Progetto Wator

Leonardo Lurci

Giugno 2015

Indice

Capitolo 1

Struttura del codice

1.1 Motivazione della suddivisione in moduli

Il progetto è stato suddiviso in moduli cosi da evidenziarne la struttura, inoltre aumenta la flessibilità del codice potendo modificare un modulo senza avere ripercussioni ai sorgenti che lo implementano. Ogni modulo corrisponde a porzione indipendente del progetto, mentre una loro *"adeguata"* unione rappresenta il programma richiesto dalle specifiche del progetto.

Ogni modulo può contenere delle funzioni **static**, e quindi visibili solo al file .c in cui essa è definito, che servono per modellare la logica del codice. Ad esempio in wator_wator.c

```
static int readWatorConf(wator_t* wator)
{
    ...
}

wator_t* new_wator (char* fileplan)
{
    int error = 0;
    wator_t* wator = (wator_t*) malloc(sizeof(wator_t));
    planet_t* planet = NULL;
    FILE* inputPlanet = NULL;

    if(wator == NULL)
            return NULL;

    error = readWatorConf(wator);
    ...
}
```

La funzione readWatorConf(wator_t wator) ci permette di evidenziare la logica relativa alla lettura del file wator.conf separandola dalla logica di inizia-

lizzazione della struttura wator_t*.
Ogni file.h contiene le dichiarazioni delle funzioni con le relative specifiche.

1.2 Il modulo wator.h

Il modulo wator.h si occupa della creazione, gestione ed eliminazione del pianeta su cui dobbiamo effettuare la simulazione. Le definizioni delle funzioni di wator.h sono state a loro volta suddivise in più file .c in modo raggruppare le funzioni che elaborano la stessa struttura dati.

- wator_wator.c: contiene le definizioni delle funzioni che interagiscono con la struttura dati wator_t*
- wator_planet.c: contiene le definizioni delle funzioni che interagiscono con la struttura dati planet_t*
- wator_animal.c: contiene le definizioni delle regole degli squali e dei pesci
- wator_util.c: contiene le definizioni delle funzioni di supporto per semplificare la gestione delle strutture dati di wator.h.

1.3 Il modulo queue.h

Il modulo queue.h si occupa della creazione e gestione di una semplice coda. Queue dichiara delle semplici funzioni **push(myQueue), pop(myQueue)** relative l'inserimento ed estrazione in testa, e la funzione **isEmpty(myQueue)** per verificare se la coda è vuota o contiene almeno un elemento. Le definizioni delle funzioni di queue.h sono contenute all'interno del file. **queue.c**

1.4 Il modulo threadPool.h

Il modulo threadPool si occupa della creazione e gestione dei thread dispatcher, workers, collector. Le definizioni delle funzioni sono suddivise in due file .c

- threadPool_init.c: contiene le definizioni delle funzioni che inizializzano la struttura dati principale e si occupa della gestione dei segnali.
- threadPool.c: contiene le definizioni delle funzioni che gestiscono i thread principali e la comunicazione con la socket.

ThreadPool.h si tratta del modulo più importante dell'intero progetto in quanto contiene tutte le funzioni relative allo svolgimento della simulazione.

1.5 Il modulo macro.h

Il modulo macro.h contiene le **#DEFINE** comuni a tutti i moduli sopraelencati.

Capitolo 2

Strutture dati utilizzate

Il processo wator, per effettuare la simulazione, si serve delle seguenti strutture dati:

- da wator.h: un mondo wator_t con il relativo planet_t per la gestione del pianeta.
- da queue.h: la coda dove verranno inseriti ed estratti i task.
- da threadPool.h: la struttura dati principale threadPool contenente
  - la coda dei task.
  - la matrice di sincronizzazione per la simulazione del pianeta in multithreading.
  - i thread dispatcher, worker, collector.
  - le lock e le relative variabili di condizione.
  - i flag per la sincronizzazione dei thread.

2.1 Le strutture di wator.h

Le strutture dati wator_t e planet_t sono esattamente quelle proposte nel frammento 1.

2.2 Le strutture di queue.h

queue.h contiene tre semplici strutture dati per la creazione e gestione dei task.

- task: struttura dati che rappresenta lo scheletro di un task, formato da
  - i, j: gli indici relativi al quadrante della matrice di sincronizzazione.
  - startXY, stopXY: offset del pianeta da elaborare.

- queue: struttura dati che rappresenta un singolo elemento della coda. Un elemento della coda è composto dal puntatore al task inserito e un secondo puntatore al prossimo elemento della coda.
- myqueue: struttura dati che contiene la coda da gestire e il numero di elementi presenti al suo interno.

2.3 Le strutture di threadPool.h

threadpool.h contiene le strutture dati principali per la simulazione del pianeta, che sono:

- KNMatrix: struttura dati relativa alla matrice della riduzione in scala (K*N) del pianeta.
- workerargs: struttura dati che contiene gli argomenti da passare ai worker.
- threadPool: struttura dati principale, si occupa della simulazione.

2.3.1 KNMatrix

```
typedef enum _status {WAITING, RUNNING, DONE} status;

struct __KNmatrix
{
        int nrow;
        int ncol;
        status** matrix;
};
```

KNMatrix si occupa della sincronizzazione fra i worker che devono elaborare quadranti adiacenti del pianeta. Ogni quadrante può assumere tre stati:

- WAITING: il quadrante sta aspettando di iniziare la sua *evoluzione*
- RUNNING: il quadrante sta eseguendo la sua *evoluzione*
- DONE: il quadrante si è *evoluto*

Ad ogni chronon la matrice matrix viene inizializzata a WAITING per permettere una nuova *evoluzione* del pianeta. L'utilizzo di degli stati **status** ci permette di sincronizzare i worker e quindi di evitare (insieme alla matrice flagMap presente in threadPool) di muovere un animale più di una volta per chronon.

2.3.2 workerargs

La funzione

```
int pthread_create(pthread_t *thread, const pthread_attr_t *attr,
                   void *(*start_routine) (void *), void *arg);
```

ci limita di passare al massimo un singolo argomento al thread che stiamo creato. I thread worker hanno bisogno di due argomenti: il proprio wid e la struttura dati principale della gestione dei thread.

Per ovviare al problema del passaggio di un singolo argomento utilizziamo la struttura dati **workerargs** che ci permette di incapsulare **wid** e **threadPool** per poi passarli ai rispettivi worker.

```
struct __workerargs
{
        int n;
        threadPool tp;
};
```

2.3.3 threadPool

Threadpool è la struttura dati principale contenente tutte le informazioni relative alla simulazione del pianeta:

- taskQueue, la coda contenente i task da elaborare.
- threads: dispatcher, workers, collector e signal_handler.
- KNM, matrice di sincronizzazione dei worker.
- flag di sincronizzazione dei thread: run, close, collectorFlag, workFlag.
- le mutex queueLock e KNMLock.
- le variabili di condizione: waitingDispatcher, waitingCollector, waitingTask e waitingWorkers.
- flagMap, matrice di controllo degli spostamenti dei pesci e squali
- wator, il mondo da simulare.

flagMap FlagMap è una matrice di dimensione pari a quella del pianeta in elaborazione. Ogni cella i, j di flagMap assume due stati:

- MOVE: l'animale presente in i, j può eseguire le sue regole.
- STOP: l'animale presente in i, j ha già eseguito le sue regole, oppure in posizione i, j non è presente alcun animale.

Ad ogni chronon le celle i, j di flagMap vingono inizializzate a **MOVE** se e solo se in posizione i, j è presente un animale, **STOP** altrimenti. FlagMap, insieme alla matrice KNM ci assicurano che ogni animale può eseguire le proprie regole una e una sola volta per chronon durante l'esecuzione della funzione **evolve(...)**.

run Il flag run permette ai thread di lavorare. Run viene settato a 1 durante la inizializzazione di threadPool (**initpool()**), mentre viene settato a 0 dal thread Collector se e solo se il flag close è stato settato a 1.

close Il flag close serve per iniziare la terminazione della simulazione. Se close viene settato a 1 allora il thread collector setterà il flag run = 0 dopo aver eseguito l'ultima stampa.

queueLock e KNMLock QueueLock, KNMLock sono le mutex usate dai thread per evitare la race condition su **taskQueue** e KNM. Il loro uso verrà spiegato nel capitolo 3.

waitingDispatcher, waitingWorkers, waitingCollector Queste sono le variabili di condizione su cui i thread possono effettuare le wait. Il loro uso verrà spiegato successivamente.

2.3.4 strutture dati interne a threadPool_init

il file.c threadPool_init oltre alle funzioni d'inizializzazione contiene anche la gestione dei segnali e quindi gli opportuni flag:

```
...
static volatile int flag_alarm = 0;
static volatile int flag_check = 0;
static volatile int flag_create = 0;
static volatile int flag_close = 0;
...
```

- flag_alarm: serve per avviare alarm() una e una sola volta. Ricevere n SIGUSR1 non deve permettere l'avvio di n alarm().
- flag_check: viene settato a 1 ogni SECS secondi per poter stampare sul file wator.check.
- flag_create: viene settato a 1 insieme al primo avvio di alarm(), serve per creare il file wator.check solo se è richiesto.
- flag_close: se settato a 1 avvia la terminazione della simulazione.

nota: i flag di threadPool.h sono stati dichiarati **volatile** per avvisare il compilatore di non effettuare spostamenti nel caso stia ottimizzando un segmento di codice dove sono presenti tali flag.

Capitolo 3

La simulazione

La simulazione viene gestita dai thread dispatcher, workers, collector. Le condizioni iniziali settate dalla funzione

```
int initpool(threadPool tp, wator_t* w)
{
    ...
        tp->run = 1;
        tp->workFlag = 0;
        tp->collectorFlag = 0;
        tp->close = 0;
    ...
}
```

impongono che il thread dispatcher sia il primo a partire.

3.0.5 Dispatcher

Il thread dispatcher ha il compito di riempire la coda dei task. Per riempire la coda acquisisce la lock su **queueLock**, se la coda è vuota la riempie tramite la funzione **populateQueue(...)**, rilascia la lock, setta a 1 il flag workFlag ed effettua una broadcast per svegliare i worker. Se la coda non è vuota allora vuol dire che i thread worker la stanno elaborando, dunque dispatcher si mette in attesa sulla variabile di condizione **waitingCollector**.

Alla chiusura del dispatcher viene eseguita una broadcast sui worker per poterli terminare.

populateQueue(threadPool tp) è la funzione che suddivide il pianeta in matrici K*N, crea i relativi task e l'inserisce all'interno della coda tramite la funzione **push()**

3.0.6 Worker

Il thread worker ha due fasi principali: la fase d'inizializzazione dove il worker crea la sua firma **wator_woker_*wid*** e il ciclo di esecuzione. Nel ciclo di esecuzione il worker acquisisce la lock su **queueLock** e verifica se **workFlag** = 0 oppure se la coda è vuota, in tal caso vuol dire che il dispatcher deve ancora preparare i task e quindi il thread worker si mette in attesa sulla variabile di condizione **waitingDispatcher**. Se **workFlag** = 1 e la coda non è vuota allora worker ritira il task in testa alla coda tramite la funzione **pop()**, rilascia la lock su **queueLock** per acquisire quella su **KNMLock**. A questo punto il thread worker deve verificare se il quadrante relativo al suo task non è adiacente a nessun'altro quadrante in evoluzione (status = **RUNNING**), in tal caso rilascia la lock, setta a **RUNNING** il quadrante relativo al task ed esegue la funzione di evoluzione **evolve(task, wator_t*, int**)**, altrimenti si mette in attesa sulla variabile di condizione **waitingTask**. Una volta completata la funzione evolve, il thread worker acquisisce nuovamente la lock su **KNMLock** per settare lo stato del task a **DONE** e rilascia la lock. Alla fine controlla se tutti i quadranti si sono evoluti, in tal caso setta a **collectorFlag** a 1 ed effettua una signal su **waitingWorkers**, infine prima di eseguire un'altro ciclo effettua una broadcast su **waitingTask**. Alla chiusura del threadWorker viene effettuato una broadcast su **waitingDispatcher** per svegliare i worker che termineranno a causa del flag **tp-run** settato a 0.

nota: i thread worker oltre che a controllare il flag **workFlag** e **!isEmpty()** controllano anche che **tp-run** sia settato a 1, questo ulteriore controllo serve per terminare tutti i thread worker che sono rimasti in attesa del dispatcher nel caso in cui tp-run sia settato a 0.

evolve: è la funzione che si occupa di *evolvere* la porzione del pianeta relativo al task passato come parametro. Evolve vuole come parametro: il task da elaborare, il mondo su cui deve effettuare *evoluzione* e la flagMap per eseguire le regole degli animali una e una sola volta per chronon.

3.0.7 Collector

Il thread collector ha il compito di stampare il pianeta e resettare le strutture dati per un nuovo chronon. Nel ciclo di esecuzione il thread collector acquisisce la lock su **KNMLock**, verifica se **collectorFlag** è settato a 0 e in tal caso si mette in attesa su **waitingWorkers**, altrimenti invia il pianeta alla socket se il chronon corrente è un multiplo di chronon. Successivamente resetta KNMmatrix, flagMap e effettua una signal su **waitingCollector** per avvisare il thread dispatcher che è possibile partire con una nuova *evoluzione*.

3.0.8 Signal_handler

il thread signal_handler ha il compito di gestire i segnali che riceve dall'esterno. Signal_handler ha diversi comportamenti a seconda dei flag attivati:

- se close è settato a 1 inizia ad avviare la terminazione del processo settando **tp-run** a 0
- se create è settato a 1 apre il file **wator.check**
- se check è settato a 1 scrive il pianeta all'interno del file **wator.check** e setta **check** a 0

Capitolo 4

La comunicazione

La comunicazione del pianeta avviene tramite la socket **visual.sck** che è condivisa tra il thread collector e il processo visualizer. La comunicazione viene eseguita nelle seguenti fasi:

fase 0.0: collector effettua la connessione con la socket.

fase 1.0: visualizer accetta la connessione del collector.

fase 2.0: collector prepara il buffer con la i-esima riga da inviare.

fase 2.1: collector effettua la write sulla socket scrivendoci il buffer appena preparato.

fase 3.0: visualizer effettua una read sulla socket e riceve il buffer da stampare.

fase 3.1: visualizer stampa il buffer ricevuto ed effettua una write sulla socket scrivendo "ok".

fase 4.0: collector effettua la read sulla socket. Se c'è altro da stampare allora si riparte dalla fase 2.0, altrimenti si passa alla fase 5.0.

fase 5.0: collector chiude la connessione con la socket.

fase 6.0: visualizer attende una nuova connessione.

4.1 Nota sulla composizione del buffer

Il progetto richiede che il passaggio dei dati sulla socket sia il minimo possibile. Per rispettare questo vincolo il numero di righe / colonne viene passato durante la creazione del processo visualizer tramite il processo wator. Il buffer contiene esattamente ncol+1 elementi ovvero, i caratteri WFS privi di spazi e il carattere terminatore '\0'. Il processo visualizer è in grado di ricostruire la riga aggiungendo gli spazi tra ogni carattere.

Capitolo 5

Wator & Visualizer

Wator rappresenta il processo lato client mentre visualizer è il processo servente per la stampa del pianeta.

5.1 Wator

Wator è il processo principale che avvia la simulazione del pianeta. Wator deve essere eseguito passandogli come argomento il nome del file.dat del pianeta su cui applicare la simulazione. Possiamo aggiungere altri parametri facoltativi come

- -n il numero di worker che devono eseguire la simulazione.
- -v ogni quanti chronon deve essere effettuata la stampa.
- -f il filedump dove vogliamo stampare il pianeta.

Una volta avviato correttamente il processo, wator inizializzerà tutte le strutture dati per la simulazione, avvierà il processo Visualizer con argomenti nrow, ncol (numero di righe e colonne del pianeta) effettuando una dup2 stdout - filedump nel caso sia stato specificato l'argomento -f, infine avvierà l'esecuzione della simulazione tramite la chiamata **initpool(...)** e si metterà in attesa dei thread tramite la funzione **makeJoin()**.
Alla cattura di un segnale SIGINT / SIGTERM i thread termineranno e questo porterà il thread Main ad continuare la sua esecuzione dopo le join, cosi facendo verranno liberate tutte le strutture dati utilizzate e il processo visualizer terminerà tramite una **kill(..)** con il segnale SIGTERM, infine il processo wator verrà terminato.

5.2 Visualizer

Il processo visualizer viene avviato da wator tramite una fork, execl e una possibile dup2 stdout - filedump nel caso wator sia stato avviato con argomento

-f. Visualizer inizia la sua esecuzione verificando se sono stati passati i due parametri interi che rappresentano il numero di riga / colonna del pianeta da stampare. Se visualizer è stato avviato soddisfacendo i parametri richiesti allora parte la fase d'inizializzazione del processo: viene creata la maschera dei segnali da gestire, viene eliminata la socket se gia presente all'interno del file, crea una nuova socket su cui applica le funzioni **bind(...), listen(...)** e si mette in attesa di un cliente. Quando un cliente si collega a visualizer, si attiva il protocollo di comunicazione descritto nel capitolo 4.
Alla cattura di un segnale SIGTERM il processo setta a 0 la variabile **run** che gestisce il ciclo vita del thread main, cosi facendo il processo termina.

Capitolo 6

Watorscript

Watorscript è uno script che possiamo utilizzare per verificare che un certo pianeta file.dat sia ben formattato secondo le specifiche del progetto, oppure per ricavare il numero di pesci / squali presenti in un determinato pianeta.

Lo script si suddivide in tre principali funzioni:

- checkParam, per la verifica dei parametri con cui abbiamo avviato lo script.
- checkFile, per la verifica della formattazione di un pianeta file.dat.
- countFishShark, per il conteggio dei pesci e degli squali.

Watorscript avvia sempre le funzioni checkParam e checkFile. CountFishShark viene avviata solo su richiesta tramite gli argomenti -f -s.

6.1 checkParam

La funzione checkParam verifica inizialmente se esiste un argomento diverso da -s -f -h ed in caso di esito positivo stampa help. Se i parametri passati rientrano fra quelli accettati allora viene effettuato il controllo su eventuali doppioni, se l'esito è negativo verifichiamo che sia stato passato il nome del pianeta su cui effettuare le operazioni. La verifica del file.dat viene effettuata controllando che il numero di parametri ancora da parsare e che non hanno il prefisso "- / –" sia esattamente 1 e che la stringa passata sia relativa a un file, se l'esito del controllo risulta negativo stampiamo help.

6.2 checkFile

La funzione checkFile apre il file.dat tramite la funzione exec. Una volta aperto il file contenente il pianeta viene esaminato il suo contenuto. I primi due controlli consistono nella verifica che le prime due righe del file contengano solo due

numeri cosi formati: la prima cifra deve essere fra 1-9, le cifre a seguire devono essere tra 0-9.

```
...
if ! [[ $nrow =~ ^[1-9][0-9]*$ || $ncol =~ ^[1-9][0-9]*$ ]] ; then
        printError
fi
...
```

Se il controllo di row, col ha esito negativo stampiamo un messaggio di errore, altrimenti procediamo con l'analisi della formattazione del pianeta. L'analisi del pianeta viene effettuata leggendo una riga alla volta. Data una riga del pianeta essa deve

1. deve iniziare con un carattere WFS seguito da uno spazio.
2. dal secondo carattere al ncol-1-esimo deve essere formata da un carattere WFS seguito da uno spazio.
3. deve finire con un carattere WFS.

```
...
if ! [[ $line =~ ^([WFS][[:space:]]){$(($ncol-1))}[WFS]$ ]] ; then
        printError
fi
...
```

Il numero di righe del pianeta viene controllato tramite un contatore che viene incrementato a ogni riga letta, se alla fine del ciclo il numero di righe nrow non è uguale al numero di righe contate si stampa un messaggio di errore, altrimenti si stampa "OK".

6.3 countFishShark

La funzione countFishShark, dato un parametro S / F a seconda della specie che vogliamo contare, utilizza grep in pipe con wc. Grep viene eseguito con l'argomento -o in modo da passare a wc esclusivamente i match fatti, wc a sua volta viene utilizzato con l'opzione -l cosi che stampi il numero di match ricevuti da grep.