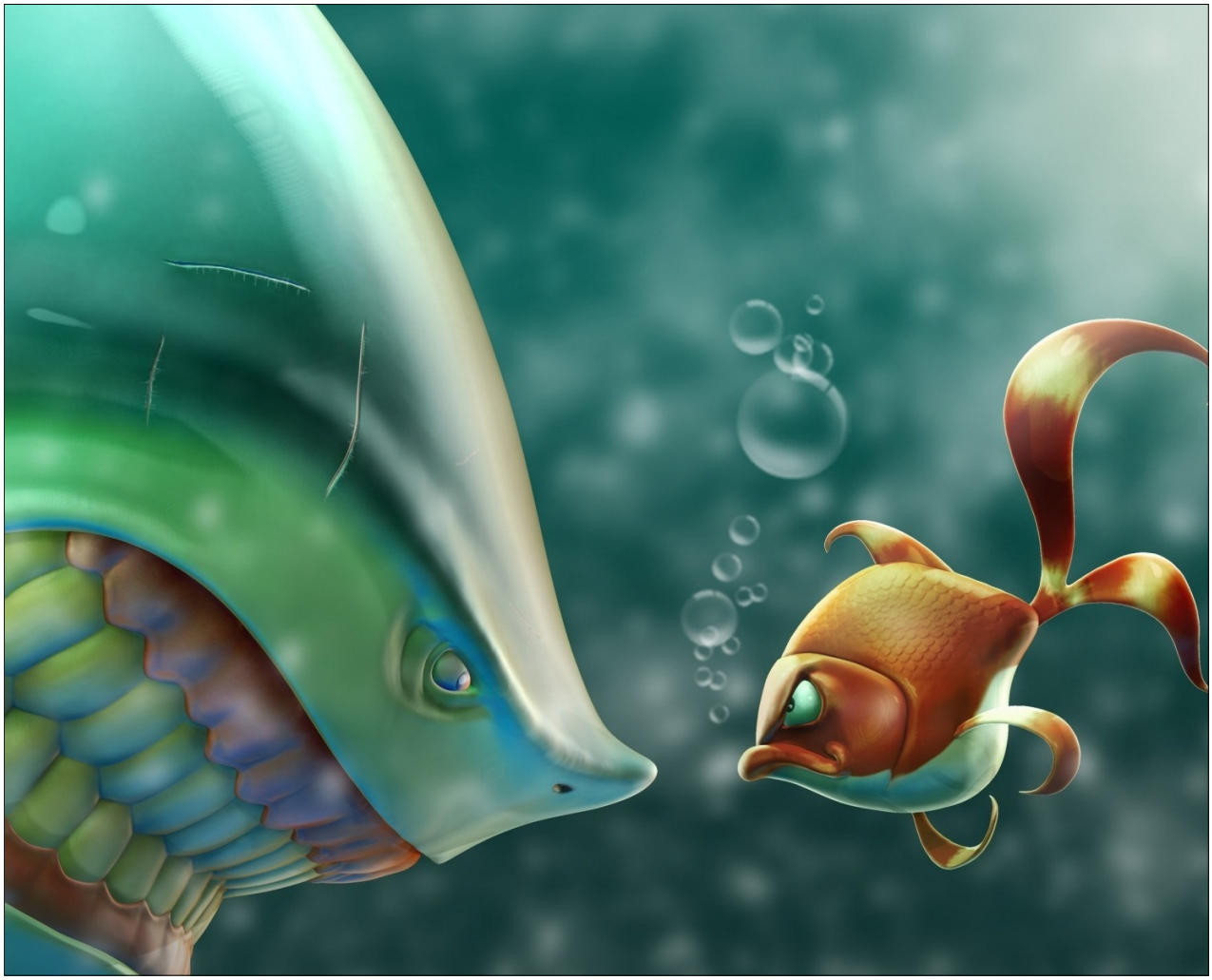


Progetto Wator

Preparato da Bachmann Gherhard
Matricola: 493593
Corso: B

CONTENUTI E COMPONENTI

Contenuto del progetto

Il progetto e' realizzato in 3 frammenti mediante i seguenti file:

- wator.h : libreria che comprende le funzioni del primo frammento.
- wator.c : implementazioni delle funzioni del primo frammento. La compilazione ed esecuzione di questo programma si trova sotto nella sezione compilazione ed esecuzione.
- processowator.c : implementazione del secondo frammento, contiene il main e le funzioni dei thread dispatcher, worker e collector. Con in più la specifica di comunicazione attraverso socket tra il processo wator e il processo visualizer.
- fram3.c : implementazione del terzo frammento. Contiene la sincronizzazione tra i thread principale, la mutua esclusione tra i vari worker e la comunicazione tra wator e visualizer.

Componenti

Il progetto e' composto dalle seguenti componenti:

- Processo principale “wator” che a sua volta e' composto da altre componenti. Il thread principale “main” il thread “dispatcher” , il thread “collector” e i vari thread “worker”
- Processo visualizer.

Il processo principale appena parte prepara l'ambiente. Crea la socket per la comunicazione con il processo visualizer dopodiché crea il processo visualizer, e prepara tutto (mutex e variabili di condizionamento) per la creazione dei thread. Ognuno dei thread appena creato esegue dei controlli di parametri dopodiché si mette in un ciclo infinito. Così facendo i thread continuano a lavorare finche non

finisce tutto il programma, cosa che accade soltanto quando viene ricevuto un SIGINT o un SIGTERM.

Il processo visualizer viene creato dal processo wator. Alla sua esecuzione il processo visualizer prepara la comunicazione con il processo wator. Non ha componenti secondarie e il suo lavoro da fare e' poco. Deve soltanto aspettare che il wator li mandi dati attraverso il canale di comunicazione (socket).

SINCRONIZZAZIONE

Sincronizzazione

La sincronizzazione tra i thread e' abbastanza semplice. Studiamo i casi della sincronizzazione tra i thread principali e la sincronizzazione tra i vari worker. Per sincronizzare i thread principali, dispatcher collector e worker, e' stata creata una variabile state di tipo "enum". La variabile ha tre stati: state_d, state_w e state_c rispettivamente dispatcher, worker e collector. La variabile e' stata inizializzata a state_d perché in ogni caso il thread worker o thread collector sarebbe andato in attesa del thread dispatcher. Partendo dal thread dispatcher, che fornisce i lavori, si fa subito una mutex_lock in modo che nessuno può agire sulla coda mentre lui ci lavora sopra. Dopo che tutti i lavori (coordinate) sono state inserite nella lista il thread dispatcher mette la variabile state a state_w e manda un broadcast a tutti worker che stavano aspettando sulla variabile di condizionamento "list". Dopodiche rilascia la mutex e si mette in attesa sulla variabile stato, e rimane li finche non viene risvegliato dal thread collector.

Passando al prossimo collegamento di thread, abbiamo i worker e il collector. I worker sono tanti quindi per far lavorare il thread collector nel momento giusto ho usato delle variabili globali. Precisamente "lavorid" e "lavoriw" che sono state definite volatile sigatomic per essere thread safe. In questo modo il thread dispatcher aumenta di 1 la variabile "lavorid" tutte le volte che inserisce nella lista. Mentre "lavoriw" viene incrementata di 1 da ogni thread worker tutte le volte che viene svolto un lavoro. Attenzione che in questo caso "lavoriw" e' in mutex perché altrimenti due worker possono incrementarla insieme ottenendo un risultato non desiderato.

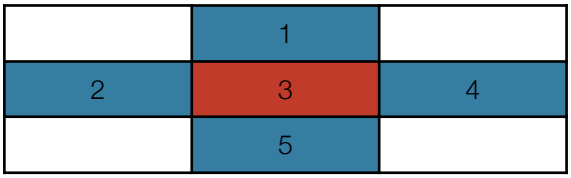
Appena le due variabili sono uguali significa che i lavori da eseguire sono finiti e quindi può partire il thread collector. Quindi viene messa a state_c la variabile state e viene azzerata la variabile “lavoriw”. In più il thread worker manda al thread collector un segnale sulla variabile di condizionamento “workersdone”, dopodiché rilascia la mutex.

Il collegamento tra collector e dispatcher e’ il più semplice dei tre perche tutto quello che succede e’ il collector che fa il suo lavoro e in fondo imposta la variabile state a state_d e manda un segnale al thread dispatcher sulla variabile di condizionamento “coll”. L’ultima parte e’ ovviamente tra una mutex_lock e una mutex_unlock.

In conclusione per sincronizzare i tre thread tra di loro in modo che ci sia un ordine e che se qualcuno parte per primo verifichi se deve prima aspettare qualcun altro oppure no, e’ stata usata una sola variabile mutex e tre variabili di condizionamento.

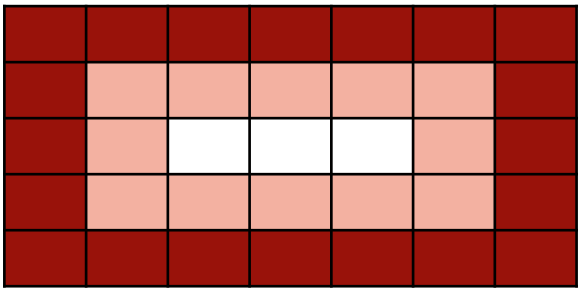
Vediamo ora il secondo caso di sincronizzazione di thread. I vari thread worker tra di loro. Si inizia sapendo che ogni worker prende una sua matrice dalla lista. Su questa matrice lui deve svolgere i suoi lavori, cioè applicare la funzione new_update (molto simile alla update_wator del primo frammento). Il problema e’ che ogni worker ha una sottomatrice della matrice grande ed e’ possibile che ci siano regioni di matrice che vengono toccate da più worker nello stesso momento. Per evitare cio’ e’ stata creata una matrice di variabili mutex.

La matrice di mutex permette di usare un approccio simile al problema dei filosofi a cena. Una cella della matrice non viene controllata affinché non ha accesso alle celle che servono prima. Nella prossima figura viene illustrato un esempio ci controllo per una cella.



La cella rossa e’ la cella selezionata e le celle blu sono quelle di cui si ha bisogno per

poter svolgere il controllo. Ovviamente questo non succede per ogni cella di ogni sottomatrice. A questo punto e' utile illustrare le regioni critiche di una sottomatrice.



Le celle rosse sono la regione critica, e richiedono una mutex_lock delle 4 celle intorno più la cella stessa. Mentre le celle rosa sono meno critiche ma con possibile concorrenza tra worker. Per loro le mutex richieste sono meno di 5 ma per semplicità ho usato le stesse lock che ho usato pure per le celle rosse. Infine le celle bianche non hanno bisogno di alcuna mutex sono celle sicure. In pratica sono le prime due righe, prime due colonne ultime due righe e ultime due colonne di ciascuna sottamatrice. Le celle blu (della prima figura) vengono bloccate in un certo ordine. Per prima quella sopra, per seconda quella a sinistra e per terza la cella stessa poi per quarta la cella a destra e per ultima blocca la cella sotto. Facendo cosi eliminiamo “quasi” la possibilita di deadlock. Un rischio di usare questo metodo e' che se per esempio ogni thread si mette in attesa di qualcun altro si ottiene un attesa di tutti thread creando poi un deadlock. Nel nostro caso in cui le matrici sono abbastanza grandi questo fatto non succede mai.

La sincronizzazione tra i due processi. Il processo wator e il processo visualizer si parlano tra di loro attraverso una socket. Il processo visualizer appena va in esecuzione si prepara per la lettura da socket e dopo che e' avvenuta la connessione si mette in attesa di ricevere dati. I dati passati sulla socket sono le “nrow” “ncol” e le celle della matrice. Dati che i visualizer sovra stampare su file o a video appena riceve un certo messaggio. Di questo messaggio parleremo nella sezione del protocollo socket.

SOCKET E SEGNALI

Gestione Segnali

La gestione dei segnali e' nel processo wator. I segnali SIGTERM , SIGUSR1 e SIGINT vengono gestiti con un thread “signal_waiter”. E' un semplice handler che si mette in attesa di questi segnali e appena ne riceve uno svolge la sua piccola parte e rida il controllo agli altri thread. Il processo visualizer invece non contiene segnali.

Socket

Sul canale di comunicazione tra wator e visualizer vengono mandati i dati in questo ordine.

1) Un carattere di operazione, che serve per la terminazione gentile.
2) Il numero di righe nrow
3) Il numero di colonne ncol
4) Una ad una le celle della matrice

TERMINAZIONE GENTILE

Il nostro programma finisce soltanto quando riceve uno dei due segnali SIGTERM o SIGINT. Nel thread signal_waiter (handler del segnale) appena uno dei due segnali viene ricevuto, la variabile globale “fine” viene impostata a 1. Con fine uguale a 1 i thread in esecuzione o in attesa capiscono che e' la loro ultima iterazione.
Analizziamo il comportamento dei thread uno ad uno quando viene ricevuto uno dei due segnali. Il thread dispatcher si trova in un ciclo do-while(!fine) quindi se il segnale viene ricevuto lui termina semplicemente l'iterazione e poi esce. I thread worker sono un po più complicati perché non possono finire subito. I thread worker nel caso in cui viene ricevuto il segnale ma la lista dei lavori da fare non e' vuota, allora loro devono finire tutti i lavori e poi possono uscire.

Per questo ogni thread e' stato messo in un ciclo while(!fine) ma subito dopo viene controllata la lista dei lavori. Se la lista non e' vuota i worker continuano a lavorare e quindi a prendere coordinate dalla lista fino a renderla vuota. Una volta finito anche i thread worker possono uscire.

Il thread collector come anche gli altri e' in un ciclo con la guardia sulla variabile fine. Il lavoro di questo thread e' semplice, deve mandare ogni nchron il pianta al processo visualizer. In caso invece di terminazione il thread collector manda a prescindere dal chron, la matrice a processo visualizer facendo un semplice "if" prima delle "write" sulla socket. Dopodiche esce anche lui dai ciclo come tutti i thread.

Il processo visualizer invece finisce grazie al collector, che manda sulla socket un messaggio "E" (sta per exit). Il messaggio viene ricevuto dal visualizer, che prima finisce il suo lavoro di lettura dalla socket, poi stampa il pianeta a video o su file, e infine imposta la variabile fine a 1 così puo' finire pure lui.

Ovviamente in caso di terminazione, prima di chiudere i processi, vengono fatte le varie free e close delle strutture necessarie.

ERRORI BUG E COMPILAZIONE

Bug

Non ho riscontrato BUG noti

Errori

Ci possono essere vari errori nel programma e questa e' la loro gestione:

- Errore di parametro passato, fa finire immediatamente il programma.
- Errore di file malformati oppure matrici che non rispettano il formato, sono gestiti dal file del primo frammento e pure in questo caso il programma finisce immediatamente.

- Errore di connessione alla socket, nel caso del wator il programma finisce subito, mentre nel caso di visualizer il programma fa un iterazione dei thread del wator dopodiché finisce con connessione fallita.
- Altri errori tipo allocazione fallita oppure la creazione dei thread fallita fanno finire il programma subito.

Compilazione

Un semplice esempio di compilazione puo' essere: gcc -Wall -pedantic -o wator wator.c -lpthread.

Tutte le compilazioni di tutti i frammenti si trovano nel file Makefile.

Conclusioni

Non sono state incontrate difficoltà particolari nel svolgere il progetto.

Si può senza dubbio migliorare l'efficienza del progetto migliorando la sincronizzazione dei thread e la sincronizzazione dei processi.

Bachmann Gherhard

Matricola : 493593

Corso : B

Progetto di Sistemi Operativi