PROCESS-CHAIN

Il progetto, come scritto da consegna, consiste nel far gestire ai processi del computer un sistema di transazioni di denaro attraverso un “libro mastro” virtuale.

COMPILAZIONE

Per compilare il programma si è scelto di utilizzare un makefile, cioè uno script, che collega tra di loro le dipendenze dei vari file, li compila, li esegue ed in fine elimina i file oggetto creati per l’esecuzione. L’intero programma inizia dall’esecuzione del Master (file “Master.c”).

INIZIALIZZAZIONE

Si è deciso di utilizzare cinque **memorie condivise** per il funzionamento del codice.
Queste aree serviranno per consentire l’accesso (in scrittura e/o in lettura) dei dati da parte dei diversi processi.
Ognuna di esse viene allocata all’inizio con una dimensione ben definita, queste sono:

- La memoria condivisa del **libro mastro** dove verranno salvate le transazioni effettuate dai processi utenti. Tutti possono accederci in lettura o in scrittura e non è previsto l’accesso in mutua esclusione per quest’area.
- La memoria condivisa per l’**indice del libro mastro** sarà l’indice che indicherà la riga del libro mastro alla quale i processi nodo sono arrivati a scrivere. Quest’area sarà acceduta in mutua esclusione dai nodi per la scrittura così da non poter permettere a più processi di modificare l’indice contemporaneamente. La lettura servirà anche per stabilire quando l’indice raggiungerà il valore di SO_REGISTRY_SIZE.

Come scelta progettuale è stato deciso, per le tre aree condivise successive, di utilizzare la posizione zero della memoria condivisa per contenere il numero di utenti e di nodi creati e le restanti celle per contenere i dati di ognuno di questi.
I seguenti frammenti di memoria condivisa verranno popolati direttamente dal master:

- La memoria condivisa contenente la **lista dei nodi** servirà per consentire la scelta del nodo, e/o quello che nella traccia viene definito “amico del nodo”, a cui inviare transazioni.
- La memoria condivisa contenente la **lista degli utenti** servirà al master per essere a conoscenza dello stato dei figli e a questi per poter sapere a chi è possibile inviare ulteriori transazioni. In questo caso i processi nodo non si attaccheranno a questo frammento di memoria perché per loro non è necessario conoscere le informazioni riguardanti gli utenti.
- La memoria condivisa contenente la **lista dei nodi amici** servirà per indicare quali saranno gli eventuali “amici” del nodo i-esimo, così da consentirne le eventuali transazioni.

Per la comunicazione delle transazioni dai processi utente ai processi nodo verrà utilizzata una **coda di messaggi.**
L'ID delle code di messaggi sarà inserito all'interno della memoria condivisa contenente la lista dei nodi, quindi ogni coda di messaggi corrisponderà ad un nodo.
Abbiamo deciso di riservare anche una coda di messaggi per il processo master che, nella versione intera, riceverà le transazioni che non sono riuscite ad essere gestite dai processi nodo dopo SO_HOPS volte.

Per la sincronizzazione del programma utilizziamo quattro set di **semafori**, creati tutti nel Master. Uno è di tipo binario, o "mutex", quindi che consentono ai processi di sincronizzarsi per poter lavorare in mutua esclusione evitando che accedano contemporaneamente alla stessa area di memoria condivisa. I restanti sono contati, quindi consentono a n-processi contemporaneamente di accedere ad un area di memoria condivisa, ma non più di n.
Questi sono:

- **Semaforo per il risveglio di tutti i processi**, verrà utilizzato con la semop a zero per attendere che da SO_USERS_NUM e SO_NODES_NUM arrivi al valore di zero, quindi ogni processo creato decrementerà di uno il semaforo, fino a quel momento nessun processo inizierà a lavorare. Questo è un semaforo di tipo contato. (Prima creo i processi nodo, dopo i processi utente).
- **Semaforo per l'accesso alla coda di messaggi**, servirà per garantire che per ogni coda di messaggi che corrisponderà ad un nodo non vengano scritte più di SO_BLOCK_SIZE-1 transazioni, che verranno trasferite alla transation pool locale del nodo per essere processato. In questo caso la coda di messaggi è il tramite tra i processi utente e i processi nodo. Il semaforo viene decrementato ogni transazione ricevuta dal nodo. Anche questo è un semaforo di tipo contato.
- **Semaforo per l'accesso all'indice del libro mastro**, questo servirà per fare in modo che un solo nodo per volta incrementi il valore dell'indice della riga del libro mastro alla quale il nodo i-esimo è arrivato a scrivere. Questo è un semaforo di tipo "mutex".
- **Semaforo per la creazione di nuovi nodi amici**, sarà utile quando avremo una transazione che dopo SO_HOPS volte non avrà trovato collocazione, a quel punto si dovrà creare un nuovo nodo per gestirla, considerando che il computer ha risorse limitate e con numeri molto distanti potrebbe eccedere nel numero di fork() da eseguire, si è deciso di creare un semaforo che ad ogni creazione di un nodo amico decrementa di uno il semaforo fino a raggiungere il limite impostato, dopo il quale non creerà più ulteriori nodi e gli utenti la normale esecuzione di SO_RETRY.

Le **variabili locali** utilizzate servono solamente al processo che sta eseguendo, non possono essere utilizzate esternamente.

Per **l'assegnazione dei valori alle variabili date dalla traccia** è stato scelto di usare una funzione che "parsifica" le stringhe riconoscendo i token come i nomi delle variabili, il delimitatore, che sarebbe il carattere "=", e il numero da assegnargli, che verrà convertito con la funzione atoi() da carattere ad intero. Questi campi devono essere riempiti a tempo di esecuzione, quindi dopo che il programma è stato avviato tramite un input, come un file, variabili d'ambiente, stdin (input da tastiera), etc…
I parametri: SO_BLOCK_SIZE e SO_TP_SIZE, invece, verranno letti a tempo di compilazione, questo significa che il compilatore conoscerà e sostituirà il valore di quelle variabili all'interno del programma prima della sua esecuzione.

CREAZIONE FIGLI

Per la **creazione dei processi figli**, il master, si servirà di due cicli "for", uno per i figli nodo ed uno per i figli utenti, dove eseguirà una fork() ad ogni passo, ma i processi saranno bloccati dal semaforo per la loro sincronizzazione così da consentirgli di iniziare ad eseguire nello stesso momento. Da qui partirà anche il conteggio del tempo per la simulazione che verrà, eventualmente, interrotto dalla funzione di alarm(), che verrà gestita dall'handler che sarà impostato a scattare alla ricezione del segnale di SIGALARM.
Nella versione completa abbiamo anche la possibilità che i figli nodo vengano creati dopo SO_HOPS volte che una transazione non è riuscita ad essere gestita, in questo caso abbiamo un controllo sulla msgrcv() della coda di messaggi del master con IPC_NOWAIT, che quindi rende la ricezione non bloccante, che effettua una fork() se trova un messaggio al suo interno.

CICLO DI VITA DEL MASTER

Il ciclo vita del processo master è contenuto all'interno di un ciclo while, dove ogni secondo:

- controlla che l'indice del libro mastro (contenuto in memoria condivisa) non abbia raggiunto la dimensione massima del libro mastro, in tal caso con la funzione raise(SIGALARM) fa scattare l'handler che terminerà la simulazione;
- stampa la situazione dei figli (budget, figli vivi etc…);
- con la funzione waitpid() verifica la terminazione dei figli decrementando il contatore locale sino ad arrivare ad avere un solo processo utente, a questo punto il master farà scattare l'handler, sempre con la funzione raise(SIGALARM), per terminare la simulazione;
- stampa il risultato finale del libro mastro con eventuale motivo di terminazione;
- dealloca tutte le memorie condivise, code di messaggi e semafori istanziati all'inizio e termina;

- controlla di non aver ricevuto transazioni nella propria coda di messaggi, in caso contrario creerà un nuovo nodo che gestirà questa transazione e verrà aggiunto alla lista nodi, con gli eventuali amici.

Per mantenere il master nel suo ciclo è stata usata una variabile “master” con assegnato il valore MASTER_CONTINUE = 1 e MASTER_STOP = 0.

TRANSAZIONI

Le transazioni saranno gestite con una struct chiamata, appunto, “trasazione”, che conterrà i campi richiesti. Ogni struct verrà inviata dagli utenti alla coda del messaggio corrispondente al nodo interessato, che la raccoglierà e la inserirà nella propria transaction pool locale, decrementando il semaforo per il conteggio del suo riempimento di uno. Quando il semaforo della i-esima coda di messaggi impostato con la NOWAIT (che rende non bloccante il semaforo), raggiungerà il valore di zero il processo che cercherà di inviare una transazione riservando il semaforo riceverà un errore di EAGAIN, provando quindi ad inviare la transazione ad un altro nodo.
Quando il nodo riuscirà a riempire un blocco di transazioni le scriverà sul libro mastro liberando il semaforo di SO_BLOCK_SIZE – 1 (meno uno perché considero la transazione di reward).

FUNZIONI SORTEGGIO RANDOM

All’interno delle funzioni per il sorteggio di valori casuali, per la funzione srand(), è stato utilizzato il valore del tempo attuale in nanosecondi (che siccome restituisce un long viene convertito in int scartando le cifre più significative) perché, per un migliore funzionamento, è necessario un valore, che assume la funzione di “seme”, che cambi ad ogni chiamata.

CICLO VITA PROCESSI UTENTE

Il processo utente tramite una execpl riceve i parametri necessari dal processo mater.
Come per il master, anche il processo utente userà un ciclo while con una variabile “utente” impostata a UTENTE_CONTINUE (che equivale al valore 1), all’interno del quale farà:

- il calcolo del proprio budget dove controllerà di aver ricevuto transazioni e nel caso incrementandole ad esso;
- l’estrazione casuale di un altro processo utente che riceverà la transazione;
- l’estrazione casuale di una coda di messaggi appartenente al nodo che dovrà processare la transazione;
- decrementerà il semaforo per l’accesso alla coda di messaggi;
- grazie ad una funzione random sorteggerà un valore per la quantità e compilerà la struct che verrà poi spedita tramite una msgsnd() alla coda di messaggi estratta;

- in caso di fallimento per mancanza di budget, dopo SO_RETRY volte, la variabile del master viene convertita in UTENTE_STOP (che equivale a zero) che causerà l'uscita dal ciclo di vita con conseguente cambiamento del proprio stato nella memoria condivisa contenente la lista degli utenti in USER_KO e la deallocazione delle risorse condivise utilizzate;
- in caso di fallimento perché trova la transaction pool piena prova ad inviare la transazione ad un nodo amico al nodo scelto e dopo SO_HOPS volte la invia al master che creerà un nuovo processo nodo per gestire la transazione e quelle a venire.

CICLO DI VITA DEL NODO

Come per il processo utente, anche il processo nodo tramite una execpl riceve i parametri necessari dal processo mater.
Identicamente ai processi utenti e al processo master, anche il processo nodo userà un ciclo while con una variabile "nodo" impostata a NODO_CONTINUE (che equivale al valore 1), all'interno del quale farà:

- la ricezione dei messaggi non bloccante che prende ogni messaggio che riesce a prelevare e lo salva nella transaction pool locale, incrementando gli indici che indicano il riempimento di quest'ultima e del blocco da inviare al libro mastro;
- saltando la prima transazione ricevuta alla sua creazione, periodicamente ad ogni ciclo, invia una transazione ad un nodo amico, rispettando la regola SO_HOPS volte come per gli utenti;
- appena possibile riempie un blocco da inviare sul libro mastro sommando i reward di tutte le transazioni per l'ultima posizione del blocco;
- riservo il semaforo che controlla l'accesso alla memoria condivisa dell'indice del libro mastro, incremento l'indice e poi rilascio subito il semaforo, lasciando la scrittura allo scheduler che scriverà unicamente nella posizione i-esima "prenotata" dal nodo i-esimo;
- scrivo sul libro mastro il blocco preparato e rilascio il semaforo di SO_BLOCK_SIZE-1 per consentire agli altri processi di inviare nuove transazioni al nodo;
- aggiorno gli indici delle variabili locali e il budget del nodo che dovrà poi essere stampato;
- quando riceverà un segnale di SIGUSR1 cambierà lo stato della variabile locale "nodo" in NODE_STOP uscendo dal ciclo e deallocando le memorie condivise utilizzate.

SEGNALI

Si è deciso che il segnale per la creazione della transazione manuale sarà SIGUSR2 mentre quello per la terminazione dei processi è SIGUSR1.
Come anticipato precedentemente SIGALARM sarà il segnale di terminazione inviato da alarm() dopo SO_SIM_SEC secondi se la terminazione non sarà conclusa.

All’inizio di ogni processo viene settato l’handler del segnale desiderato al valore da attendere, quando il segnale atteso viene ricevuto dal processo viene eseguito l’handler che effettuerà le operazioni dentro definite.
Nel caso di SIGUSR2 sceglierà un utente alla quale farà effettuare un invio “straordinario” al processo indicato nel terminale, nel caso di SIGALARM concluderà la simulazione mandando un segnale di SIGUSR1 ai processi per terminarli.
SIGUSR1 verrà inviato anche nel caso di esaurimento di spazio nel libro mastro e/o nel caso della terminazione prematura da parte di tutti i processi utente.

FINE DELLA SIMULAZIONE

Una volta scattato l’handler, quindi sapendo che è avvenuto un evento di terminazione del programma, eseguiamo prima la terminazione dei processi ancora in vita che deallocheranno le risorse utilizzate e successivamente, il master, deallocherà ed eliminerà le memorie condivise e i semafori allocati inizialmente.