> PROCESSCHAIN

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, per questo motivo ri-battezzato “processchain”.

COMPILAZIONE

Per compilare il programma verrà utilizzato un makefile, cioè uno script, che, nel nostro caso, 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”).

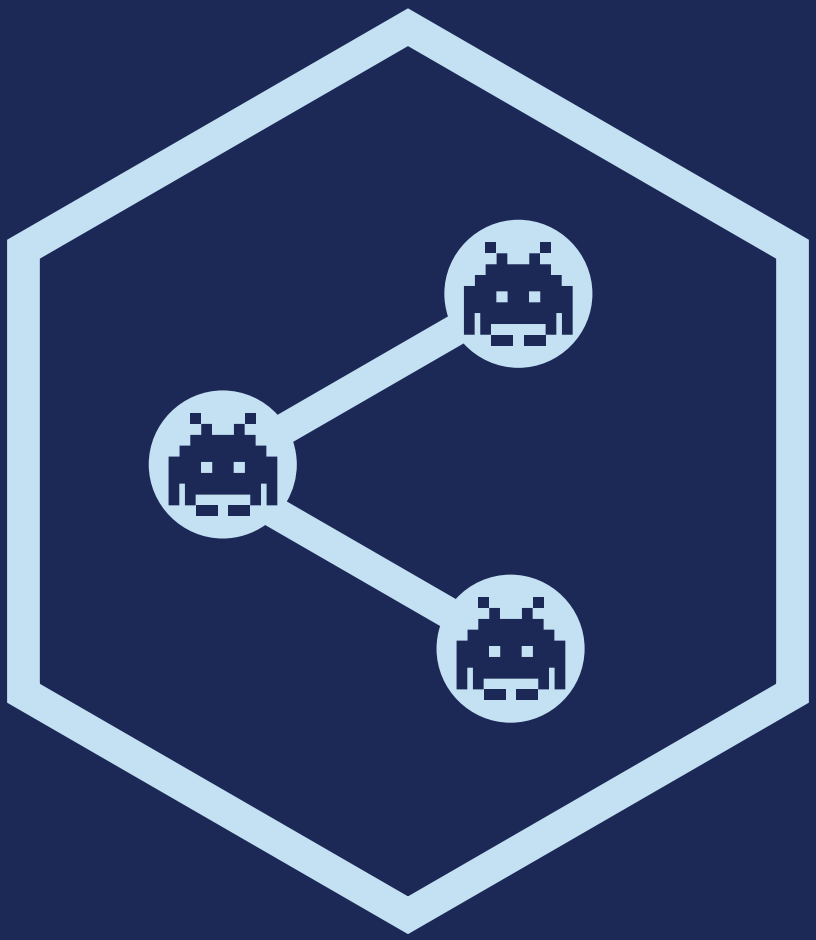
INIZIALIZZAZIONE

Si è deciso di utilizzare cinque **memorie condivise** per il funzionamento del codice.

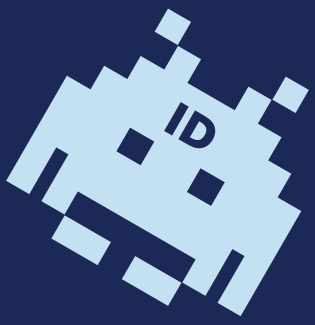
Queste aree serviranno per consentire l'accesso (in scrittura e/o in lettura) dei dati da parte dei diversi processi.

Ognuna di esse viene allocata all'inizio con una dimensione ben definita, queste sono:

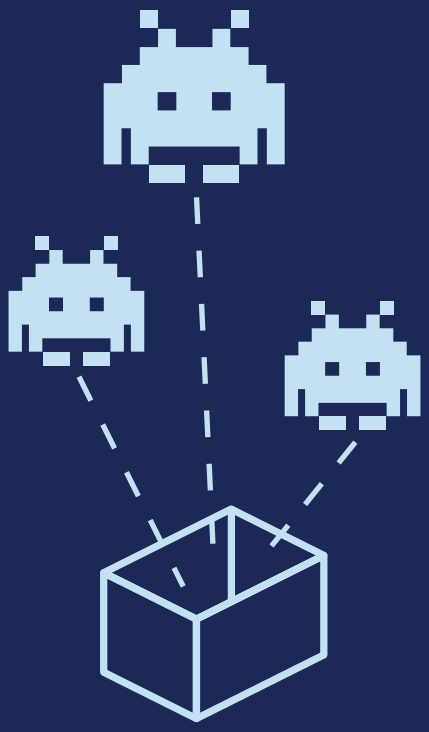
- La memoria condivisa del **libro mastro** dove verranno salvate le transazioni effettuate dai processi utente, è gestito come una matrice le cui righe rappresentano un blocco scritto da un nodo. L'accesso a tale area è libero in lettura ma gestito in mutua esclusione durante la scrittura.
- La memoria condivisa per l'**indice del libro mastro** sarà l'indice che indicherà la riga della matrice alla quale i processi nodo sono arrivati a scrivere. Analogamente all'area di memoria del libro mastro, l'accesso a quest'area verrà eseguito 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.



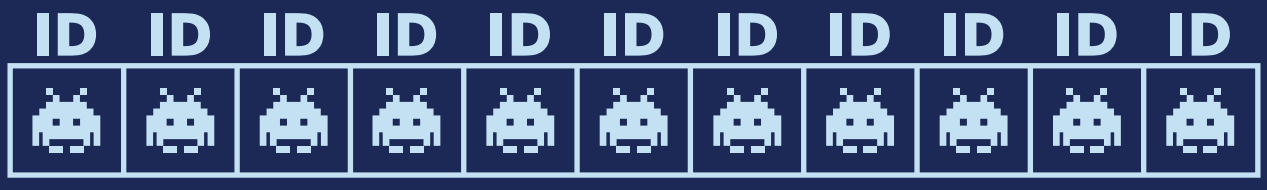
Come scelta progettuale è stato deciso, per le tre aree condivise successive, di utilizzare la posizione zero della memoria condivisa per contenere il numero di utenti e di nodi creati e le restanti celle per contenere i dati di ognuno di questi. I seguenti frammenti di memoria condivisa verranno popolati 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. I processi nodo non effettueranno *"l'attach"* a questo frammento di memoria in quanto non necessitano di essere a conoscenza dello stato degli utenti.
- La memoria condivisa contenente la **lista dei nodi amici** servirà per indicare quali saranno gli eventuali "amici" del nodo i-esimo, così da consentire l'invio delle eventuali transazioni.



Per la comunicazione delle transazioni dai processi utente ai processi nodo verrà utilizzata una **coda di messaggi** univoca per ogni nodo della simulazione**.**
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 "normal", riceverà le transazioni che non sono riuscite ad essere gestite dai processi nodo dopo SO_HOPS volte.



Per la sincronizzazione del programma utilizziamo i successivi 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 contatori, quindi consentono a n-processi contemporaneamente di accedere, nel nostro caso, alle code di messaggi e per rendere possibile la sincronizzazione inziale.
Questi sono:

- **Semaforo per il risveglio di tutti i processi**. Verrà utilizzato per sincronizzare l'inizio della simulazione, prima dei processi nodo e in seguito dei processi utente. La prima inizializzazione di tale semaforo verrà effettuata con valore SO_NODES_NUM, ogni nodo decrementerà di uno il semaforo ed effettuerà una chiamata wait for zero. Una volta creati tutti i nodi il master li farà partire decrementando a sua volta il semaforo. Tale operazione verrà eseguita in maniera analoga per gli utenti.
- **Semaforo per l'accesso alla coda di messaggi**. Tale semaforo, inizializzato a valore SO_TP_SIZE, servirà per garantire che ciascun nodo riceva al massimo sulla propria coda di messaggi SO_TP_SIZE transazioni che in seguito copierà e processerà nella propria transaction pool. Allo stesso modo gestiamo in maniera automatica il rifiuto in caso di raggiungimento di tale limite. L'utente verrà notificato durante la semop dell'impossibilità di inviare la transazione. In questo caso la coda di messaggi è il tramite tra i processi utente e i processi nodo.
- **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. In fase di verifica delle richieste di nuovi nodi, come scelta progettuale, si è deciso eseguire la creazione del nuovo nodo periodicamente e di assegnargli, se presenti, al massimo SO_TP_SIZE transazioni ricevute dal processo master. Questa decisione si sostituisce alla richiesta di generare un nuovo nodo ogniqualvolta il processo master riceva una transazione, ottimizzando l'utilizzo dei nodi e delle risorse.

I parametri necessari per l'esecuzione della simulazione vengono inseriti in un file.conf che verrà parsificato tramite una funzione sviluppata che riconosce come token il nome del parametro e il relativo valore, utilizzando come delimitatore il carattere '='. Il valore verrà convertito con la funzione atoi() da carattere ad intero.

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.

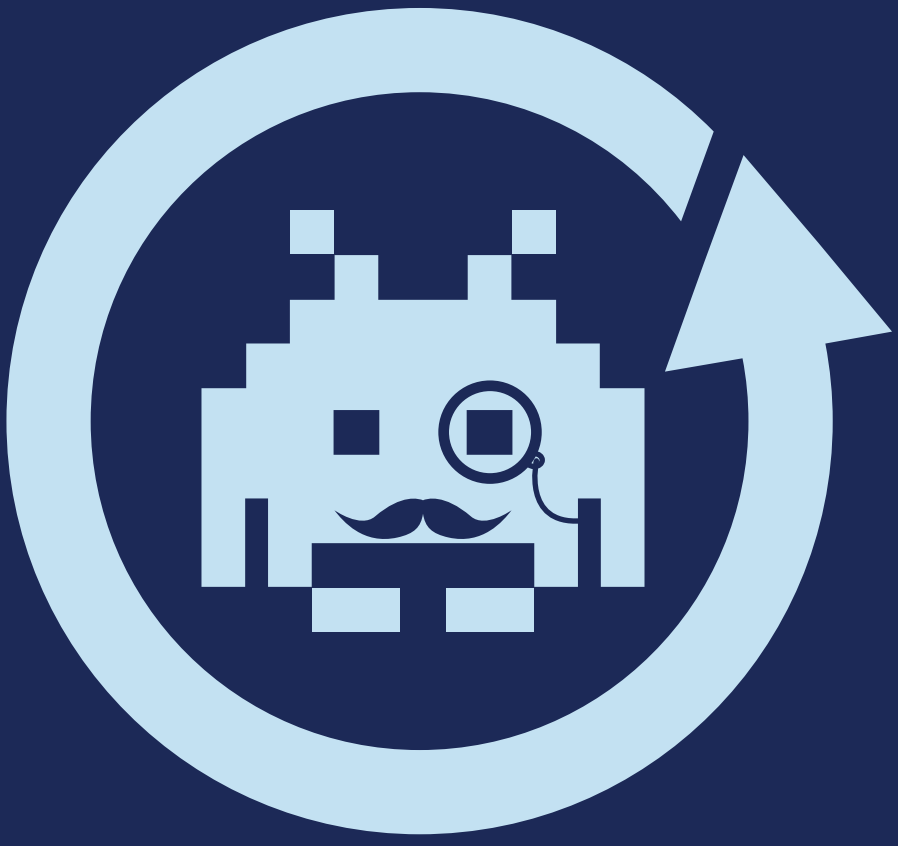
Nella versione completa abbiamo anche la possibilità che i figli nodo vengano creati dopo SO_HOPS volte che una transazione non è riuscita ad essere gestita, in questo caso abbiamo un controllo sulla msgrcv() della coda di messaggi del master con IPC_NOWAIT, che quindi rende la ricezione non bloccante, 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:

- verfica le condizioni di terminazione della simulazione e in tal caso con la funzione raise(SIGALARM) fa scattare l'handler che terminerà la simulazione e ne stampa il risultato finale con eventuale motivo di terminazione;
- stampa la situazione dei figli (budget, figli vivi etc…);
- controlla di non aver ricevuto transazioni nella propria coda di messaggi, in caso contrario creerà al massimo due nuovi nodi che gestiranno le transazioni appese, essendo successivamente aggiunti alle varie liste;
- dopo aver atteso la terminazione di tutti i processi figli, dealloca tutte le memorie condivise, code di messaggi e semafori istanziate all'inizio e termina;

Per mantenere il master nel suo ciclo è stata usata una variabile "master" con assegnato il valore MASTER_CONTINUE = 1 e MASTER_STOP = 0. L'ultima macro verrà impostata dall'handler in seguito alla ricezione di un segnale di tipo ALARM e SIGINT che causeranno la fine della simulazione e il corretto deallocamento di tutti gli oggetti IPC.



CICLO VITA PROCESSI UTENTE

Il processo utente tramite una execpl riceve i parametri necessari dal processo master.

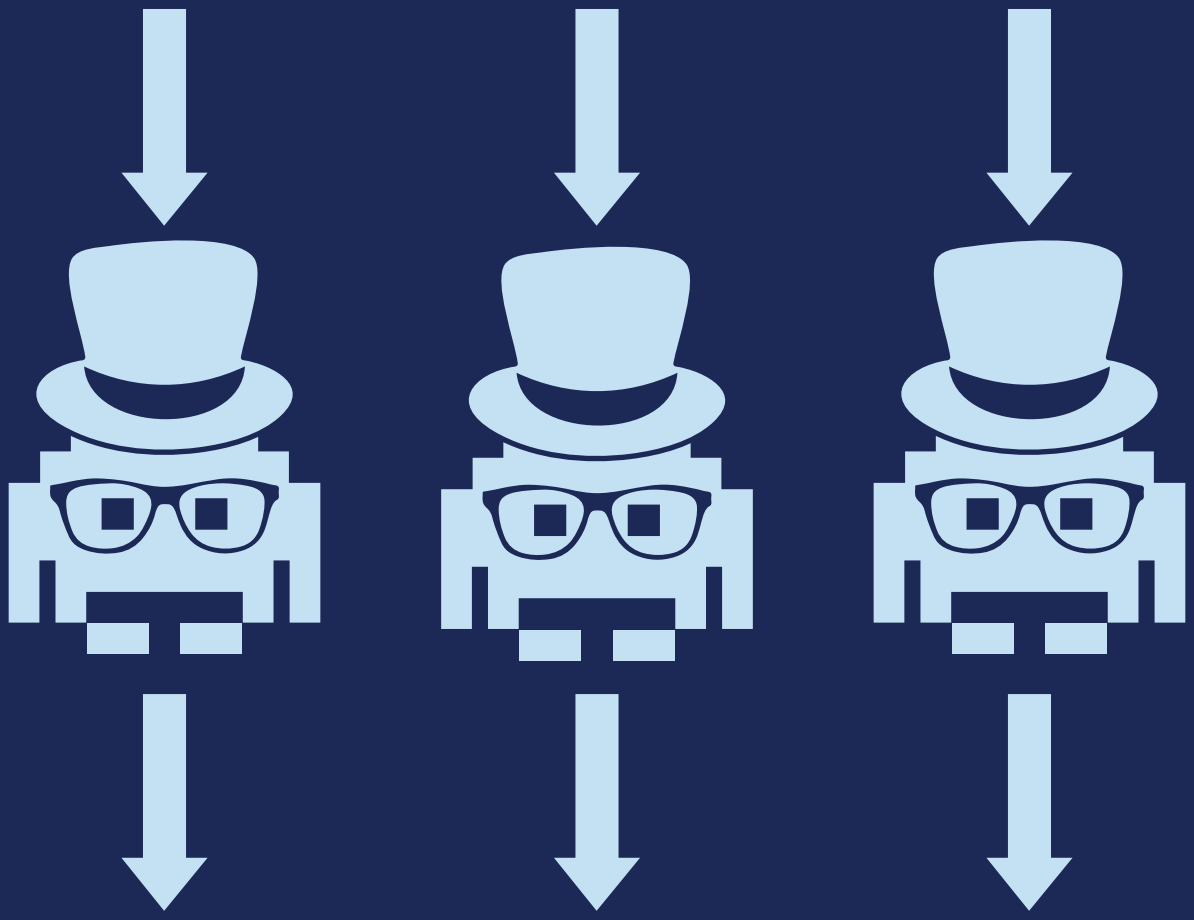
Come per il master, anche il processo utente userà un ciclo while con una variabile “utente” impostata a UTENTE_CONTINUE (che equivale al valore di 1), all’interno del quale farà:

- il calcolo del proprio budget dove controllerà di aver ricevuto transazioni, leggendole dal libro mastro, e nel caso sommandole ad esso;
- le estrazioni casuali di code di messaggi, utenti e quantità di denaro da inviare per creare la transazione;
- decrementerà il semaforo per l’accesso alla coda di messaggi;
- in caso di fallimento per mancanza di budget, dopo SO_RETRY volte, la variabile dell’utente viene impostata a UTENTE_STOP (che equivale a zero) che causerà l’uscita dal ciclo di vita con conseguente cambiamento del proprio stato in USER_KO nella memoria condivisa degli utenti e la deallocazione delle risorse utilizzate;
- se la semop sul semaforo associato alla coda di messaggi del nodo restituisce errno settato a EAGAIN, e la transaction pool è quindi piena, l’utente proverà ad inviare la transazione ad un nodo amico del nodo scelto e dopo SO_HOPS volte la invierà 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 master.
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:

- controlla la presenza di messaggi sulla propria coda, in caso affermativo li salva nella transaction pool locale;
- 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 e lo processa;
- accede in mutua esclusione al libro mastro per la scrittura del blocco;
- 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 permette al master di deallocare le risorse utilizzate.



TRANSAZIONI

Le transazioni saranno gestite con una struct chiamata, appunto, “transazione”, che conterrà i campi richiesti. Ogni struct verrà inserita in un messaggio che avrà come campo anche soHops, per permettere di gestire l’invio ai nodi amici, e 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.

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 simulazione non sarà conclusa.
Nel caso di SIGALARM la simulazione si concluderà 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

Un file Registro.dat verrà creato e conterrà tutti i blocchi scritti dai nodi durante la simulazione.

GAME OVER