WATOR

0.1 – Contenuto del progetto

Questa relazione è volta a commentare gli aspetti principale di implementazione del progetto Wator.
Il progetto è realizzato mediante i seguenti file (a partire dal wator.h fornito) :

- core.h : header di una libreria che comprende funzioni di utilità quali code, ecc...
- core.c : implementazione della precedente
- wator.first.c : implementazione del primo frammento con leggere modifiche per funzionare come implementazione di wator.h per i frammenti successivi. Al fine di avere il comportamento originale di questo programma, vedere la sezione opzioni di compilazione .
- main_header.h : header che raccoglie le principali costanti non definite in core.h e qualche metodo che verrà implementato nei successivi file.
- socketutils.c : implementa dei metodi di utilità per scrivere la sulla socket.
- dispacher.c : contiene il main e funzioni “private” del thread dispacher.
- collector.c : contiene il main e funzioni “private” del thread collector.
- worker.c : contiene il main e funzioni “private” del thread worker.
- wator.c : contiene il main del processo, e funzioni di inizializzazione / distruzione.
- visualizer.c : sorgente del processo visualizer.
- watorscript : script richiesto dalle specifiche.

0.2 – Bug noti

Non sono noti Bug.

0.3 – Gestione degli errori

La gestione degli errori è particolarmente rigida:

- Un errore di connessione alla socket può esser ritenuto legittimo per cui, prima di esser considerato un errore, l'operazione viene reiterata un numero limitato di volte.
- Ogni altro tipo di errore è considerato “fatale” nel senso che causa la terminazione immediata del processo. Un messaggio di errore viene visualizzato.
- In caso di fault del visualizer, il processo wator realizza l'errore dopo le ripetute connessioni fallite.
- Non è gestita la terminazione di V in seguito di un fault di wator (si ritiene sufficientemente robusto da non generare errori dopo che V sia stato avviato)

1 – Componenti

Il progetto si compone delle seguenti componenti conformemente alle specifiche (con alias):

- Processo wator : composto da
  - M) Thread principale
  - D) Thread dispacher
  - C) Thread collector
  - Wi) Thread worker i (con i che va da 0 ad nwork -1)
- V Processo visualizer

Ogni componente al momento della propria attivazione, dopo una serie di controlli e inizializzazioni entra in un ciclo “infinito” (da ora in poi verrà chiamato event loop) dal quale sarà possibile solo un'uscita gentile. Di seguito con componenti si intenderà una delle precedenti.

2 – Sincronizzazione delle componenti

La sincronizzazione tra thread e processi è realizzata mediante una coda sincronizzata, che di seguito verrà indicata come coda degli eventi (nel caso di C → V invece verrà usata come “coda” la socket, ma in seguito non ne verrà fatta distinzione).
La coda realizza il modello produttore-consumatore ed è implementata mediante una lista, quindi

- La enqueue può sempre avvenire, viene eseguita in mutua esclusione e genera una signal.
- La dequeue può avvenire solo se almeno un elemento è presente nella coda, e viene fatta in mutua esclusione. Nel caso in cui la coda sia vuota, allora la componente consumatore si mette in wait, aspettando una signal dal produttore.

In particolare ogni componente, è il consumatore di una coda degli eventi, mentre uno dei produttori è una qualunque altra componente. Il ciclo degli eventi di ogni componente estrae un comando da una coda degli eventi, e la esegue.
N.B. poiché la enqueue non è bloccante, quindi, non si ha rischio di dead lock.
In genere la coda degli eventi ha un solo consumatore (la componente associatavi), tuttavia W* sono tutti consumatori di una stessa coda degli eventi, mentre hanno un solo produttore: D .
La coda degli eventi appena citata assumerà lo scopo specifico di esser il pool delle matrici da elaborare (di seguito indicata semplicemente come pool).
Per quanto riguarda la socket, non è necessario sia sincronizzata:

- La comunicazione è unilaterale e si ha un solo “consumatore” (lettore), ovvero V .
- Benché le scritture possano esser fatte da componenti diverse (esempio: la chiusura di V è comandata mediante socket da M, mentre la richiesta di visualizzazione del pianeta, viene richiesta da C), non si ha rischio di avere una race condition poiché le operazioni di scrittura vengono effettuati in momenti precisi, e implicitamente mutuamente esclusi mediante la gestione ad eventi (vedi evoluzione della comunicazione).

3 – Evoluzione della comunicazione

L'evolversi della comunicazione segue il seguente modello a partire da M che estrae un messaggio attraverso le code degli eventi:

M:
- se: M ← “exit”
  - M do “exit” → D
    - D ← “exit”
      - D do “exit” → C e termina.
        - C ← “exit”
          - C termina.
  - M do “exit” → W* e termina.
    - W* ← “exit”
      - W* temina.
- se: M ← “update”
  - se: “segnale exit”
    - M do “exit” → M
    - M do “last show” → C
      - C ← “[last] show”
        - C do <“visualizza”,planet> → V
          - V ← <“visualizza”,planet>
            - V visualizza planet
        - se msg è “show” senza “last” : C do “update” → M
  - altrimenti: M do “update” → D
    - D ← “update”
      - D do <matrice da aggiornare> → Wi
        - Wi ← <matrice da aggiornare>
          - Wi aggiorna <matrice da aggiornare>
          - Wi do <i> → C
            - C ← <i>
              - se C ha num_of_subs indici
                - se : sono passati chronon updates
                  - C do “show” → C
                - altrimenti : C do “update” → M

Si consideri, potendo ogni componente, eseguire un solo comando della coda degli eventi alla volta, non si possono incrociare i rami dell'albero sopra illustrato.
Nell'implementazione del pool, i messaggi sono la referenza alla matrice da aggiornare; qualora questo valore fosse 0, quindi Null, viene considerato come messaggio di exit. Wi al termine dell'elaborazione della sotto matrice, come indicato precedentemente, invierà a C il proprio wid (ovvero il proprio indice i). Poichè tutte le code lavorano su tipi primitivi void*, per trasmettere il wid, viene fatto ricorso all'indirizzo che lo contiene. C quindi riceverà un valore che considererà come messaggio esplicito (definito da delle define) se ha valori bassi, altrimenti, verrà considerato un indirizzo e quindi un messaggio rappresentante un generico Wi. Si fa quindi affidamento sul fatto che gli indirizzi di variabili allocate nello Heap non possano essere troppo vicine a 0.

4 – Sotto matrici

Una volta definito come le componenti “dialogano”, è necessario chiarire cosa sia l'oggetto della manipolazione della routine di un Wi. Infatti esso, dome indicato dalle specifiche e illustrato precedente, estrarrà una matrice dal pool (sul quale tutti i W* concorrono) ed inizierà a lavorare su di essa. Ma che cosa è in concreto ciò che estrae dal pool ? Una sub_planet_t! (o meglio, per lo stesso motivo espresso al termine della precedente sezione, una referenza ad essa).
Una sub_planet_t è sostanzialmente quella che intende essere una sotto matrice, sulla quale dovrebbe operare un singolo Wi. In particolare essa è definita da una dimensione _nrow, _ncol ed una “cell” matrice di real_cell_t (definite nella sezione 5 – Real Cell). _nrow ed _ncol sono sostanzialmente K ed N indicati delle specifiche, tuttavia, poiché non è fatta assunzione che K ed N siano dei divisori delle rispettive dimensioni di planet, allora potrei avere 4 tipi di matrici diverse che si differenziano per dimensioni:

- Matrici KxN
- Matrici (nrow%K)xN (in basso)
- Matrici Kx(ncol%N) (a sinistra)
- Matrici (nrow%K)x(ncol%N) (in basso a destra)

Da qui la necessità di memorizzare anche la dimensione di ogni sotto matrice.
La matrice “cell” di real_cell_t è sovradimensionata rispetto _nrow ed _ncol, ciò è volto ad enfatizzare quello che verrà poi chiarito nella sezione 6 – Funzionamento di una sub planet. In particolare “cell” ha dimensione (_nrow+WEIGHT*2)x(_ncol+WEIGHT*2) dove WEIGHT sarà in seguito definita, e rappresenta l'area di planet nella propria sotto area con indici che partono da WEIGHT e terminano a WEIGHT+_n<dim>.

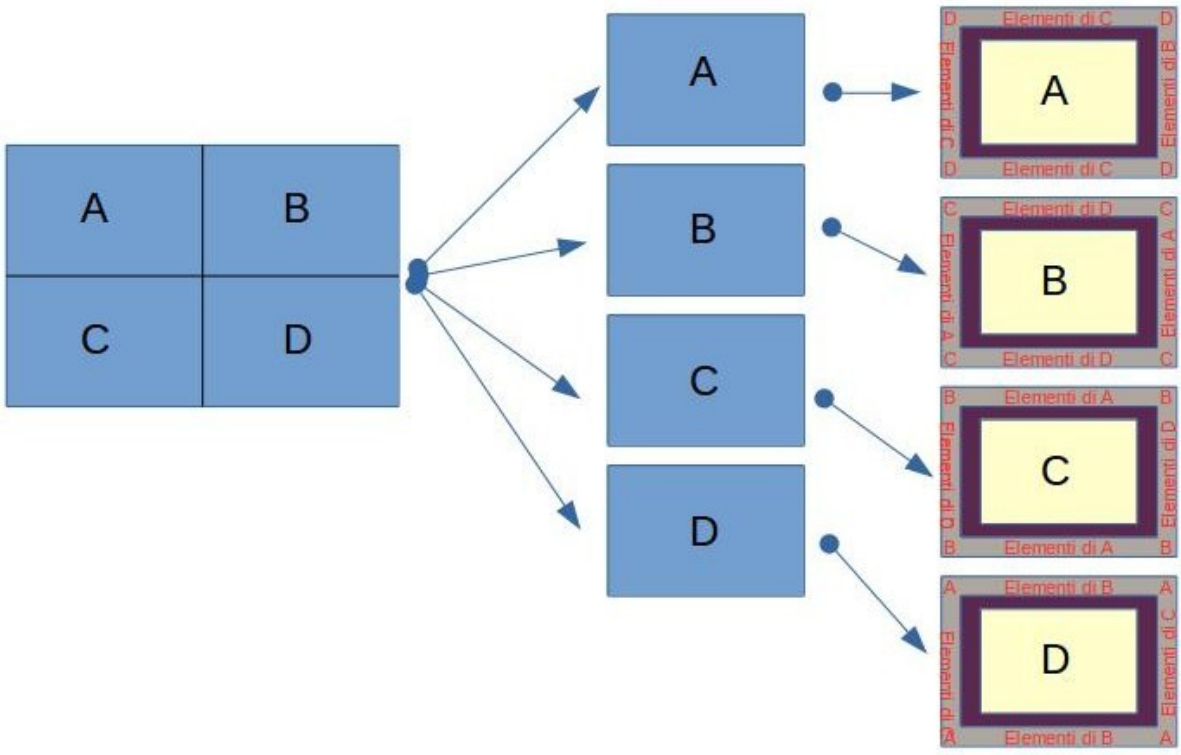
5 – Real Cell

Una real cell è sostanzialmente un agglomerato di referenze a planet (poiché come indicato dalle specifiche, dovrà essere la matrice su cui si opera immediatamente), con le rispettive coordinate i,j nel sistema di riferimento di planet. Si noti che le funzioni di applicazione delle <animale>_rule<#> vengono invocate passando tali indici ed esse sono sostanzialmente quelle del primo frammento. Quindi la modalità di aggiornamento di una cella è invariata, anche se in multithreading, poiché sarà compito di un generico Wi garantire che tutte le celle che verranno coinvolte nell'applicazione della “rule” siano safe. Ciò avviene grazie alle real cell che conterranno anche una mutex ed uno stato. Lo stato servirà per evitare che un animale venga mosso ripetutamente. La mutex è una per cella, in particolare, ad ogni cella di planet, viene associata una ed una solo mutex oppure null. Quindi benchè real cell diversse appartengano a sotto matrici divere, esse avranno la stessa referenza a mutex.

6 – Funzionamento di una sub Planet

Prima di tutto occorre fare un osservazione : poiché un animale prima si riproduce e poi si muove, esso può raggiungere una area intorno a se limitata dalle potenziali celle in cui può

muoversi. Sarebbe stata un'area più grande se un animale prima si fosse mosso e poi si fosse riprodotto (circa un rombo attorno ad esso). Nel caso in cui la riproduzione avvenga prima del movimento si ha un rombo più piccolo che da ora in poi sarà indicato come “il rombo”. Sia dunque WEIGHT = 1 cioè la massima distanza in valore assoluto che ogni coordinata può raggiungere rispetto a quella di partenza.
Sia planet la matrice originale, essa verrà divisa in tante sotto matrici, ognuna delle quali conterrà riferimenti alla matrice originale, nella seguente maniera:



In sostanza ad ogni sotto matrice, viene aggiunto un bordo (quello grigio) che contiene riferimenti alle aree limitrofe. La cornice grigia è spessa tante celle quanto è WEIGHT. Ogni sotto matrice a sua volta avrà una cornice interna che pur essendo nella propria area, risulta anche nella cornice delle sotto matrici vicine (l'area viola). L'area gialla è quella appartenente alla sotto matrice e non raggiungibile (e quindi non modificabile) a partire da qualunque altra sotto matrice. Ogni sotto matrice “possiede” l'unine della propria area gialla con quella viola, dove con possiede si intende che è l'area di planet ad essa associata.
Ma cosa si intende con una cella è raggiungibile a partire da un'altra ? Una cella u è raggiungibile da una cella v se e solo se u risiede nel rombo di v.
Alla base di questa considerazione si ha che le aree grige e viola sono quelle sulla quale si può avere una race condition (da ora in poi indicate come zone o aree condivise). L'area gialla invece è safe, per cui a queste celle non sarà associata una mutex.
Per garantire un corretto funzionamento delle funzioni “rule” sarà sufficiente garantire che l'accesso a tutto il rombo sia safe da race condition. Ogni Wi si occuperà di muovere tutti quegli animali che stanno al centro di un rombo il cui centro risiede nella propria sotto matrice. Per cui, qualora si voglia agire su un elemento in posizione (i,j), poiché questo potrebbe dipendere da zone condivise, è necessario prendere in considerazione il rombo con

centro (i,j), valutare quali celle del rombo hanno una mutex associata (e quindi sono in aree condivise) e fare una lock su ogni cella che appartiene al rombo.
Suddetta lock è necessaria eseguirla sempre, anche prima di guardare il contenuto della cella (i,j) (che quindi potrebbe esser non movibile o acqua) e su tutto il rombo, anche se alcune celle non sono accedi-bili dall'animale, per il seguente motivo: nel momento in cui si effettua una lock su un rombo, la si fa sulle celle che hanno una mutex associata; poiché solo le celle in aree condivise hanno una mutex associata, allora, si effettua la lock solo su celle condivise; poiché un animale potrebbe muoversi o comunque raggiungere tutto il rombo, è necessario effettuare la lock su tutto il rombo prima di guardarne il contenuto, altrimenti un'altro thread potrebbe modificare il valore delle celle (e quindi muovere gli animali) nel breve periodo che intercorre tra l'osservazione del rombo e l'acquisizione dell'insieme di lock associate; per lo stesso motivo è necessario effettuare la lock sulla cella che si vuole osservare, prima di guardarne il valore.

7 – Dead lock freedom

Una prima osservazione che potrebbe esser fatta sull'operazione di lock appena discussa è la seguente: è possibile effettuare la lock su una cella e solo successivamente effettuare il lock sull'intero rombo in caso di necessità ? No. Neppure effettuare il lock solo sulla cella di destinazione. Infatti supponiamo che osservando una cella (i,j) si trovi un animale. Allora un Wk, proverà ad acquisire la lock su tutto il rombo circostante, però, qualora si avesse un Wj che volendo operare sulla cella (i+1,j) ha effettuato il lock su di essa, Wk entrerebbe in attesa attiva che si liberi la lock sulla cella (i+1,j). Se sfortunatamente, anche Wj volesse acquisire il rombo, i due thread entrerebbero in deadlock. Questo problema deriva dal fatto che si perde un'ordine nell'acquisizione dei lock.
Poniamoci invece nella situazione descritta dalla precedente sezione: ogni Wi prima di muovere una qualunque cella, effettuerà la lock sull'intero rombo. Inoltre, come osservabile dall'implementazione, le lock sono aquisite sempre linearmente dall'alto verso il basso. La situazione descritta all'inizio di questa sezione, non costituirebbe più un problema, poiché Wk avrà acquisito tutti i lock prima di operare, mentre Wj prima di agire sulla propria cella avrebbe dovuto aquisire (i,j) che però ha acquisito prima Wk. Inoltre ogni altra cella precedentemente occupata da Wj non è utile a Wk a causa della forma a rombo di interesse per muovere la cella (i,j). Alla luce di ciò Wj entrerà in attesa attiva, fino a che Wk non ha terminato di applicare le due rule.
E riguardo i filosofi? Un rischio di questa politica è quello che ogni thread si metta in attesa di ogni altro causando la deadlock di tutti i thread. Tuttavia con una dimensione della sotto matrice minima (almeno 3x3) questa situazione è impossibile, poiché ogni Wi durante la propria esecuzione, potrebbe causare l'attesa di 3 altri Wj,k,m ovvero:

- Wi vuole operare sul proprio angolo in basso a dx (o situazione analoga)
- Wj che opera sulla matrice a destra di quella di Wi vuole modificare la cella in basso a sx.
- Wk che opera sulla matrice sotto a quella di Wi vuole modificare la cella in altro a dx.
- Wm che opera sotto Wj vuole agire sulla cella in alto a sx.

Questa appena illustrata è la peggire ipotesi che potrebbe avvenire, tuttavia, è evidente che, sotto l'assunzione che la lock venga acquisita con lo stesso ordine da tutti, almeno una di esse può terminare l'operazione di acquisizione della lock dell'intero rombo (assumendo una dimensione minima, un rombo non può sporgere da tutti i lati della sotto matrice).

8 – Movimenti multipli

Al fine di evitare il muoversi più volte di uno stesso animale, è stato introdotto uno stato nel tipo di cella reale. In particolare, questo stato può assumere tre valori, che indicano se l'animale si può muovere o meno. Quando un Wi osserva una cella in un'area condivisa (anche per ogni altra cella, ma non è significativo), viene testato lo stato della cella, e qualora esso indichi che la cella non è movibile, non vengono applicate le rule. Se invece la cella ha uno stato che permette l'applicazione delle rule, allora esse vengono applicate e lo stato modificato.
Questo meccanismo permette a sotto matrici che condividono un bordo di impedire a Wi di muovere più volte lo stesso animale. Al termine dell'update sarà compito del C di ripristinare lo stato di ogni cella, quando nessun W* può agirvi.
Per sgravare il carico sul C una politica più specifica è seguita: ogni Wi dopo aver aggiornato una sotto matrice, la ri-scandisce tutta, e ripristina lo stato delle celle nell'area gialla. Se invece sono stati cambiati stati di celle in un area condivisa, allora, la cella viene inserita in una coda, che consumerà poi C.

9 – Segnali

La gestione dei segnali è delegata a M come indicato nella sezione 3. Un segnale attiva un heandler che setta delle apposite variabili che verranno poi testate nel ciclo degli eventi da M. In particolare solo tra un update ed un altro. Questa scelta garantisce che la modifica della matrice sia in qualche modo atomica rispetto all'arrivo dei segnali.

10 – Protocollo su socket

Per minimizzare la quantità di byte trasmessi tramite socket, viene usato un protocollo su domanda unidirezionale, dal processo wator a V. In particolare, dopo l'apertura della connessione, viene trasmesso un codice che rappresenta un comando. In caso questo comando sia “exit”, allora V chiude la socket e termina. Se il comando è “quit” viene chiusa la connessione. Se il comando è “init” viene trasmessa la dimensione della matrice e V inizializza opportuni buffer. Se è “show” viene ricevuta la matrice e visualizzata.
Una tipica comunicazione potrebbe esser “init” → <“show”, matrice > → “quit”. Per comprimere ulteriormente la quantità di byte, questo modello viene rappresentato da un comando ausiliario che assume che la “init” sia stata fatta precedentemente. Per questo motivo, V terrà traccia dell'ultima “init” fatta.
Tutta via è evidente che la maggior quantità di byte trasmessa sulla socket sia, non tanto dovuta ai comandi, ma alla matrice. Quindi anziché inviare sulla socket la matrice come caratteri, ne viene inviata una sua immagine compressa.

Algoritmo di compressione : ogni cella può avere tre stati, per cui sono sufficiente 2 bit per rappresentare lo stato di una cella. Un primo approccio è di comprime la matrice da sequenza di caratteri, a sequenza di coppie di bit. In questo modo la quantità di byte trasmessi si riduce a circa un quarto. Sfruttando il quarto stato rappresentabile, verrà trasmesso uno stato “jolly” che ha il significato di precedere, non uno stato ma un numero che indica quante volte ripetere il carattere precedente al “jolly”. Quindi il “jolly” è un moltiplicatore ma il fattore che segue è comunque rappresentato con due bit. Osservando i possibili utilizzi del moltiplicatore, si può assumere che esso venga usato solo se il carattere che precede il jolly sia da ripetere almeno 4 volte (infatti, per un numero di ripetizione inferiore a 4, conviene riscrivere esplicitamente il carattere più volte), per cui il numero che segue il jolly va considerato incrementato di 4.
Esempi, dove C è un generico carattere rappresentato con 2 bit e la coppia di bit jolly è R:

- C → C
- CC → CC
- CCC → CCC
- CCCC → CR0
- CCCCC → CR1
- ecc....

Questo algoritmo può essere esteso concatenando R, ma per semplicità è implementato con una sola moltiplicazione.
Quindi inviare una matrice significherà inviare un intero che rappresenta la lunghezza della sequenza e l'immagine della matrice, ovvero una sequenza di caratteri, che verranno letti bit a bit.
Nel caso peggiore questo algoritmo genera una sequenza lunga ¼ quella originale, ovvero quando non si hanno ripetizioni più lunghe di 3. Tuttavia, spesso si ha una matrice in cui si hanno lunghe sequenze di uno stesso animale o di acqua, per cui il numero di byte trasmessi è inferiore a quello sopra esposto.

11 – Opzioni di compilazione.

È possibile inserire dei flag durante la compilazione per ottenere degli specifici comportamenti:

- -D_ONLY_ONE_ : compila wator.first.c per avere un comportamento conforme a quanto richiesto per il primo frammento. Infatti, per adattare il codice ai frammenti successivi, le modifiche ai contatori nf e ns sono state rimosse poiché la modifica era concorrente.
- -D_DEBUG_ : compila abilitando i log di debug. Durante l'esecuzione saranno disponibili molti dati, tra cui l'evolversi della comunicazione tra le varie componenti. Inoltre tra un update e l'altro sarà necessario premere invio. (*)
- -DCOLOR_DEBUG : compila abilitando i colori durante l'output: la matrice sarà colorata.
- _DO_NOT_UPDATE_ : compila con un comportamento conforme a quello del secondo frammento: non viene aggiornata la matrice. (*)
- -D_SUB_MATRIX_DEBUG_ : compila abilitando un output visuale della

suddivisione di planet in sotto matrici (usare anche _DEBUG_). (*) (**)
- -D_DEBUG_COMPRESS_ : compila dando un output visuale della compressione del pianeta. (*)

ATTENZIONE:
(*) Poiché queste funzioni sono di debug (lasciate nel codice solo per eventualmente esporre o giustificare un “evento” durante la relazione) non sono state implementate allo scopo di esser usate in una versione definitiva. In particolare, non viene fatta muta esclusione durante l'esecusione della printf che non essendo atomica, porta ad avere righe di debug mischiate.
(**) Si suggerisce di usarlo con nwork = 1.