Relazione os-proj

Scritta da *Roberto Carletto* (matr: **947997**) e *Filippo Bogetti* (matr: **944849**).

Fetch dei parametri della configurazione

I parametri disponibili a **compile time** si trovano nel `makefile`, è possibile definire dei valori custom modificando le *linee 4 e 5* oppure utilizzare i valori delle varie configurazioni da testare.

- Per compilare con i valori custom di `SO_BLOCK_SIZE` e `SO_REGISTRY_SIZE`, lancia `make all`.
- Per compilare con i valori della configurazione 1, lancia `make all cfg=1`.
- Per compilare con i valori della configurazione 2, lancia `make all cfg=2`.
- Per compilare con i valori della configurazione 3, lancia `make all cfg=3`.

Tutti i parametri disponibili a **running time** sono prelevati dalle variabili d'ambiente che devono, per ovvie ragioni, essere caricate prima di poter lanciare il `master`.

Per caricare le variabili d'ambiente, utiliziamo l'utility `source` di `bash`, `zsh`, ..., ma non c'è per la shell `sh`.

Se si ha l'utility `source`, lancia `source cfg/<conf_file>.cfg`. Anche in questo caso:

- Per utilizzare i valori custom dei parametri di *runtime* modifica il file `cfg/custom.cfg` e lancia `source cfg/custom.cfg`.
- Per utilizzare i valori della configurazione 1, lancia `source cfg/conf_1.cfg`.
- Per utilizzare i valori della configurazione 2, lancia `source cfg/conf_2.cfg`.
- Per utilizzare i valori della configurazione 3, lancia `source cfg/conf_3.cfg`.

Se, invece, non si ha l'utility `source`, bisogna appendere a mano il contenuto del file di configurazione della directory `cfg/` che si intende utilizzare nel file `~/.shrc` o `~/.bashrc` o `~/.zshrc`. In questo caso, è anche necessario riavviare il terminale. Sicuramente, è molto più pulita la versione con `source` perchè una volta che chiudiamo il terminale, quelle variabili d'ambiente non saranno più disponibili e sarà necessario rilanciare uno dei comandi precedenti.

La funzione responsabile del fetch dei parametri è `get_configuration()` del file `master.c`.

I nomi dei parametri di runtime sono hardcodati nell'array di stringhe `conf_names[N_RUNTIME_CONF_VALUES]`, dove `N_RUNTIME_CONF_VALUES` è una macro che indica il numero di parametri di runtime da prelevare. La `get_configuration()` cicla su questo array e utilizza la `getenv()` per prelevare i valori delle variabili d'ambiente, la `strtoul()` per convertirli in `unsigned long` e poi verifica la loro validità. Ovviamente, controlla anche la presenza dei parametri di compile time. Se non ci sono errori, alla fine dell'esecuzione di questa funzione troveremo in shared memory la configurazione, altrimenti si stampa il messaggio d'errore e si termina.

Parametri di compilazione

Di default, i parametri di compilazione di `gcc` sono `std=c89 -pedantic -O2 -D SO_BLOCK_SIZE=$(SO_BLOCK_SIZE) -D SO_REGISTRY_SIZE=$(SO_REGISTRY_SIZE)` dove `SO_BLOCK_SIZE` e `SO_REGISTRY_SIZE` possono essere custom oppure definiti da una delle 3 configurazioni. Per ulteriori informazioni vedi il paragrafo precedente.

Se si volesse compilare in modo da semplificare il debugging, bisogna lanciare `make all` oppure un altro comando per scegliere la configurazione, seguito da `debug=1`. Ad esempio: `make all debug=1` oppure `make all cfg=2 debug=1`. I parametri di compilazione di `gcc` in questo caso sono `std=c89 -pedantic -O0 -g -D DEBUG`.

Inoltre, si suggerisce di abilitare il *core dumping* con il comando `ulimit -c unlimited` e di utilizzarlo con `gdb` in caso di errori.

Inizializzazione degli oggetti IPC

All'avvio del processo `master` si **creano** gli oggetti IPC utilizzati:

- 5 segmenti di **shared memory**.
    1. Parametri di configurazione *runtime*
    2. Informazioni sui *processi utente*
    3. Informazioni sui *processi nodo*
    4. Libro Mastro (*blockchain*)
    5. Numero del blocco
- 5 **semafori** per l'accesso in lettura/scrittura in shared memory. Non c'è il semaforo per l'accesso al segmento di shared memory dedicato alla configurazione perchè tutti i processi utente e nodo possono accedere solo in lettura. Inoltre il processo master non la modifica dopo aver generato i processi figli.

    1. Semaforo per le informazioni sui processi utente
    2. Semaforo per le informazioni sui processi nodo
    3. Semaforo per il Libro Mastro
    4. Semaforo per il numero del blocco
    5. Semaforo per l'inizio della simulazione
- N **message queues** utilizzate come *transaction pools*, dove N è il numero di processi nodo.

Il semaforo per l'inizio della simulazione `semSimulation` è inizializzato a $\texttt{SO\_NODES\_NUM} + \texttt{SO\_USERS\_NUM}$ e viene decrementato da ogni processo figlio quando è pronto a inviare / ricevere transazioni. Tutti i processi figli fanno una *wait for zero* su questo semaforo con la `semop()` e l'operazione a zero. Quando il semaforo arriva a zero, la simulazione inizia.

Tutti gli oggetti IPC, tranne le *message queues*, sono creati utilizzando una chiave in `common.h`. Le *message queues*, invece, vengono create con `ftok()` passando il pathname del nodo `./bin/node` e il suo PID. In questo modo i processi utente possono inviare transazioni alle *transaction pools* dei nodi in base al loro PID.

Tutti gli ID degli oggetti IPC creati vengono scritti su un file `./out/ipc_ids` per poter controllare che siano stati rimossi una volta terminata l'esecuzione.

Prima della generazione dei processi figli (utenti e nodi), si controlla se la configurazione permette di creare delle Message Queues che contengono `SO_TP_SIZE` transazioni. Si effettua questo controllo perchè la dimensione di $\texttt{sizeof}(\texttt{msgbuf}) \cdot \texttt{SO\_TP\_SIZE}$ può eccedere i limiti di sistema. Di default, nei nostri sistemi operativi, la dimensione massima delle message queues è di `ulimit -q` $(= 819200)$ bytes.

La configurazione 1 eccede questo limite e per poter essere testata è necessario modificare questo limite con i privilegi di `root`. Se non si supera questo limite, vengono generati i processi figli.

Le procedure `init()`, `init_conf()`, `init_semaphores()` e `init_sharedmem()` sono responsabili per la creazione degli oggetti IPC necessari. Le message queues sono create nel momento in cui si generano i processi nodo.

All'avvio del processo `user` si **accede** agli oggetti IPC utilizzati:

- 5 segmenti di **shared memory**.
    1. Parametri di configurazione *runtime* in modalità `SHM_RDONLY`

    2. Informazioni sui *processi utente*
    3. Informazioni sui *processi nodo* in modalità `SHM_RDONLY`
    4. Libro Mastro (*blockchain*) in modalità `SHM_RDONLY`
    5. Numero del blocco in modalità `SHM_RDONLY`
- 5 **semafori** per l'accesso in lettura/scrittura in shared memory.
    1. Semaforo per le informazioni sui processi utente
    2. Semaforo per le informazioni sui processi nodo
    3. Semaforo per il Libro Mastro
    4. Semaforo per il numero del blocco
    5. Semaforo per l'inizio della simulazione
- 1 **message queue** / *transaction pool* di un nodo casuale a cui mandare la transazione ogni volta che si crea una transazione.

Prima di poter creare le transazioni, decrementa il semaforo `semSimulation` e fa una *wait for zero*.

Le procedure `init()`, `init_conf()`, `init_semaphores()` e `init_sharedmem()` sono responsabili per l'accesso agli oggetti IPC necessari. L'accesso alla message queue avviene alla creazione della transazione.

All'avvio del processo `nodo` si **accede** agli oggetti IPC utilizzati:

- 4 segmenti di **shared memory**.
    1. Parametri di configurazione *runtime* in modalità `SHM_RDONLY`
    2. Informazioni sui *processi nodo*
    3. Libro Mastro (*blockchain*)
    4. Numero del blocco
- 4 **semafori** per l'accesso in lettura/scrittura in shared memory.
    1. Semaforo per le informazioni sui processi nodo
    2. Semaforo per il Libro Mastro
    3. Semaforo per il numero del blocco
    4. Semaforo per l'inizio della simulazione

- 1 **message queue** / *transaction pool* di proprietà del nodo.

All'inizializzazione della message queue, si modifica la dimensione massima a `SO_TP_SIZE`. Prima di poter ricevere le transazioni, decrementa il semaforo `semSimulation` e fa una *wait for zero*.

Le procedure `init()`, `init_conf()`, `init_semaphores()`, `init_sharedmem()` e `init_msgqueue()` sono responsabili per l'accesso agli oggetti IPC necessari.

Semafori e Shared Memory

Le seguenti funzioni definite in `common.c` sono simili a quelle utilizzate negli esercizi di laboratorio con il **Prof. Radicioni**, servono per l'inizializzazione e per le operazioni sui semafori.

```
int initSemAvailable(int, int);
int initSemInUse(int, int);
int reserveSem(int, int);
int releaseSem(int, int);
int initSemSimulation(int, int, int, int);
```

Le seguenti funzioni vengono usate per le operazioni sui semafori quando si accede in lettura / scrittura in shared memory.

```
void initReadFromShm(int);
void endReadFromShm(int);
void initWriteInShm(int);
void endWriteInShm(int);
```

Rimozione degli oggetti IPC

Per quanto riguarda i segmenti di **shared memory**, i processi figli chiamano la `shmdt()` prima di morire per una condizione d'errore o per la fine della simulazione. Il processo master chiama la `shmdt()` seguita dalla rimozione dell'oggetto IPC.

I **semafori** e le **message queues** vengono rimosse dal processo master al termine della simulazione o in condizioni d'errore.

Il processo master ha più condizioni d'errore dei processi figli che vengono generati se e solo se tutto ciò che viene prima è andato a buon fine, quindi sono state introdotte delle variabili "booleane" `nodes_generated`, `users_generated` che vengono utilizzate per capire quali oggetti IPC sono stati inizializzati e si possono rimuovere.

La procedura `shutdown()` del master e dei processi figli è responsabile della rimozione o del detach dagli oggetti IPC.

Segnali

Le seguenti funzioni definite in `common.c` sono simili a quelle utilizzate negli esercizi di laboratorio con il **Prof. Bini**.

```
sigset_t block_signals(int count, ...);
sigset_t unblock_signals(int count, ...);
void reset_signals(sigset_t old_mask);
struct sigaction set_handler(int sig, void (*func)(int));
```

All'avvio del processo `master` la procedura `init_sighandlers()` è responsabile per settare i **signal handlers**. Questa procedura richiama la `set_handler()` che, a sua volta, utilizza `sigaction()` per settare il signal handler con il flag `SA_NODEFER`.

Quando ci si trova in sezione critica, ad esempio quando si trattiene un semaforo per la scrittura in shared memory, viene chiamata la `block_signals()` passando come parametri tutti i segnali che un certo processo potrebbe ricevere. Alla fine della sezione critica, si richiama la `unblock_signals()` con gli stessi segnali. Alla scrittura della relazione, ci siamo resi conto che sarebbe stato meglio utilizzare la più semplice e ottima `reset_signals()` del **Prof. Bini** che tramite la `sigprocmask()` setta la vecchia maschera dei segnali ritornata da `block_signals()`.

Il processo `master` utilizza 4 *signal handlers*:

- **SIGINT** e **SIGTERM** vengono gestiti dal `sigterm_handler()` che termina semplicemente la simulazione in modo pulito, ovvero killa tutti i processi, stampa le statistiche della simulazione e rimuove gli oggetti IPC.
- **SIGUSR1** è gestito dal `sigusr1_handler()`. Questo segnale viene inviato da un processo nodo quando si accorge che la *blockchain* ha raggiunto `SO_REGISTRY_SIZE` blocchi. L'handler permette di terminare la simulazione in modo pulito appena il Libro Mastro è pieno.
- **SIGALRM** è gestito dal `sigalrm_handler()`. Questo segnale arriva quando dall'avvio della simulazione sono trascorsi `SO_SIM_SEC` secondi. L'handler permette di terminare la simulazione in modo pulito appena scade il tempo.
- **SIGCHLD** è gestito dal `sigchld_handler()`. Questo segnale arriva quando uno dei processi figli muore per errore oppure perchè ha terminato l'esecuzione. Se il segnale arriva durante la simulazione, significa che un processo `user` è

morto. (Sarebbe più opportuno effettuare un controllo sul *sender* del segnale e vedere se effettivamente si tratta del PID di un utente). I processi nodo possono morire solo se ricevono un `SIGINT` (dal master) alla fine della simulazione oppure se viene rimossa la sua message queue (evento che può avvenire solo dall'esterno). Quando muore uno `user`, se la simulazione non è finita si incrementa la variabile `early_deaths` che indica il numero di utenti morti in modo prematuro.

Il processo `user` utilizza 2 *signal handlers*:

- **SIGINT** viene gestito dal `sigint_handler()` che chiama la `getBilancio()` per effettuare un ultimo calcolo del budget per scriverlo in shared memory e poi termina in modo pulito, ovvero chiama la procedura `shutdown()` che fa il detach / la rimozione degli oggetti IPC. Questo segnale viene inviato dal processo master alla fine della simulazione.
- **SIGUSR1** viene gestito dal `sigusr1_handler()` che scatena, se possibile, la creazione di una transazione.

Il processo `nodo` utilizza 1 *signal handler*:

- **SIGINT** viene gestito dal `sigint_handler()` che calcola il numero di transazioni rimaste nella *transaction pool* più il numero di transazioni processate ma non ancora scritte sul libro mastro. Scrive questo valore in shared memory e poi termina in modo pulito con la procedura `shutdown()`. Questo segnale viene inviato dal processo master alla fine della simulazione.

Ciclo di vita

Master

Quando si avvia il processo master, viene letta la configurazione e vengono inizializzati gli oggetti IPC. Successivamente inizia la generazione dei processi nodo. La procedura `nodes_generation()` crea `SO_NODES_NUM` processi figli con la `fork()`, crea le loro message queues, e scrive su shared memory le informazioni iniziali.

Per ogni nodo vengono salvate le seguenti informazioni:

```
typedef struct
{
    pid_t pid;
    int reward;
    int unproc_trans;
} node;
```

Quindi si salva il PID del nodo appena creato e si settano `reward` e `unproc_trans` a zero. Successivamente si fa la syscall `execve()`.

In seguito, inizia la generazione dei processi utente. La procedura `users_generation()` crea `SO_USERS_NUM` processi figli con la `fork` e scrive su shared memory le informazioni iniziali.

Per ogni utente vengono salvate le seguenti informazioni:

```
typedef struct
{
    pid_t pid;
    int budget;
    int alive;
} user;
```

Quindi si salva il PID dell'utente appena creato e si settano `budget` a `SO_BUDGET_INIT` e `alive` a 1. Successivametne si fa la syscall `execve()`.

Il processo master tiene traccia del numero di processi utente e nodo ancora attivi con le variabili `remaining_nodes` e `remaining_users`.

Infine, si chiama `alarm()` per far scattare un segnale dopo che sono passati `SO_SIM_SEC` secondi e si avvia un loop infinito grazie al quale si effettua la stampa ad ogni secondo. Se il numero di utenti o nodi è > 5, si stampano solo le informazioni dei più ricchi e dei più poveri, altrimenti si stampano tutti.

Alla fine della simulazione, il master killa tutti i processi inviando il segnale `SIGINT`, stampa tutte le informazioni sui nodi e utenti, salva sul file `out/blockchain` il contenuto del libro mastro, rimuove gli oggetti IPC e termina.

Nodo

Quando si avvia il processo nodo, si effettua l'accesso agli oggetti IPC e si avvia un loop infinito grazie al quale il nodo può leggere dalla message queue e scrivere i blocchi di transazioni sul libro mastro. La `msgrcv()` non usa il flag `IPC_NOWAIT`, quindi il nodo aspetta finchè non arriva una transazione.

Il nodo lavora con queste strutture dati:

```
typedef struct
{
    struct timespec timestamp;
    int sender;
    int receiver;
```

```
    int quantity;
    int reward;
} transaction;

typedef struct
{
    long mtype;
    transaction trans;
} msgbuf;

typedef struct
{
    unsigned int block_number;
    transaction transBlock[SO_BLOCK_SIZE];
} block;
```

Le transazioni in arrivo vengono aggiunte in un blocco fino a quando la sua dimensione arriva a `SO_BLOCK_SIZE-1`. Poi si aggiunge la transazione di reward per il nodo e si scrive il blocco sul libro mastro. Ogni volta che si aggiunge la transazione di reward, si aggiorna l'informazione `reward` del nodo in shared memory.

Quando termina la simulazione, il nodo conta il numero di altre transazioni ancora presenti sulla transaction pool e fa la somma con il numero di transazioni processate ma non ancora scritte sul libro mastro. Aggiorna di conseguenza il valore `unproc_trans` in shared memory.

Successivamente, si sgancia dai segmenti di shared memory utilizzati e termina.

User

Quando si avvia il processo utente, si effettua l'accesso agli oggetti IPC e si avvia un loop fino a quando l'utente non esaurisce `SO_RETRY` tentativi di creaazione di una transazione. Nel loop l'utente calcola sempre il proprio bilancio (`getBilancio()`) partendo da `SO_BUDGET_INIT` e considerando sia i blocchi che trova sul libro mastro, sia le transazioni inviate ma che non sono state ancora scritte sul libro mastro. Viene fatto per evitare che un utente possa inviare transazioni con una quantità che supera il proprio budget!

Per fare ciò si utilizza una lista che tiene traccia di tutte le transazioni in fase di elaborazione da parte di un nodo:

```
struct pendingTr
{
    transaction trans;
    struct pendingTr *next;
};
```

Si utilizzano delle funzioni che lavorano con questa lista:

```
void addToPendingList(transaction tr);
void removeFromPendingList(transaction tr);
void freePendingList();
```

Ogni volta che si calcola il bilancio e si legge dal libro mastro, se si trova una transazione presente nella lista, questa viene rimossa perchè significa che è stata processata dal nodo. Quindi si sommano al bilancio tutte le transazioni che hanno come receiver il PID dell'utente, mentre si tolgono al bilancio tutte le transazioni che hanno come sender il PID dell'utente. In seguito, si tolgono al bilancio calcolato tutte le transazioni pendenti presenti nella lista. Si aggiorna il valore `budget` in shared memory.

Se il bilancio è ≥ 2 si può creare una transazione, altrimenti si incrementa il numero di fallimenti. La transazione creata con `createTransaction()` necessita di un utente destinatario; il ricevitore della transazione può essere solo un utente vivo, per cui si può tentare di estrarre casualmente un utente ancora vivo al massimo 5 volte. Se non si trova, si incrementa il numero di fallimenti.

La `msgsnd()` utilizza il flag `IPC_NOWAIT`, quindi si prova ad inviare una transazione e se avviene un errore questo è considerato come un tentativo fallito. Questa è una condizione che avviene spesso nella configurazione 2 con la quale le transaction pools dei nodi si saturano molto velocemente.

Una volta raggiunto il numero massimo di tentativi oppure viene killato dal processo master, il processo utente termina sganciandosi dai segmenti di shared memory a cui aveva fatto accesso.

Simulazioni

Configurazione 1

La configurazione 1 ha come parametri compile time:

```
SO_BLOCK_SIZE=100
SO_REGISTRY_SIZE=1000
```

e di runtime:

```
SO_USERS_NUM=100
SO_NODES_NUM=10
SO_BUDGET_INIT=1000
SO_REWARD=1
SO_MIN_TRANS_GEN_NSEC=100000000
SO_MAX_TRANS_GEN_NSEC=200000000
SO_RETRY=20
SO_TP_SIZE=1000
SO_MIN_TRANS_PROC_NSEC=10000000
SO_MAX_TRANS_PROC_NSEC=20000000
SO_SIM_SEC=10
SO_FRIENDS_NUM=3
SO_HOPS=10
```

La caratteristica di questa configurazione è quella di possedere una `SO_TP_SIZE` troppo grande. In altre parole, una message queue non può contenere `SO_TP_SIZE` messaggi (`msgbuf`) senza modificare i limiti di sistema!

Configurazione 2

La configurazione 2 ha come parametri compile time:

```
SO_BLOCK_SIZE=10
SO_REGISTRY_SIZE=10000
```

e di runtime:

```
SO_USERS_NUM=1000
SO_NODES_NUM=10
SO_BUDGET_INIT=1000
SO_REWARD=20
SO_MIN_TRANS_GEN_NSEC=10000000
SO_MAX_TRANS_GEN_NSEC=10000000
SO_RETRY=2
SO_TP_SIZE=20
SO_MIN_TRANS_PROC_NSEC=1000000
SO_MAX_TRANS_PROC_NSEC=1000000
SO_SIM_SEC=20
SO_FRIENDS_NUM=5
SO_HOPS=2
```

La caratteristica di questa configurazione è che il numero di utenti è molto più alto rispetto ai nodi, il numero di tentativi concessi agli utenti è basso e le transaction pools possono contenere al massimo 20 transazioni.

A causa di questi parametri, in un determinato momento, un gran numero di utenti fallisce perchè le message queues dei nodi sono piene.

Configurazione 3

La configurazione 3 ha come parametri compile time:

```
SO_BLOCK_SIZE=10
SO_REGISTRY_SIZE=1000
```

e runtime:

```
SO_USERS_NUM=20
SO_NODES_NUM=10
SO_BUDGET_INIT=10000
SO_REWARD=1
SO_MIN_TRANS_GEN_NSEC=10000000
SO_MAX_TRANS_GEN_NSEC=20000000
SO_RETRY=10
SO_TP_SIZE=100
SO_MIN_TRANS_PROC_NSEC=10000000
SO_MAX_TRANS_PROC_NSEC=20000000
SO_SIM_SEC=20
SO_FRIENDS_NUM=3
SO_HOPS=10
```

La caratteristica di questa configurazione è che il libro mastro può contenere un gran numero di blocchi, ma gli utenti non hanno abbastanza budget per poterlo riempire totalmente. Per cui la simulazione converge nella situazione in cui tutti gli utenti muoiono per poco budget.