RELAZIONE PROGETTO UNIX

VERSIONE MINIMA

Stefano Cipolletta

matricola 948650

stefano.cipolletta@edu.unito.it

Corso A – Turno T1

A.A 2021/2022

INDICE

SCELTE IMPLEMENTATIVE

Strutture utilizzate

- Ho deciso di implementare 6 strutture fondamentali:
  - ***transaction***: contiene le informazioni di una singola transazione;
  - ***block***: contiene un intero senza segno (*size*) e un array di transazioni lungo al massimo SO_BLOCK_SIZE;
  - ***ledger***: (***Libro Mastro***) contiene un intero senza segno (*size*) e un array di blocchi lungo al massimo SO_REGISTRY_SIZE;
  - ***userProcess***: contiene le informazioni di un processo **utente** quali il PID, il bilancio e lo stato del processo (vivo/morto);
  - ***nodeProcess***: contiene le informazioni di un processo **nodo** quali il PID, il bilancio e la grandezza della *Transaction Pool;*
  - ***message***: contiene un campo *mtype* che serve per il destinatario del messaggio e un campo per la transazione da inviare.
- Ed un'enumerazione per una miglior gestione dei semafori:
  - ***Sem***: ad ogni valore testuale attribuitogli ne corrisponde uno numerico. Ad esempio *userSync* corrisponde al valore numerico *0*, *userShm* al valore *2* ecc. così facendo ogniqualvolta vado ad utilizzare una ***reserveSem*** o una ***releaseSem***, non c'è bisogno che mi ricordi il numero del semaforo (*semNum*) ma basta che inserisca l'identificativo presente nell'enumerazione.

    Esempio:

```
reserveSem(semId, userSync);
*activeUsers = 0; /* utenti attivi: per sincronizzazione */
releaseSem(semId, userSync);
```

Oggetti IPC utilizzati

- Gli oggetti IPC utilizzati sono stati i seguenti:
  - ***SEMAFORI*** (#6)
    0. *userSync*: utilizzato per accedere alla memoria condivisa di sincronizzazione degli utenti;
    1. *nodeSync*: utilizzato per accedere alla memoria condivisa di sincronizzazione dei nodi;
    2. *userShm*: utilizzato per l'accesso in mutua esclusione alla memoria condivisa ***users***;
    3. *nodeShm*: utilizzato per l'accesso in mutua esclusione alla memoria condivisa ***nodes***;
    4. *ledgerShm*: utilizzato per l'accesso in mutua esclusione alla memoria condivisa ***ledger***;
    5. *print*: utilizzato per non sovrapporre la stampa a video.

  - ***MEMORIE CONDIVISE*** (#5)
    0. *activeUsers*: per la sincronizzazione dei processi utente e il tracciamento del loro stato;
    1. *activeNodes*: per la sincronizzazione dei processi nodo;
    2. *users*: per il salvataggio di tutte le informazioni di tutti gli utenti;
    3. *nodes*: per il salvataggio di tutte le informazioni di tutti i nodi;
    4. *ledger*: (***Libro Mastro***) per il salvataggio di tutte le transazioni avvenute con successo.

  - ***CODE DI MESSAGGI*** (#2)
    1. *Message Queue*: per l'invio di transazioni dall'utente al nodo;
    2. *Response Queue*: per l'invio di transazioni non aggiunte alla *Transaction Pool* di un nodo al mittente della transazione;

CICLO DI VITA DEI PROCESSI

Master

Alcune tra le più importanti operazioni che il processo master esegue:

Creazione e Sincronizzazione dei Processi

1. Come prima cosa esegue una `fork();`
2. [1]Dopodiché `execv("./nodo.o", arg);` per i nodi e `execv("./utente.o", arg);` per gli utenti, procederanno a creare i rispettivi processi con *arg* array di argomenti;
3. Una volta creati, il master attende che tutti i processi aggiornino la memoria condivisa per la loro sincronizzazione e, fatto ciò, incrementa i corrispettivi semafori per l'accesso in memoria condivisa;

Inizio della Simulazione

4. Impostare `alarm(SO_SIM_SEC);` in modo tale da far partire il timer della simulazione;
5. Stampare ogni secondo un riepilogo delle informazioni;
6. Estrazione casuale di un numero tra 0 e 5. Se esce 0 inviare a un utente random il segnale **SIGUSR1**, per forzare la creazione di una transazione.

Utente

Alcune tra le più importanti operazioni che il processo utente esegue:

Bilancio, Transazione di Risposta e Nuova Transazione

1. Come prima cosa l'utente aggiorna il suo bilancio tramite `balanceFromLedger(getpid(), &lastVisited)` , andando a controllare nel ***Libro Mastro*** se si sono aggiunte delle nuove transazioni di cui lui ne è il destinatario;
2. Dopodiché controlla se sono presenti transazioni fallite. Se non sono presenti transazioni, *try* (variabile che tiene traccia del numero di transazioni consecutive fallite) viene resettato a 0. Altrimenti *try* viene incrementato e viene verificato se ha raggiunto `SO_RETRY`; in tal caso l'utente imposta il suo stato a *morto*, dealloca tutte le strutture IPC, notifica al master che sta per terminare (tramite **SIGUSR1**) e termina;
3. Infine, se il suo bilancio è sufficiente, si occuperà di creare ed inviare una transazione e successivamente aggiornare il bilancio andando a togliere una quantità di denaro pari alla quantità presente nella transazione da inviare sommata al reward per il nodo che la elaborerà.

Nodo

Transaction Pool

La Transaction Pool è un array di transazioni la cui dichiarazione viene fatta all'interno del file *nodo.h*.

La prima operazione che un nodo fa, ovviamente dopo aver inizializzato le variabili di configurazione, è quella di allocare la memoria necessaria per la Transaction Pool nel seguente modo:

```
pool = (transaction*)calloc(SO_TP_SIZE, sizeof(transaction));
```

In questo modo vado ad allocare uno spazio in memoria grande `SO_TP_SIZE` × `sizeof(transaction)` (numero di elementi × la grandezza di ogni elemento) che viene inizializzato a **0**.

[1] **N.B.**
Scelta implementativa per rendere il codice del progetto il quanto più possibile modulare.

Nuova Transazione, Blocco Candidato e Aggiornamento Strutture

1. Tramite una `msgrcv` controlla se sono presenti delle transazioni da aggiungere alla Transaction Pool, se sono presenti e la Transaction Pool non è piena le inserisce, altrimenti rispedisce le transazioni ai mittenti;
2. Dopodiché crea un ***Blocco Candidato*** da inviare al ***Libro Mastro*** e aspetta una quantità di tempo compresa tra `SO_MIN_TRANS_PROC_NSEC` e `SO_MAX_TRANS_PROC_NSEC`
3. Infine, se la creazione del ***Blocco*** ha avuto successo, il nodo si occuperà di aggiornare il ***Libro Mastro*** e successivamente di eliminare le transazioni del ***Blocco*** dalla Transaction Pool. Se tutte queste operazioni hanno avuto esito positivo, il ***Blocco*** viene svuotato tramite `memset(&b, 0, sizeof(block));` . Se il ***Libro Mastro*** risulta pieno, viene inviato un segnale **SIGUSR2** al processo master per terminare la simulazione.