)` è chiamata esclusivamente dai worker e da collector (per processare le righe e/o colonne di scarto).

1.3 Gestione dei segnali

La gestione dei segnali avviene nel main thread (del processo wator) in maniera asincrona, tramite la system call sigwait. I segnali catturati e i relativi comportamenti sono:

- SIGTERM e SIGINT: settano la variabile condivisa `toClose` a 1, usata per comunicare a tutti i thread del processo di terminarsi al completarsi del chronon corrente;
- SIGUSR1: stampa il pianeta nel file `wator.check`;

Tutti gli altri segnali al processo (che possono essere gestiti) sono catturati e ignorati.

1.4 Script `watorscript`

Lo script *watorscript* (implementato secondo le specifiche fornite dal docente) si propone di realizzare delle statistiche su dei pianeti (passati per parametro) calcolando il numero di pesci o di squali (a seconda dell'invocazione) oppure controllare se un pianeta è ben formattato (la matrice rispetta le dimensioni specificate e il contenuto di ogni cella è pertinente).

2 Il dispatcher

Le prime operazioni del dispatcher sono puramente matematiche: vengono calcolate quante sotto-matrici di dimensione fissata nelle macro `ROW` e `COL` ve ne sono per righe e per colonne (la loro somma è il numero totale di sotto-matrici in cui è stato diviso il pianeta di partenza), successivamente calcolo quante righe e quante colonne sono rimaste fuori dalla divisione in sotto-matrici (le celle appartenenti a queste righe e/o colonne saranno processate sequenzialmente dal thread collector una volta terminata la computazioni

di tutti i worker per quel chronon).
A questo punto si entra in un ciclo *for* di lunghezza pari al numero di chronon da simulare nella quale si procede ad inserire nella coda FIFO condivisa le sotto-matrici da processare, si avvisano i thread workers che la coda non è più vuota e ci si mette in attesa che il chronon attuale sia terminato (condition variable settata dal thread collector) prima di passare a quello successivo.
Il ciclo for oltre al numero di chronon da eseguire ha un'altra condizione, quella di chiusura (`closing()`) che lo termina appena diviene vera.
All'uscita del ciclo, cancello i thread workers e termino l'esecuzione.

3 Struttura della coda condivisa

Il mezzo di comunicazione tra i thread collector e tutti i workers è la coda condivisa, qui ne presentiamo la struttura e le funzionalità.
La coda è implementata mediante una *linked list* i cui record sono cosi strutturati:

```
typedef struct job {
   int i; /* coordinata base di riga della sottomatrice */
   int j; /* coordinate base di colonna della sottomatrice */
   unsigned char flag; /* if assigned 1, if completed 2 */
   job_t * next_job;
} job_t;
```

Il mutex relativo alla coda è `queue_mtx` e sono disponibili queste operazioni su essa:

- `enqueue(i,j)`: inserisce in coda una sotto-matrice di coordinate di base `i` e `j`;
- `assign()`: restituisce la prima sotto-matrice non completata ne assegnata disponibile e modifica il suo flag ad 1 (assigned), il mutex viene bloccato al suo interno;
- `completed(job)`: modifica il flag relativo al job passato per parametro a 2 (completed);
- `isEmpty()`: ritorna 1 (true) se la coda è vuota, 0 (false) altrimenti;
- `allassigned()`: ritorna 1 (true) se tutte le sotto-matrici sono state assegnate, 0 (false) altrimenti, il mutex viene bloccato al suo interno;
- `allcompleted()`: ritorna 1 (true) se tutte le sotto-matrici sono state completate, 0 (false) altrimenti;
- `erase_queue()`: elimina tutti gli elementi nella coda;

Per tutte le operazioni, tranne quelle dove esplicitamente espresso che il mutex viene bloccato al suo interno, è necessario aver ottenuto il mutex prima di ogni invocazione.

4 I worker

I worker sono il cuore del processo principale, qui viene applicato l'algoritmo di mutua esclusione presentato precedentemente nel paragrafo 1.1. Il numero di worker istanziabili è dinamico (all'invocazione dal parametro `-nwork` oppure se non specificato dalla macro `NWORK_DEF`. Il codice di questo thread sostanzialmente è contenuto in un ciclo infinito nella quale: ci si mette in attesa che vi siano elementi da processare nella coda, si controlla che tutti i "lavori" non siano stati assegnati, in questo caso se ne preleva uno e si inizia la computazione, in caso contrario si procede diversamente come descritto qui di seguito.
Il meccanismo di sincronizzazione usato nell'algoritmo di mutua esclusione usa delle barriere settate esattamente al numero di worker istanziati. Quindi ad ogni momento devono lavorare esattamente tutti i worker: per far si che tutti si sincronizzino pur non avendo da assegnare a tutti un lavoro per quel chronon, viene assegnato ad un worker una computazione fittizia nella quale vengono invocate solo system call delle barriere. Ciò è possibile farlo con le operazioni sulla coda condivisa.
Esempio: supponiamo di avere 5 worker e di dividere il pianeta in 12 sottomatrici: le prime 5 vengono processate in sincronia, poi si passa alle seconde 5 come le prime, infine le ultime due vengono assegnate a due thread ed i restanti 3 thread prendono in carico un "lavoro" fittizio.
Il codice seguente implementa l'algoritmo di mutua esclusione discusso precedentemente:

```
for(i=0;i<3;i++)
    for(j=0;j<3;j++){
       for(ii=job->i+i;ii<job->i+ROW;ii+=3) /* di 3 in 3 */
          for(jj=job->j+j;jj<job->j+COL;jj+=3) /* di 3 in 3 */
             update_cell(ii,jj);
       /* sincronizzazione per cui: tutte le righe % 3 = i
           e tutte le colonne % 3 = j sono state processte */
       pthread_barrier_wait(&brr);
    }
```

Completato il proprio "lavoro" il thread chiama `completed(job)` e successivamente controlla mediante `allcompleted()` se tutti sono stati completati, in tal caso sveglia il thread collector con la condition variable `task_cmp_cond` e immediatamente elimina tutto cio che è nella coda con `erase_queue()`, altrimenti si limita a re-iniziare da capo e mettersi in attesa di una nuova computazione.

5 Il collector

Il collector consiste in un ciclo infinito bloccato all'inizio dalla condition variable `task_cmp_cond` che viene modificata e sbloccata come prima descritto dall'ultimo worker che completa la computazione. A questo punto si passa a processare le righe e/o colonne che sono rimaste fuori dalla divisione in sotto-matrici, non processando due volte le celle che fanno parte sia delle righe di resto sia delle colonne di resto.
Successivamente, se il chronon attuale modulo chronon da eseguire è uguale a zero (ultimo chronon da eseguire) oppure sto per chiudermi (`closing() == 1`) invio al processo visualizer la matrice secondo il protocollo descritto nel paragrafo 6.1 con il flag di chiusura settato a 1.
Nel caso sia l'ultimo ciclo o meno sveglio comunque il dispatcher segnalandogli sulla condition variable `chron_cmp_cond` che il chronon corrente è stato completato ed è possibile passare al successivo (se ve ne è uno).
Prima di concludere il ciclo un semplice check sul flag di invio matrice mi dice se deve essere terminato il thread, in tal caso si esce dal ciclo e si termina.

6 Il processo Visualizer

Il processo visualizer (single thread) implementa un server in ascolto su uno specificato socket AF_UNIX, quando un client (processo wator) si collega ed invia i dati secondo il protocollo descritto nel paragrafo 6.1 il processo stampa sul file specificato come parametro al momento dell'invocazione (fork-exec dal processo visualizer) oppure se nulla è spcificato nello `stdout`, il pianeta ben formattato. Dopo la stampa, chiude il socket e si mette di nuovo in ascolto.
Il processo rimane attivo finché un client, nell'inviare i propri dati, specifica il flag di chiusura (descritto anch'esso nel paragrafo 6.1).

6.1 Comunicazione tra i processi wator e visualizer

La comunicazione tra i due processi avviene attraverso l'uso delle socket AF_UNIX (come prima descritto), utilizzate comunemente per la comunicazione tra processi risiedenti nella stessa macchina.
Il protocollo usato è quello presentato nella Tabella 1. Per diminuire il numero di byte scambiati, l'invio della matrice avviene attraverso la sua codifica in char (riducendo cosi di 3/4 i byte da inviare). I campi sono cosi strutturati:

- #row : numero di righe della matrice da visualizzare (tipo: intero - 4 byte)

- #col : numero di colonne della matrice da visulizzare (tipo: intero - 4 byte)
- DATA : array di lunghezza (#row * #col), contenuto della matrice (tipo: char - #row*#col bytes)
- flag: eventuali flag, nel nostro caso è servito per comunicare al visualizer di chiudersi (flag == 1) dopo aver terminato la stampa corrente, è possibile aggiungere ulteriori flag per altre comunicazioni o funzionalità implementabili (tipo: unsigned char - 1 byte)

Tabella 1: Protocollo

# row	**# col**	**DATA**	**flags**
4 byte	4 byte	(#row * #col) byte	1 byte

7 Compilazione e organizzazione file

Per la compilazione del progetto si consiglia di usare sequenzialmente `make lib`, `make wator`, `make visualizer` e per l'attivazione si consulti il documento messo a disposizione dal docente.

7.1 File

- `wator.c` (`wator.h`): codice della libreria *libWator*
- `wator_main.c` (`wator_main.h`): codice del processo *wator*
- `visualizer.c` (`visualizer.h`): codice del processo *visualizer*

Ulteriori chiarificazioni sul comportamento di ogni singola parte sono possibili consultando i commenti allegati al codice.

Nicolò Tonci
503919