

PROGETTO S.O.

Alberto Pochettino
Federico Pitaccolo
a.a. 2021/22

Si intende simulare un libro mastro contenente i dati di transazioni monetarie fra diversi utenti. A tal fine sono presenti i seguenti processi:

- un processo *master* che gestisce la simulazione, la creazione degli altri processi, etc.
- SO_USERS_NUM processi *utente* che possono inviare denaro ad altri utenti attraverso una *transazione*
- SO_NODES_NUM processi *nodo* che elaborano, a pagamento, le transazioni ricevute.

SCHEMI SIMULAZIONE

LEGENDA	
★	processo master
△	processi nodo
□	processi utente
○	libro mastro

-FASE SET-UP

★ processo master;

- vengono presi i parametri di configurazione da un file
- allocate e inizializzate le variabili globali
- creati i due semafori: uno per il *master-book*, l'altro per il *friend-to-add*
- crea la message queue
- allocate le variabili globali che puntano alla memoria condivisa
- set dei signal handler necessari

-FASE PRINCIPALE

★ processo master;

- crea SO_NODES_NUM processi nodo
- crea SO_USERS_NUM processi user
- ciclicamente; (tramite SIGALRM)

stampa ogni secondo fino a fine simulazione

↳ aggiorna il bilancio dei processi user e dei processi nodo
leggendo le transazioni scritte nel libro mastro nel secondo passato

- contemporaneamente rimane attivo per gestire eventuali signal che potrebbe ricevere; SIGUSR1 → crea un nuovo processo nodo
SIGALRM → stampa della situazione ogni secondo

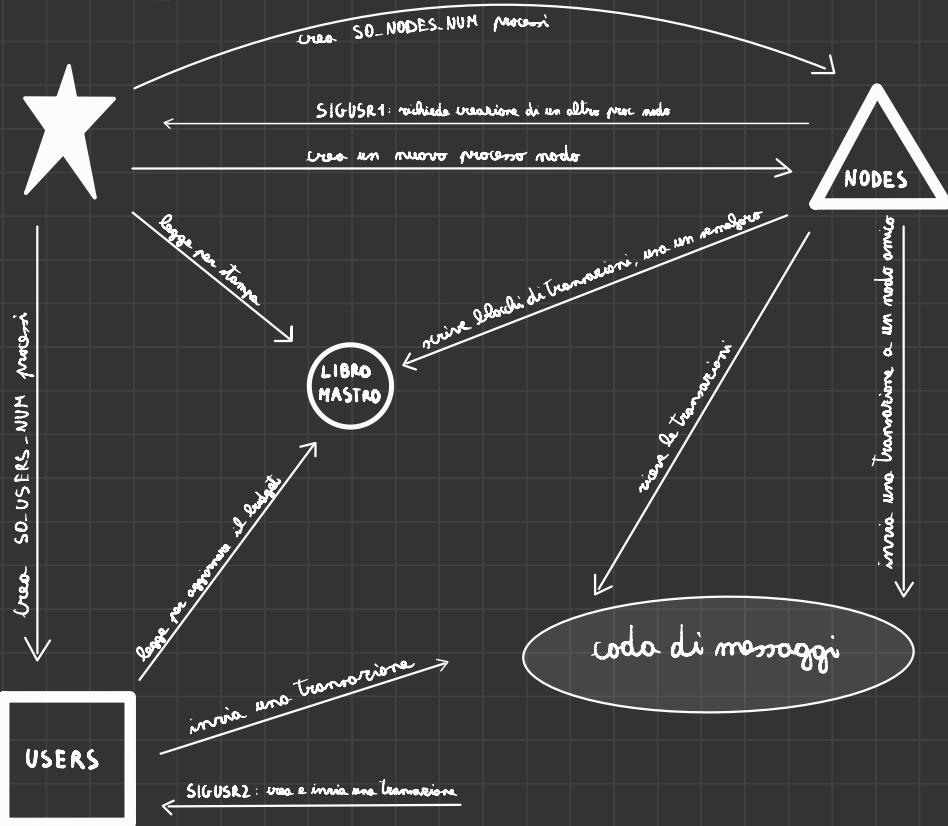
△ processi nodo;

- creiamo la lista dei friends (nel metodo wrap-node) prima di invocare il metodo node che fa partire il ciclo di vita del processo
- si inseriscono nella lista di processi nodo contrassegnando come attivi
- allocano la propria transaction pool
- ciclicamente;
 - accedono alla propria coda di messaggi per ricevere le transazioni e inviare quelle in eccesso a nodi amici
 - controllano che la transaction pool sia in regola
 - eventualmente aggiungono un amico alla propria lista
 - eventualmente inviano a un amico una transazione dalla tp
 - controllano che la transaction pool sia in regola
 - se ci sono almeno $SO_BLOCK_SIZE - 1$ transazioni nella tp scrivere un blocco di transazioni nel libro mastro, usando il semaforo
 - nanosleep
 - scrivere in shm il numero di transazioni rimanenti nella transaction pool
- deallozano la tp, e la lista di friends, si contrassegnano come non attivi e poi terminano

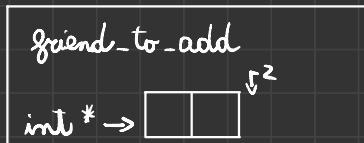
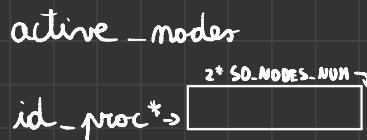
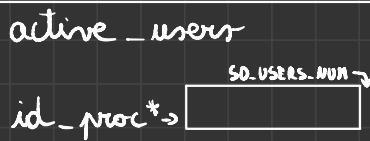
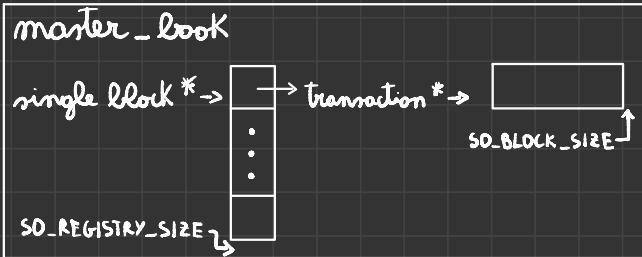
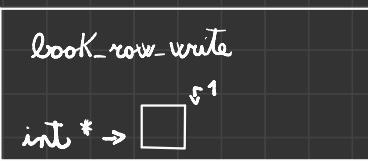
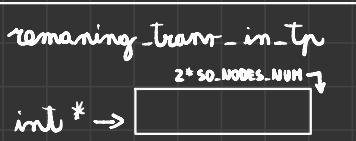
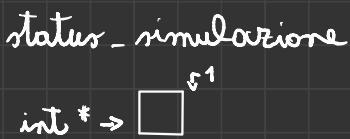
processi user;

- si inseriscono nella lista di processi user contrassegnando come attivi
- ciclicamente:
 - accedendo al libro mastro, calcolando il proprio budget a partire da quello iniziale (SO-BUDGET-INIT), quando anche l'array not-official
 - creano una nuova transazione se la transazione è valida la manda a un processo nodo e la inserisce nell'array not-official
 - nanosleep
- si contrassegnano come non attivi
 - se rimane solo un processo user setta status_rimborso()=3
 - terminano
- + gestisce il SIGUSR2 se lo riceve → crea e invia una transazione

da dove parte la freccia
è dove invia l'azione



→ SITUAZIONE DELLA SHARED MEMORY



-FASE FINALE

★ processo master;

- stampa il riassunto della situazione
- termina tutti i processi attivi rimasti
- dealloca lo spazio della memoria usato

5.1 Le transazioni

Una transazione è caratterizzata dalle seguenti informazioni:

- *timestamp* della transazione con risoluzione dei nanosecondi (si veda funzione `clock_gettime(...)`)
- *sender* (implicito, in quanto è l'utente che ha generato la transazione)
- *receiver*, utente destinatario della somma
- *quantità* di denaro inviata
- *reward*, denaro pagato dal sender al nodo che processa la transazione

```
typedef struct {
    pid_t sender;
    pid_t receiver;
    struct timespec timestamp;
    quantity;
    int reward;
} transaction;
```

La transazione è inviata dal processo utente che la genera ad uno dei processi nodo, scelto a caso.

5.2 Processi utente

I processi utente sono responsabili della creazione e invio delle transazioni monetarie ai processi nodo. Ad ogni processo utente è assegnato un budget iniziale `SO_BUDGET_INIT`. Durante il proprio ciclo di vita, un processo utente svolge iterativamente le seguenti operazioni:

1. Calcola il bilancio corrente a partire dal budget iniziale e facendo la somma algebrica delle entrate e delle uscite registrate nelle transazioni presenti nel libro mastro, sottraendo gli importi delle transazioni spedite ma non ancora registrate nel libro mastro.
 - Se il bilancio è maggiore o uguale a 2, il processo estrae a caso:
 - Un altro processo utente destinatario a cui inviare il denaro
 - Un nodo a cui inviare la transazione da processare
 - Un valore intero compreso tra 2 e il suo bilancio suddiviso in questo modo:
 - * il reward della transazione pari ad una percentuale `SO_REWARD` del valore estratto, arrotondato, con un minimo di 1,
 - * l'importo della transazione sarà uguale al valore estratto sottratto del reward
 - Esempio: l'utente ha un bilancio di 100. Estraendo casualmente un numero fra 2 e 100, estrae 50. Se `SO_REWARD` è pari al 20 (ad indicare un reward del 20%) allora con l'esecuzione della transazione l'utente trasferirà 40 all'utente destinatario, e 10 al nodo che avrà processato con successo la transazione.
 - Se il bilancio è minore di 2, allora il processo non invia alcuna transazione

2. Invia al nodo estratto la transazione e attende un intervallo di tempo (in nanosecondi) estratto casualmente tra `SO_MIN_TRANS_GEN_NSEC` e massimo `SO_MAX_TRANS_GEN_NSEC`.

Inoltre, un processo utente deve generare una transazione anche in risposta ad un segnale ricevuto (la scelta del segnale è a discrezione degli sviluppatori). → **SIGUSR2**

Se un processo non riesce ad inviare alcuna transazione per `SO_RETRY` volte **consecutive**, allora termina la sua esecuzione, notificando il processo master.



processi UTENTE

- budget iniziale `SO_BUDGET_INIT`
- se il bilancio è ≥ 2 si decide randomicamente:
 - un altro processo □ a cui inviare denaro
 - un processo Δ a cui inviare la transaz.
 - quantità soldi da inviare tra 2 e il bilancio
- se bilancio < 2 non invia nessuna transazione

↳ invio al Δ la transazione

poi faccio una nanosleep() con durata
tra `SO_MIN_TRANS_GEN_NSEC` e `SO_MAX_TRANS_GEN_NSEC`

5.3 Processi nodo

Ogni processo nodo memorizza privatamente la lista di transazioni ricevute da processare, chiamata *transaction pool*, che può contenere al massimo $S_0_TP_SIZE$ transazioni, con $S_0_TP_SIZE > S_0_BLOCK_SIZE$. Se la *transaction pool* del nodo è piena, allora ogni ulteriore transazione viene scartata e quindi non eseguita. In questo caso, il *sender* della transazione scartata deve esserne informato. → PROGETTO da 30

Le transazioni sono processate da un nodo in *blocki*. Ogni blocco contiene esattamente $S_0_BLOCK_SIZE$ transazioni da processare di cui $S_0_BLOCK_SIZE - 1$ transazioni ricevute da utenti e una transazione di pagamento per il processing (si veda sotto).

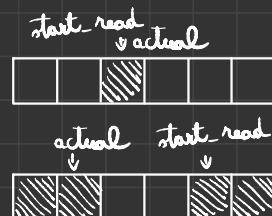
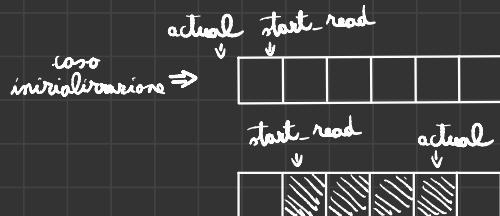
STRUTTURA DELLA TRANSACTION POOL



la transaction pool è un array che consideriamo circolare, con due indici importanti:

- *actual* è l'indice dell'ultima transazione inserita
- *start-read* è l'indice da cui partiremo per scrivere le transazioni nel libro mastro

caso esempio della Transaction pool ; = celle non vuote



Il ciclo di vita di un nodo può essere così definito:

- Creazione di un blocco candidato
 - Estrazione dalla transaction pool di un insieme di $S_0_BLOCK_SIZE - 1$ transazioni non ancora presenti nel libro mastro
 - Alle transazioni presenti nel blocco, il nodo aggiunge una transazione di reward, con le seguenti caratteristiche:
 - * *timestamp*: il valore attuale di `clock_gettime(...)`
 - * *sender*: -1 (definire una MACRO...)
 - * *receiver*: l'identificatore del nodo corrente
 - * *quantità*: la somma di tutti i reward delle transazioni incluse nel blocco
 - * *reward*: 0

- Simula l'elaborazione di un blocco attraverso una attesa non attiva di un intervallo temporale casuale espresso in nanosecondi compreso tra SO_MIN_TRANS_PROC_NSEC e SO_MAX_TRANS_PROC_NSEC.
- Una volta completata l'elaborazione del blocco, scrive il nuovo blocco appena elaborato nel libro mastro, ed elimina le transazioni eseguite con successo dal transaction pool.



processi NODO

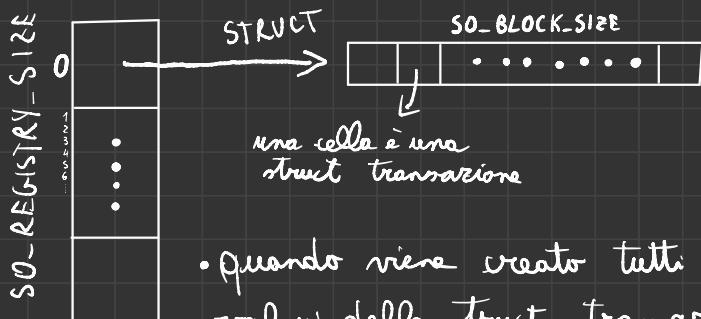
- aggiunge all'array trans_pool le nuove transazioni in entrata e invia quelle in eccesso a un nodo amico
- scrive nel libro mastro un blocco di transazioni eliminandole dallo trans_pool
- fa una nanosleep con durata tra SO_MIN_TRANS_PROC_NSEC e SO_MAX_TRANS_PROC_NSEC

5.4 Libro mastro

Il libro mastro è la struttura condivisa da tutti i nodi e gli utenti, ed è deputata alla memorizzazione delle transazioni eseguite. Una transazione si dice confermata solamente quando entra a far parte del libro mastro. Più in dettaglio, il libro mastro è formato da una sequenza di lunghezza massima SO_REGISTRY_SIZE di blocchi consecutivi. All'interno di ogni blocco sono contenute esattamente SO_BLOCK_SIZE transazioni. Ogni blocco è identificato da un identificatore intero progressivo il cui valore iniziale è impostato a 0.

Una transazione è univocamente identificata dalla tripletta (*timestamp, sender, receiver*). Il nodo che aggiunge un nuovo blocco al libro mastro è responsabile anche dell'aggiornamento dell'identificatore del blocco stesso.

STRUTTURA DEL LIBRO MASTRO



- la cella di indice SO_BLOCK_SIZE-1 è la transazione record
- le celle in [0, SO_BLOCK_SIZE-2] sono transazioni create da user

- quando viene creato tutti i valori delle struct transazione sono impostati a 0

5.5 Stampa

Ogni secondo il processo master stampa:

- numero di processi utente a nodo attivi
- il budget corrente di ogni processo utente e di ogni processo nodo, così come registrato nel libro mastro (inclusi i processi utente terminati). Se il numero di processi è troppo grande per essere visualizzato, allora viene stampato soltanto lo stato dei processi più significativi: quelli con maggior e minor budget.

- ci sono due array di questa struct in memoria globale, uno per i processi user e uno per i node
- ogni secondo questi array vengono aggiornati leggendo la nuova porzione del libro mastro scritto (salviamo l'indice dell'ultima riga letta dal libro mastro in book_row_print)
- per la stampa ci sono due metodi, invocati sulla base della lunghezza di questi array:
 - print-limits() viene invocato se gli array sono troppo lunghi
 - print-all() altrimenti

```
typedef struct {  
    pid_t pid;  
    int balance;  
} balance_proc;
```

5.6 Terminazione della simulazione

La simulazione terminerà in uno dei seguenti casi:

- sono trascorsi SO_SIM_SEC secondi,
- la capacità del libro mastro si esaurisce (il libro mastro può contenere al massimo SO_REGISTRY_SIZE blocchi),
- tutti i processi utente sono terminati.

se `status_simulazione[0] != 0` la simulazione è finita

- nel master controlliamo il tempo che passa, se passano SO_SIM_SEC secondi \rightarrow `status_simulazione[0] = 1`
- se un nodo scrive nell'ultimo blocco del libro mastro \rightarrow `status_simulazione[0] = 2`
- se l'ultimo processo utente termina
 \hookrightarrow `status_simulazione[0] = 3`

Alla terminazione, il processo master obbliga tutti i processi nodo e utente a terminare, e stamperà un riepilogo della simulazione, contenente almeno queste informazioni:

- ragione della terminazione della simulazione \rightarrow `switch(status_simulazione) { ... }`
- bilancio di ogni processo utente, compresi quelli che sono terminati prematuramente,
- bilancio di ogni processo nodo \rightarrow metodi per la stampa a ogni secondo
- numero dei processi utente terminati prematuramente \rightarrow `stato_utente_terminato`
- numero di blocchi nel libro mastro \rightarrow `book_read_write` \rightarrow `remaining_trans_in_lp`
- per ogni processo nodo, numero di transazioni ancora presenti nella transaction pool

6 Descrizione del progetto: versione "normal" (max 30)

All'atto della creazione da parte del processo master, ogni nodo riceve un elenco di SO_NUM_FRIENDS nodi amici.

Il ciclo di vita di un processo nodo si arricchisce quindi di un ulteriore step:

- periodicamente ogni nodo seleziona una transazione dalla transaction pool che è non ancora presente nel libro mastro e la invia ad un nodo amico scelto a caso (la transazione viene eliminata dalla transaction pool del nodo sorgente)

Quando un nodo riceve una transazione, ma ha la transaction pool piena, allora esso provvederà a spedire tale transazione ad uno dei suoi amici scelto a caso. Se la transazione non trova una collocazione entro SO_HOPS l'ultimo nodo che la riceve invierà la transazione al processo master che si occuperà di creare un nuovo processo nodo che contiene la transazione scartata come primo elemento della transaction pool. Inoltre, il processo master assegna al nuovo processo nodo SO_NUM_FRIENDS processi nodo amici scelti a caso. Inoltre, il processo master sceglierà a caso altri SO_NUM_FRIENDS processi nodo già esistenti, ordinandogli di aggiungere alla lista dei loro amici il processo nodo appena creato.

nel Δ cose in più;

- all'inizio viene data una lista SO_NUM_FRIENDS di nodi amici
ogni TOT tempo il nodo sceglie un nodo dalla
TRANSACTION POOL (non ancora nel master)
e la invia a un nodo amico o suo
- se un nodo riceve una transaz. ma ha la transaction
pool piena lo manda a un amico
 \hookrightarrow dopo SO_HOPS il nodo notifica il processo master (SIGUSR1)
che crea un nuovo processo nodo

7 Configurazione

I seguenti parametri sono letti a **tempo di esecuzione**, da file, da variabili di ambiente, o da `stdin` (a discrezione degli studenti):

- `SO_USERS_NUM`: numero di processi utente
- `SO_NODES_NUM`: numero di processi nodo
- `SO_BUDGET_INIT`: budget iniziale di ciascun processo utente
- `SO_REWARD`: la percentuale di reward pagata da ogni utente per il processamento di una transazione
- `SO_MIN_TRANS_GEN_NSEC`, `SO_MAX_TRANS_GEN_NSEC`: minimo e massimo valore del tempo (espresso in nanosecondi) che trascorre fra la generazione di una transazione e la seguente da parte di un utente
- `SO_RETRY`, numero massimo di fallimenti consecutivi nella generazione di transazioni dopo cui un processo utente termina
- `SO_TP_SIZE`: numero massimo di transazioni nella transaction pool dei processi nodo
- `SO_MIN_TRANS_PROC_NSEC`, `SO_MAX_TRANS_PROC_NSEC`: minimo e massimo valore del tempo simulato (espresso in nanosecondi) di processamento di un blocco da parte di un nodo
- `SO_SIM_SEC`: durata della simulazione (in secondi)
- `SO_NUM_FRIENDS` (solo versione max 30): numero di nodi amici dei processi nodo (solo per la versione full)
- `SO_HOPS` (solo versione max 30): numero massimo di inoltri di una transazione verso nodi amici prima che il master crei un nuovo nodo

Un cambiamento dei precedenti parametri non deve determinare una nuova compilazione dei sorgenti.

Invece, i seguenti parametri sono letti a **tempo di compilazione**:

- `SO_REGISTRY_SIZE`: numero massimo di blocchi nel libro mastro
- `SO_BLOCK_SIZE`: numero di transazioni contenute in un blocco

La seguente tabella elenca valori per alcune configurazioni di esempio da testare. Si tenga presente che il progetto deve poter funzionare anche con altri parametri.

parametro	letto a ...	“conf#1”	“conf#2”	“conf#3”
<code>SO_USERS_NUM</code>	run time	100	1000	20
<code>SO_NODES_NUM</code>	run time	10	10	10
<code>SO_BUDGET_INIT</code>	run time	1000	1000	10000
<code>SO_REWARD</code> [0–100]	run time	1	20	1
<code>SO_MIN_TRANS_GEN_NSEC</code> [nsec]	run time	100000000	10000000	10000000
<code>SO_MAX_TRANS_GEN_NSEC</code> [nsec]	run time	200000000	10000000	20000000
<code>SO_RETRY</code>	run time	20	2	10
<code>SO_TP_SIZE</code>	run time	1000	20	100
<code>SO_BLOCK_SIZE</code>	compile time	100	10	10
<code>SO_MIN_TRANS_PROC_NSEC</code> [nsec]	run time	1000000	1000000	?
<code>SO_MAX_TRANS_PROC_NSEC</code> [nsec]	run time	20000000	1000000	?
<code>SO_REGISTRY_SIZE</code>	compile time	1000	10000	1000
<code>SO_SIM_SEC</code> [sec]	run time	10	20	20
<code>SO_FRIENDS_NUM</code>	run time	3	5	3
<code>SO_HOPS</code>	run time	10	2	10

8 Requisiti implementativi

Il progetto (sia in versione “minimal” che “normal”) deve

- essere realizzato sfruttando le tecniche di divisione in moduli del codice,
- essere compilato mediante l'utilizzo dell'utility `make`
- massimizzare il grado di concorrenza fra processi
- deallocare le risorse IPC che sono state allocate dai processi al termine del gioco
- essere compilato con almeno le seguenti opzioni di compilazione:

```
gcc -std=c89 -pedantic
```

- poter eseguire correttamente su una macchina (virtuale o fisica) che presenta parallelismo (due o più processori).

Per i motivi introdotti a lezione, ricordarsi di definire la macro `_GNU_SOURCE` o compilare il progetto con il flag `-D_GNU_SOURCE`.