Progetto di sistemi operativi

Cristina Andrea Giacolono	952573	cristina.giacolono@edu.unito.it
Matteo Marengo	945772	matteo.marengo44@edu.unito.it

Master

Inizializza alcune macro sfruttando il makefile leggendo le variabili d'ambiente.

Inizializza il signal handler per poter gestire i vari segnali che gli arrivano:

→ **SIGINT**: il programma termina per causa dell'utente usando la shortcut CTRL+C oppure tutti gli utenti sono morti prematuramente (i due casi sono distinti dal valore contenuto nella variabile exitReason).

→ **SIGALRM**: il programma termina per tempo scaduto.

→ **SIGUSR1**: il programma termina perché il masterbook è pieno e non può ospitare ulteriori blocchi.

→ **SIGUSR2**: il programma termina perché è stato raggiunto il numero massimo di nodi.

Inizializza le risorse IPC che serviranno per gestire i nodi, gli user e il masterbook, le risorse utilizzate sono:

→ code di messaggi:

1. Ogni nodo ha una coda di messaggi personale usata per ricevere le transazioni da processare.
2. **ID_QUEUE_PID_FRIENDS:** una coda unica per far inviare al master i pid degli amici ai nodi (il pid del receiver è impostato come message type e permette ad ogni nodo di distinguere i messaggi destinati a lui).
3. **ID_QUEUE_FRIENDS:** una coda unica per i nodi (usata anche dal master) utile per ricevere le transazioni inviate dagli amici tenendo traccia degli hops.

→ semafori:

1. Un semaforo per sincronizzare la creazione di nodi, utenti e il master.
2. Un semaforo per accedere ai valori che vengono modificati nella memoria condivisa user.
3. Un semaforo per accedere ai valori che vengono modificati nella memoria condivisa nodes.
4. Un semaforo per accedere ai valori che vengono modificati nella memoria condivisa delle informazioni sul master.
5. Un semaforo per accedere alle transazioni scritte sul masterbook.

→ memorie condivise:

1. masterbook
2. informazioni sui nodi
3. informazioni sugli utenti
4. informazioni sul masterbook

Il master crea:

→ **SO_NODES_NUM** processi nodo per gestire le transazioni inviate dagli utenti, verranno eseguiti tramite execve;

→ **SO_USERS_NUM** processi utenti che invieranno le transazioni ai nodi,verranno eseguiti tramite execve.

Durante la creazione viene inizializzata la memoria condivisa dei nodi e degli user con i loro dati:

```
typedef struct _node_struct {
    pid_t pid;
    int id_mq;
    int budget;
    int status;
    int tp_size;
} node_struct;

typedef struct _user_struct{
    pid_t pid;
    int budget;
    int status; /* 0 dead, 1 alive */
    int last_block_read;
} user_struct;
```

Il master attende che tutti i processi figli abbiano inserito i propri dati dentro la memoria condivisa e procede:

→ sorteggiando **SO_NUM_FRIENDS** amici tra i processi nodo, controllando che il pid estratto non sia uguale al pid del nodo destinatario e che non venga sorteggiato due volte lo stesso pid.

→ dopo averli sorteggiati, li invia tramite una coda di messaggi chiamata **ID_QUEUE_PID_FRIENDS**.

A questo punto inizia la routine del master:

→ controlla se ci sono messaggi inviati dai nodi tramite la coda di messaggi **ID_QUEUE_FRIENDS.**

→ Se ci sono messaggi e il numero di nodi è minore del numero di nodi massimo creabili, il master procede creando un nuovo nodo, inviandogli gli amici e tutti i messaggi ricevuti fino a quel momento tramite la sua coda di messaggi personale.

→ Infine estrae **SO_NUM_FRIENDS** nodi esistenti, escluso sé stesso, e procede inviando il nuovo nodo, come amico, a quelli estratti

→ Se il numero massimo dei nodi creabili viene raggiunto, lancia un segnale SIGUSR2 tramite kill() per terminare il programma.

→ Stampa:

1. le informazioni richieste controllando se tutti gli utenti sono morti prematuramente (in questo caso lancia un segnale SIGINT tramite kill per terminare)
2. se il numero di processi utente è maggiore del numero di processi stampabili verranno ordinati tramite insertion sort e verranno stampati i primi **MAX_PROC_PRINTABLE/2** processi utente (con budget più

basso) e gli ultimi **MAX_PROC_PRINTABLE/2** (con budget più alto).

Attenderà un secondo tramite una nanosleep per riprendere la routine.

NODO

Alloca lo spazio per le struct, si attacca alle risorse IPC presenti nel codice.
Attende che tutti gli utenti e i nodi vengano creati e inizia la sua routine.
Esegue **SO_NUM_FRIENDS** volte la msgrcv su **ID_QUEUE_PID_FRIENDS** per ricevere gli amici, salvandoli in un suo array nominato friends.
Inizia la routine controllando se sono arrivati altri amici, in caso positivo incrementa num_friends allocando nuovo spazio su friends per ospitare un amico in più e inserendo il nuovo amico.
Successivamente esegue una msgrcv sulla sua coda personale in attesa di nuovi messaggi:

→ se riceve un messaggio e la transaction pool è piena, sceglie un amico random a cui inviare la propria transazione inizializzando hops a 0, dopodiché esegue la msgsnd su **ID_QUEUE_FRIENDS**.

→ se la transaction pool non è piena aggiunge la transazione alla sua transaction pool e incrementa la propria tp_size nella memoria condivisa dei nodi.

A seguire controlla se ci sono dei messaggi ricevuti sulla coda degli amici **ID_QUEUE_FRIENDS**, nel caso in cui ci sono messaggi:

→ se la transaction pool è piena e gli hops sono minori di **SO_HOPS** sorteggia un amico random e gli invia il messaggio incrementando gli hops.

→ se la transaction pool è piena e ha raggiunto il numero massimo di hops, invierà la transazione al master che la processerà come descritto precedentemente.

→ se la transaction pool non è piena aggiunge la transazione alla sua transaction pool e incrementa la propria tp_size nella memoria condivisa dei nodi.

Se c'è almeno una transazione nella transaction pool sceglie un amico random e gli invia la prima transazione della transaction pool su **ID_QUEUE_FRIENDS**, successivamente scarta la transazione dalla transaction pool e decrementa tp_size.
Costruisce un nuovo blocco prendendo massimo **SO_BLOCK_SIZE-1** transazioni, in ordine di arrivo, dalla transaction pool e tiene traccia dei reward inviati utilizzando la variabile block_reward.
Se è stato preso il numero di transazioni necessarie procederà a creare un nuovo blocco inserendo una nuova transazione denominata transazione di reward, il cui amount sarà uguale al valore di block_reward.
Attenderà un tempo random tra **SO_MIN_TRANS_PROC_SEC** e **SO_MAX_TRANS_PROC_SEC** tramite una nanosleep, infine prende un nuovo id del blocco (sincronizzandosi con gli altri processi che usano quella variabile) e scriverà il blocco sul masterbook se il suo id è minore di **SO_REGISTRY_SIZE**, altrimenti lancia **SIGUSR1** tramite kill() per notificare al master che il masterbook è pieno.

USER

Alloca lo spazio per le struct e si attacca alle risorse IPC presenti nel codice.
Inizializza il signal handler per poter gestire i vari segnali che gli arrivano:

→ **SIGUSR1**: segnale lanciato per far inviare una nuova transazione allo user.

Attende che tutti gli utenti e i nodi vengano creati e inizia la sua routine.

Utilizzando il semaforo aggiorna last_block, che tiene traccia dell'ultimo blocco letto dal masterbook, e il suo budget. Questi vengono aggiornati leggendo i blocchi presenti nel masterbook che non sono ancora stati letti e incrementando il suo balance dell'amount della transazione ogni volta che il receiver in una transazione è lo user stesso.

Tramite un semaforo incrementa l'ultimo blocco letto e aggiorna il budget presente nella memoria condivisa degli user.

Estrae un nodo random a cui far processare la transazione, inserendo il pid come message type, genera la transazione usando la funzione build_transaction() che setta il resto delle specifiche richieste.

Dopo aver creato la transazione tenta di inviarla per un massimo di **SO_RETRY-1** volte:

→ se la msgsnd non dovesse fallire, aggiorna il suo current_balance sottraendo l'amount e il reward e aggiorna anche il budget presente nella memoria condivisa.

1. attenderà un tempo random tra SO_MIN_TRANS_GEN_SEC e SO_MAX_TRANS_GEN_SEC tramite una nanosleep prima di riprendere la sua esecuzione.

→ se la msgsnd fallisce **SO_RETRY** volte, verrà impostato il suo status a 0 per notificare al master che lo user è morto prematuramente e si eseguirà una exit(EXIT_FAILURE) per farlo terminare.

ESEGUIRE IL CODICE

Per eseguire il codice basta lanciare a terminale i seguenti comandi:

→ make master

→ make run