Progetto Sistemi Operativi 2021/2022: relazione conclusiva

Come avviare il progetto:

1. Compilare utilizzando la regola all del makefile: make all
2. Avviare il processo master: ./master.out
3. In alternativa usare make run che combina i punti 1 e 2

La directory Conf contiene alcune configurazioni di prova: per scegliere quale utilizzare è sufficiente specificarne il percorso come primo ed unico parametro del processo master.
La regola rm consente di rimuovere tutti i file creati durante l'utilizzo e la compilazione del progetto.
Compilando con il flag -DESSENTIALS_PRINTS = 1 vengono effettuate alcune stampe non essenziali.
Se il flag non viene specificato le stampe aggiuntive vengono effettuate.
Compilando make allEssentialsPrints vengono effettuate solo le stampe essenziali.

Struttura logica del progetto:
Data la complessità del software da realizzare, abbiamo deciso di scomporne le funzionalità in tre moduli interagenti: Master, User e Node.
Essi corrispondono alle tre sezioni omonime indicate nella specifica del progetto.
Lo pseudocodice dei tre moduli è il seguente:

Modulo	Pseudocodice

Master	`Master() {`
	`  leggiParametriConfigurazione();`
	`  impostaHandler();`
	`  creaIPCFacilities();`
	`  inizializzaIPCFacilities();`
	`  creaProcessiUser();`
	`  creaProcessiNode();`
	`  impostaTimerSimulazione();`
	`  /* porta a 0 il valore di fairStartSem */`
	`  faiPartireSimulazione();`
	`  while(1) {`
	`  controlloLibroMastroPieno();`
	`  ricalcoloDeiBudget();`
	`  stampaDeiBudget();`
	`  controlloTerminazioneAnticipataUsers();`
	`  controlloTerminazioneAnticipataNodes();`
	`  if(numUsersAttivi == 0)`
	`  terminaSimulazione();`
	`  else if(numNodesAttivi == 0)`
	`  terminaSimulazione();`

	aspettaUnSecondo();
	}
	}

Node	Node() {
	leggiVarAmbiente();
	collegamentoIPCFacilities();
	leggiAmici();
	if(Node == “NORMAL”)
	waitForZeroSuFairStartSem();
	impostaHandlerSegnali();
	while(!aspettaTerminazione) {
	estraiBlocco();
	aggiuntaTransazioneReward();
	elaborazioneBlocco();
	scriviBloccoSuLibroMastro();
	if(libroMastroPieno) {
	segnalaLibroMastroPienoAMaster();
	aspettaTerminazione = 1;
	}
	controlloMessaggiTPPiena();
	invioTransazioneAdAmico();
	}
	scollegamentoIPCFacilities();
	segnalazioneTerminazioneAMaster();
	exit();
	}

User	User() {
	numFallimenti = 0;
	leggiVarAmbiente();
	collegamentoIPCFacilities();
	impostaHandlerSegnali();
	waitForZeroSuFairStartSem();
	while(1) {
	controlloTransazioniFallite();
	calcoloBilancio();
	if(bilancio ⩾ 2) {
	generaTransazione();
	attendiGenerazioneTransazione();
	}
	else {
	numFallimenti++;
	if(numFallimenti == SO_RETRY)
	terminaEsecuzione();
	}
	}
	}

Stante la complessità del progetto, un unico sorgente con tutto il codice sarebbe risultato scomodo da sviluppare e poco pratico da manutenere.

Inoltre, questa suddivisione ha facilitato la suddivisione dei compiti, sia in fase di sviluppo, che di refactoring e testing, tra i vari componenti del gruppo.
L'utilizzo di questi tre moduli ha consentito di separare nettamente i compiti dei tre principali attori del progetto.
Un altro vantaggio derivante da tale scelta è la possibilità di sviluppare un certo modulo focalizzandosi interamente su di esso, potendo contare sul fatto che determinate funzionalità, come l'allocazione degli oggetti IPC e la segnalazione della fine della simulazione, fossero già state implementate in altri moduli.

Questa scelta presenta anche alcuni svantaggi, tutti derivanti dal dover utilizzare un'execve per caricare il codice di utente e nodi.
La semantica di tale system call prevede infatti di scollegare il processo dai segmenti di memoria e, in generale, operando una sostituzione completa della vecchia immagine di memoria del processo, complica la condivisione di alcuni dati, come i parametri di configurazione.
Alcuni di questi problemi hanno richiesto di eseguire nuovamente delle operazioni, come il collegamento agli oggetti IPC, già eseguite dalla fork, cosa che non sarebbe stata necessaria qualora avessimo lavorato su di un solo modulo.

Un ulteriore modulo, error, contiene il codice necessario alla stampa di messaggi di errore, corredati di PID del processo e linea del sorgente.
Il modulo contiene due funzione: safeErrorPrint ed unsafeErrorPrint.
L'operato delle due funzioni è uguale, a differenziarli sono le funzioni utilizzate: in unsafeErrorPrint abbiamo utilizzato write in loco della printf, in modo da disporre di una funzione async-signal-safe, da usare negli handler.
Abbiamo deciso di creare un modulo a parte perché tale codice è comune a tutti e tre i moduli e quindi avremmo dovuto duplicarlo, perdendo in manutenibilità e leggibilità.
In questo modo, qualsiasi modifica alla segnalazione degli errori non richiederà la compilazione di master, nodo ed utente ma solo quella del modulo error.
Inoltre, tale modulo può essere effettivamente caricato in memoria principale solo se necessario.

Strutture IPC
Salvo quando diversamente specificato nell'elenco seguente, tutte le facilities IPC vengono create e deallocate dal master.
Questa soluzione ci garantisce che gli oggetti vengano creati sicuramente prima dell'utilizzo e deallocati solo dopo la terminazione di tutti i potenziali utilizzatori.
Data la delicatezza di questa operazione, qualora si verifichi un errore la simulazione verrà terminata.

Generazione chiavi

Abbiamo deciso di generare la chiave di ogni oggetto IPC utilizzando la ftok.
Questa funzione ci garantisce una chiave quasi sempre univoca, anche se esistono delle configurazioni (come params_5.txt) in cui l'elevato numero di nodi causa il fallimento della creazione della transaction pool del nuovo nodo.
La ftok consente una condivisione molto semplice della chiave, dato che è sufficiente richiamarla con gli stessi parametri per ottenere il medesimo risultato. Quest'ultimo è aspetto molto importante in questo caso, dato che dopo la fork per la generazione di nodi ed utenti eseguiamo un'execve, che non consente lo scambio degli ID tramite variabili locali.
Per generare la chiave la funzione viene richiamata passando la costante simbolica contenete il pathname del file corrispondente al tipo dell'oggetto (shmfile.txt, semfile.txt, msgfile.txt) ed un ulteriore seme di generazione, costituito da una costante simbolica definita in info.h oppure dal pid del processo possessore della facility.

Il libro mastro

Abbiamo implementato questa struttura con un'area di memoria condivisa.
Siamo giunti a tale decisione considerando che:

i) La semantica delle operazioni di lettura deve essere non distruttiva
ii) Durante la simulazione viene effettuato un numero molto elevato di accessi, tale da richiedere tempi di lettura e scrittura il più contenuti possibile
iii) Alcuni processi (gli utenti) non possono modificare il contenuto del libro mastro

Il meccanismo IPC più aderente a questi requisiti è proprio la memoria condivisa, dato che essa non richiede l'intervento del kernel per letture e scritture, consente collegamenti in sola lettura e garantisce una lettura che non altera i dati.

Il punto ii) ci ha spinto a riflettere sul fatto che l'esistenza di un unico segmento potesse introdurre un collo di bottiglia e ridurre le prestazioni, ragion per cui abbiamo deciso di partizionare il libro mastro in tre sezioni, ciascuna di dimensione

```
#define REG_PARTITION_SIZE ((SO_REGISTRY_SIZE + REG_PARTITION_COUNT - 1) / REG_PARTITION_COUNT)
```

Questa soluzione ha il vantaggio di distribuire letture e scritture su tre diversi segmenti, aspetto apprezzabile soprattutto con configurazioni dotate di un numero molto elevato di nodi e/o utenti.

Gli svantaggi principali sono due: il fatto che la somma delle dimensioni delle tre partizioni approssimi per eccesso quella specificata nella configurazione con SO_REGISTRY_SIZE ed un tempo di setup e terminazione del progetto più elevati, dovuti al fatto che occorre creare e deallocare più segmenti, e quindi eseguire più system call di tipo shmget, shmat, shmdt ed shmctl.

Grazie alla possibilità, offerta dalla specifica System V, di lavorare su set di semafori, l'implementazione dell'algoritmo di sincronizzazione non ha risentito del passaggio a tre partizioni.

Le transaction pools

Per immagazzinare le transazioni ricevute da un nodo abbiamo deciso di utilizzare una coda di messaggi per ciascuno di essi, ciascuna di dimensione `(sizeof(MsgTP) - sizeof(long)) * SO_TP_SIZE)` byte.
I messaggi scambiati su di essa sono di tipo MsgTP, una struct contente il tipo, ovvero il PID del nodo, e la transazione.
Questa coda viene creata dal master stesso subito dopo la fork, in modo da poterne memorizzare l'ID in un'apposita lista.
I nodi effettuano il collegamento dopo l'execve, per poi scollegarsi, senza eliminarla al termine della loro esecuzione.
Ad eliminare la coda sarà il master, dopo aver provveduto alla stampa delle transazioni rimanenti.
Gli utenti si collegano alla transaction pool del nodo al momento dell'invio di una transazione, sfruttando il fatto che la chiave della coda è stata generata con ftok, a partire da un file, il cui percorso è contenuto in MSGFILEPATH e dal pid del nodo possessore.
Una volta inviata la transazione, l'utente si scollega dalla coda.

Abbiamo optato per questa soluzione perchè ci consente di modellare l'idea intuitiva di sequenza di messaggi che si accumulano, formano proprio una coda, in attesa che le richieste degli utenti vengano servite dal master.
L'uso della coda di consente inoltre di processare le transazioni nell'ordine in cui arrivano e, soprattutto, fornisce automaticamente un servizio di sincronizzazione e controllo dello stato.
Tali servizi sono particolarmente utili, dato che consentono di bloccare l'esecuzione del nodo, senza eseguire un'attesa attiva, in assenza di transazioni da processare oppure di segnalare all'utente quando inviare una transazione sulla coda globale, con annesso decremento di SO_HOPS.
Qualora avessimo utilizzato un segmento di memoria condivisa avremmo dovuto implementare daccapo tutti questi servizi, cosa che avrebbe richiesto l'uso di segnali e system call come pause, rivelandosi quindi decisamente più articolata.
Inoltre la semantica distruttiva delle letture da questo oggetto IPC fa sì che l'eliminazione delle transazioni sia "automatica" e non richieda operazioni aggiuntive.

Gli svantaggi principali sono due: la necessità di modificare la dimensione di sistema consentita per le code di messaggi e la necessità di reinserire le transazioni in caso di errore.
In alcune configurazioni, come params_4.txt, e su determinati sistemi operativi, la formula sopra riportata per il calcolo della dimensione delle transaction pool restituisce un valore che eccede la dimensione massima definita dal sistema operativo per le code di messaggi.

Abbiamo inserito un controllo che approssimasse la dimensione con quella massima consentita ma, per una maggior aderenza ai requisiti progettuali e per apprezzare il pieno potenziale della simulazione, sarebbe preferibile innalzare tale limite.
Dopo aver consultato la man page della system call msgsnd, abbiamo appurato che per fare ciò, in una distribuzione Linux Ubuntu-based, è necessario modificare i seguenti file:
/proc/sys/kernel/msgmax e /proc/sys/kernel/msgmnb
Essi contengono, rispettivamente, la dimensione massima del campo text di un messaggio di un messaggio e la dimensione massima in byte della coda.
Valori di 2^13 e 2^20 byte dovrebbero essere sufficienti anche per le configurazioni con TP grandi e numero di transazioni generate elevato.

Avremmo anche potuto eseguire il progetto con permessi root, che consentono all'applicazione in uso di eccedere il limite di sistema, ma dato l'elevato rischio connesso all'uso di tali permessi abbiamo ritenuto migliore la soluzione illustrata.
Ad ogni modo, questa soluzione si rende necessaria solo per configurazioni molto particolari e il progetto è comunque in grado di funzionare anche in tali casi, grazie al controllo sopra citato, inserendo nel libro mastro un numero inferiore di blocchi.

Un esempio concreto del secondo problema menzionato in precedenza riguarda la scrittura di un blocco di transazione: in caso di fallimento della registrazione sul libro mastro è necessario reinserire le transazioni non processate correttamente nella transaction pool del nodo in questione.
Questo problema riguarda in realtà tutte le code usate nel processo, ma si può risolvere in maniera relativamente semplice (anche se non efficientissima), effettuando una msgsnd per scrivere nuovamente il messaggio sulla coda

Code globali
Preso atto della necessità dei processi di scambiare messaggi "di servizio" abbiamo deciso di utilizzare tre code di messaggi, a cui sono collegati tutti i processi.

Una prima coda, procQueue, contiene messaggi inviati dal master per comunicare al nodo l'elenco degli amici e dagli utenti e nodi al master per segnalare la loro terminazione.

La coda transQueue è invece adibita all'invio di transazioni che non hanno trovato collocazione nella transaction pool del nodo estratto (perché piena) e di messaggi per segnalare al sender il fallimento del processamento o dell'invio di una transazione.
Qualora quest'ultima coda sia piena quando si tenta di notificare la presenza di una transazione che non ha trovato posto su una transaction pool, il nodo manda una richiesta di creazione nuovo nodo al master.
Questa soluzione ci consente di ridurre l'intasamento della coda ed evitare che il nodo vi si blocchi, con conseguente rallentamento della simulazione.

Qualora anche la richiesta di creazione nuovo nodo non vada a buon fine viene segnalato al sender il fallimento della transazione.

Infine, nodeCreationQueue viene utilizzata per richiedere al master la creazione di un nuovo nodo, cui affidare il processamento di una transazione che non ha trovato collocazione, oppure per richiede ad un nodo esistente di aggiungere il nuovo processo alla sua lista amici.

In principio, avevamo deciso di utilizzare una sola coda su cui scambiare tutte queste tipologie di messaggi ma ci siamo immediatamente accorti che, specie con configurazioni con un numero di utenti e/o nodi elevato (come params_2.txt), il numero di messaggi scambiato era troppo elevato, tanto da provocare una rapida saturazione della coda.
Alla luce di quanto osservato, abbiamo deciso di introdurre le tre code di cui sopra, per una miglior distribuzione del carico.
Tale soluzione non è ovviamente esente da difetti, dato che richiede l'esecuzione di più system call per la lettura, scrittura, e setup che penalizzano il tempo di esecuzione dell'applicazione e si traduce in un codice più complesso.

Segmenti di memoria condivisa
Stante la necessità di nodi ed utenti di conoscere i PID degli altri processi, in modo da poter selezionare, nel caso dell'utente, il ricevente della transazione ed il nodo che si occupi del processamento e, nel caso del nodo, gli amici a cui mandare le transazioni abbiamo deciso di memorizzare, in due segmenti di memoria condivisa la lista dei pid e lo stato (attivo o terminato) di nodi ed utenti.
Tali segmenti sono creati e valorizzati dal master, mentre nodi ed utenti vi si collegano in sola lettura.

Abbiamo optato per tale soluzione poiché essa ci consente una facile condivisione delle informazioni, consente operazioni di lettura con semantica non distruttiva e, grazie al flag SHM_RDONLY, ci consente di garantire che solo il master, in quanto fonte più autorevole dell'applicazione, possa modificare le liste.

L'unico inconveniente di tale soluzione sta nel fatto che la lista dei nodi debba essere sovradimensionata, ad un valore di maxNumNode * sizeof(ProcListElem), dove maxNumNode è il numero massimo di nodi che possono comporre l'applicazione.
Se il numero di nodi creati durante tutto il corso della simulazione (ivi compresi quelli terminati in anticipo) supera questa soglia la simulazione viene terminata, perché lo spazio necessario per registrare le informazioni sui nodi è terminato.
Tale problema si manifesta soprattutto in configurazioni come params_5.txt, nelle quali viene effettuato un numero di richieste di creazione di nuovi nodi molto elevato.

Dato che il processo utente ha bisogno di conoscere il numero di nodi per poter estrarre quello a cui richiedere il processamento della transazione, abbiamo deciso di memorizzare tale informazione in un segmento di memoria.
Questa soluzione ci consente di rendere immediatamente disponibile qualsiasi modifica effettuata al contatore dal master in caso di terminazione o creazione di nodi.

Semafori per la sincronizzazione
Data la necessità di implementare la modifica dei dati condivisi tramite memoria principale tramite una regione critica abbiamo dovuto fornire una soluzione al problema che garantisse le proprietà di mutua esclusione, progresso ed attesa limitata.

Alcuni processi devono eseguire accessi in lettura ed altri in scrittura e tali cicli devono essere separati ed eseguiti in maniera alternata.
Le scritture devono essere in mutua esclusione, sia rispetto ad altri cicli di scrittura che a quelli di lettura, mentre più cicli di lettura possono essere eseguiti in contemporanea.
Entrambe le tipologie di cicli hanno la stessa priorità.
Data la tipologia del problema, abbiamo deciso a tal fine di implementare la soluzione fair al problema dei lettori e scrittori presentata dal professor Gunetti.

Questa soluzione evita che vi sia starvation di lettori o scrittori, perché le richieste di entrambi i tipi di processo vengono trattate con la stessa priorità.
Tale soluzione prevede di utilizzare, per ciascun dato condiviso, tre semafori ed una variabile.
Nel seguito, considereremo il caso dell'accesso alle partizioni del libro mastro.

Essendo tale struttura divisa in tre partizioni, ci avvaliamo della possibilità offerta dallo standard System V di creare set contenenti più semafori.
Nello specifico, utilizziamo tre insiemi da tre semafori ciascuno.
Il primo, wrPartSem, consente ai processi di riservare una o più partizioni per scrivere al loro interno nuovi blocchi.
Il master provvede a creare tale set prima dell'inizio della simulazione, inizializzandone tutti i semafori ad 1.
Il secondo, rdPartSem, consente ai processi di riservare le partizioni per leggerne i blocchi.
Anche i semafori di tale insieme sono inizializzati ad 1: in questo modo non imponiamo alcun ordine tra gli accessi in lettura e quelli in scrittura.
Il terzo ed ultimo set, mutexPartSem, contiene i semafori utilizzati per manipolare in mutua esclusione la variabile che conta il numero di processi intenti a leggere i dati di ciascuna partizioni.
Un vettore di tre interi, noReadersPartitions, consente di contare il numero di lettori in ciascuna partizione.
Ogni elemento del vettore è ovviamente inizializzato a zero, non essendoci, prima dell'inizio della simulazione alcun processo lettore.

Un processo che desidera scrivere dati su una partizione del libro mastro deve eseguire due wait, una sul semaforo rdPartSem e l'altro su wrPartSem della partizione di interesse.
La wait su rdPartSem contribuisce all'equità della soluzione, evitando che gli scrittori sopravanzino eventuali lettori, entrati in esecuzione dopo il primo lettore e prima del primo scrittore.
Con queste operazioni il processo attende la terminazione di eventuali cicli di lettura o scrittura già in corso e blocca gli scrittori ed i lettori schedulati dopo di lui e prima del termine della sua operazione di scrittura (tali processi troveranno il semaforo rdPartSem a 0, e quindi vedranno la loro esecuzione sospesa).
Al termine delle proprie operazioni, il processo rilascia la partizione con due signal.
La entry section del processo lettore inizia invece con una wait sull'rdPartSem della partizione contente i dati desiderati: tale operazione consente di attendere la terminazione di un ciclo di scrittura già in corso e di bloccare eventuali scrittori/lettori entrati in esecuzione successivamente.
Questa wait contribuisce a rendere la soluzione equa, perché eventuali scrittori successivi al primo e precedenti al primo lettore, non vengono sopravanzati dai lettori successivi, fino all'inizio del primo ciclo di lettura.
Successivamente occorre eseguire una wait su mutexPartSem per assicurarsi di manipolare il contatore dei lettori in mutua esclusione.
Tale variabile consente, qualora il lettore sia il primo, di bloccare eventuali scrittori futuri e, qualora sia l'ultimo, di segnalare l'inizio di un nuovo ciclo di scrittura.
In sostanza, consente al primo lettore di riservare l'accesso al file in lettura per sé e per tutti i lettori successivi, fino alla fine dell'ultimo ciclo di lettura.
Dato che le letture non modificano il contenuto del libro mastro è possibile eseguirle contemporaneamente perché non c'è alcun rischio di race condition.
Per questo motivo, dopo aver manipolato la variabile condivisa, il processo lettore esegue una signal su rdPartSem, in modo che eventuali lettori successivi possano procedere con la loro esecuzione.

Inizio sincronizzato della simulazione
Per garantire che la simulazione abbia inizio allo stesso istante per tutti i processi abbiamo deciso di sfruttare la possibilità di eseguire una attesa dello zero su un semaforo, offerta dallo standard System V.
Per implementare questo meccanismo utilizziamo il semaforo fairStartSem, che viene inizializzato ad un valore pari a SO_USERS_NUM + SO_NODES_NUM + 1.
Ogni processo, dopo aver eseguito alcune operazioni di setup, come la lettura degli amici ed il collegamento agli oggetti IPC, esegue la wait for zero, mettendosi in attesa dell'inizio della simulazione.
Il master provvederà a decrementare il valore del semaforo di un'unità ad ogni processo creato e, dopo aver inizializzato i budget e gli amici di ogni nodo, eseguirà un ulteriore decremento.
Tale operazione porterà il valore del semaforo a zero dando finalmente inizio alla simulazione.

Bug Noti
Quando vengono creati molti nodi, ovvero in configurazioni come params_5, la creazione delle transaction pool di alcuni nuovi nodi fallisce, perché non sempre la ftok riesce a creare una chiave univoca.
Il master è in grado di rilevare questo errore, terminando anzitempo l'esecuzione del nuovo nodo.
In configurazioni come params_4 il numero molto elevato di transazioni generate e di messaggi scambiati potrebbe causare la saturazione delle code, specie con limiti di sistema bassi.
In tal caso gli utenti procederanno sempre più lentamente, fino a bloccarsi sulle code, prima che la simulazione termini per fine tempo.

Passaggio Amici
Prima di inviare i nodi amici, questi devono essere estratti tra quelli disponibili. La funzione "estrai" (presente nel master) implementa l'algoritmo che permette di estrarre SO_FRIENDS_NUM nodi amici per ogni nodo.
L'idea di base è che per un nodo non può essere estratto come amico sé stesso; inoltre la lista non può comprendere più volte lo stesso nodo amico.
Per realizzare quanto appena descritto, alla funzione estrai viene passato il parametro k, il quale rappresenta l'indice (rispetto alla lista di nodi nodesList) del nodo che chiama la funzione.
Si esegue quindi un ciclo, che finirà una volta estratti SO_FRIENDS_NUM nodi, in cui come prima operazione, estraiamo un indice (compreso tra 0 e il numero di nodi disponibili), che controlliamo essere diverso da k: in caso sia uguale estraiamo un altro valore, altrimenti continuiamo l'esecuzione dell'algoritmo. Estratto l'indice dobbiamo verificare che questo non sia già presente nella lista. Scorriamo gli indici già estratti: se incontriamo lo stesso valore ci fermiamo e procediamo all'estrazione di un nuovo numero, altrimenti lo inseriamo nella lista. Continuiamo così fino a quando non abbiamo ottenuto una lista di SO_FRIENDS_NUM nodi amici.
A questo punto, il master invia, per ciascun nodo, tanti messaggi di tipo FRIENDINIT sulla coda procQueue quanti sono gli amici.

I nodi provvedono poi, subito dopo l'execve, a prelevare tali messaggi dalla coda. I principali vantaggi offerti da tale soluzione sono le già menzionate funzionalità di sincronizzazione offerte dal sistema operativo ed il fatto che si debba fare ricorso ad un solo oggetto IPC per completare l'operazione.

Lo svantaggio sta nel fatto che, in configurazioni con molti nodi, la coda potrebbe riempirsi, rallentando l'esecuzione di master e nodi.

Rilevamento Errori
Abbiamo deciso di terminare la simulazione, richiamando l'apposito handler, nel caso in cui si verifichi un errore nelle operazioni di setup, come la creazione di nodi ed utenti, la lettura degli amici e l'inizializzazione del timer per la durata della simulazione.

Si tratta di una soluzione restrittiva, che evita però inconsistenze e consente una piena aderenza ai parametri della configurazione.
Il fallimento di una di queste operazioni impedirebbe al processo di proseguire, riducendo le capacità della simulazione.

Generazione di transazioni su richiesta
Il segnale da inviare ad un processo utente qualsiasi per ottenere la generazione di una transazione è SIGUSR2.

Condivisione parametri di configurazione
I parametri di configurazione vengono letti da file dal master e caricati in omonime variabili d'ambiente, per tale motivo, il file in questione deve contenere righe della forma nome=valore.
L'ambiente contenente tali variabili viene passato come argomento della execle, scelta appositamente, tra le varie API della execve proprio per tale possibilità.
I processi nodo ed utente eseguono poi una lettura di tali variabili all'inizio della loro esecuzione.

Questa soluzione evita di dover rileggere il file durante l'esecuzione di ogni processo: essendo questa un'operazione molto costosa in termini computazionali, cioè si traduce in una riduzione del tempo di esecuzione dei processi.
Inoltre, adottando questa soluzione non dobbiamo allocare alcun oggetto IPC.
La soluzione è adatta la contesto dato il ridotto numero di parametri da scambiare tra i vari processi.

Terminazione simulazione
Nella nostra applicazione soltanto il master può ordinare, tramite l'apposito segnale SIGUSR1, la terminazione della simulazione.
Dato che un primo invio di tale segnale, tramite la system call kill, potrebbe fallire abbiamo previsto un apposito meccanismo per tentare ripetutamente la terminazione.

Nel caso in cui il segnale venga consegnato con successo, il master si mette in attesa della terminazione effettiva dei figli, in modo da poter procedere senza errori nella deallocazione degli oggetti IPC e nella stampa delle informazioni di riepilogo.
Qualora non si riesca a consegnare il segnale entro il numero di tentativi previsto il master termina, deallocando comunque le facilities IPC.

Qualora un nodo rilevi l'esaurimento dello spazio disponibile nel libro mastro esso provvederà ad inviare il segnale SIGUSR1 al master, il quale, in risposta a tale segnale, darà inizio alla procedura di fine simulazione.
Dopo aver segnalato l'esaurimento di spazio al master, questo eseguirà la system call pause, in modo da attendere che il master invii il segnale di fine simulazione.

Qualora il l'invio del segnale fallisca, il nodo esegue comunque la pause: così facendo il suo PCB verrà spostato dalla coda di ready, aumentando la probabilità che il master venga schedulato e possa rilevare, nell'ambito della verifica compiuta ad ogni iterazione del suo ciclo di vita oppure perché notificato da un altro nodo, l'assenza di spazio sul libro mastro ed ordinare la fine della simulazione.
In questo modo evitiamo di implementare un meccanismo di retry per l'invio del segnale.

Nel nostro progetto la simulazione può terminare per diverse ragioni:

- Terminazione di tutti i nodi
- Terminazione di tutti gli utenti
- Riempimento libro mastro
- Scadenza timer simulazione
- Errori irrecuperabili (esempio: fallimento creazione transaction pool nodo)
- Esaurimento risorse per creazione nuovi nodi
- Invio del segnale di interrupt da tastiera (CTRL + C)

Stampa transazioni non inserite
Abbiamo osservato che la simulazione potrebbe terminare mentre un nodo sta creando un nuovo blocco di transazioni, ma prima che possa registrarlo nel libro mastro.
In tale situazione, si perderebbe traccia delle transazioni in questione, non essendo esse ne presenti nella transaction pool del nodo (perché estratte per comporre il blocco) ne registrate nel libro mastro,
Per porre rimedio a questo problema, abbiamo deciso di stampare a video, come parte del report di fine esecuzione, anche l'elenco delle transazioni presenti nel blocco che ciascun nodo stava elaborando quando interrotto.

Implementazione attesa non attiva
Per realizzare questo punto abbiamo utilizzato la nanosleep.
Questa soluzione consente di impostare la durata dell'attesa con la granularità richiesta (nanosecondi) e consente una migliore rilevazione degli errori.
Nanosleep infatti imposta errno e restituisce -1 nel caso in cui venga interrotta da un segnale, cosa che invece la sleep non fa, limitandosi a restituire il valore del timer.
Infine, è preferibile non combinare sleep e alarm (che utilizziamo per il timer che registra la durata della simulazione) perché su alcuni sistemi l'una potrebbe essere implementata usando l'altra.

Algoritmo ottimizzato di stampa del budget dei processi
Per la stampa dei budget degli utenti e dei nodi abbiamo utilizzato un algoritmo ottimizzato che sfrutta l'immutabilità del libro mastro.
Infatti, una volta che un blocco di transazioni viene inserito nel libro mastro, non viene più modificato; questo può essere utilizzato per ridurre la complessità del calcolo dei budget dei processi.

Avendo suddiviso il libro mastro in 3 partizioni, l'algoritmo sviluppato, alla prima esecuzione, inizia a leggere le transazioni dal primo blocco di ogni partizione e, scorrendo le transazioni nel blocco, aggiorna il budget dei processi sender e receiver della transazione (tranne nel caso in cui la transazione ha sender -1, che corrisponde alla transazione di reward del nodo e quindi dobbiamo aggiornare solo il budget del receiver). Il budget di ogni processo, per essere memorizzato e successivamente aggiornato, viene memorizzato in un array di struct così definite:

```
typedef struct proc_budget
{
      pid_t proc_pid;
      float budget;
      int p_type /* type of process: 0 if user, 1 if node */
} proc_budget;
```

Dopo aver letto tutte le transazioni in tutti i blocchi correntemente nel libro mastro, per ogni partizione viene salvato l'indice del blocco a cui ci si è fermati (l'ultimo che in quel momento è presente in quella partizione, sappiamo che tanto in quel momento siamo in sezione critica e nessun nodo può aggiungere un blocco). Questo servirà alla successiva esecuzione dell'algoritmo per riprendere a leggere le transazioni dal blocco a cui ci si era fermati all'esecuzione precedente (le transazioni dei blocchi precedenti non possono subire modifiche e quindi sono già state conteggiate nel budget attuale dei processi). Quello che faremo sarà aggiornare il budget precedentemente calcolato con le nuove transazioni inserite nel libro mastro, senza dover ricalcolare tutti i budget daccapo.
L'array di struct utilizzato per memorizzare i budget è ordinato in modo crescente in base al budget dei processi (quindi in prima posizione ci sarà il processo con budget minore e in ultima posizione quello con budget maggiore). Questo permette una stampa dei budget dei processi ordinati e, nel caso in cui il numero di processi sia troppo elevato, rende più veloce la stampa dei processi con budget minimo e massimo (non è necessario un algoritmo di ricerca, basta leggere il budget dei processi in prima e ultima posizione nell'array dei budget).

Generazione numeri casuali

Per generare numeri casuali abbiamo deciso di non utilizzare srand, ma di implementare una soluzione che garantisce una maggior variabilità dei numeri generati.
Tale soluzione prevede di recuperare, tramite clock_gettime, il numero di nanosecondi trascorsi dal 01/01/1970, misurato con CLOCK_REALTIME.
Calcoliamo poi il resto della divisione tra tale valore ed il massimo dell'intervallo più uno, sommando eventualmente il minimo dell'intervallo, qualora questo sia maggiore di 0.
In questo modo otteniamo un valore compreso tra [min, max].
Questa soluzione riduce il rischio di ottenere più volte lo stesso valore, soprattutto quando la generazione viene effettuata molto frequentemente in un breve intervallo di tempo.

Rallentamento esecuzione utente

Abbiamo osservato che l'esecuzione dell'utente risultava essere troppo veloce rispetto a quella dei nodi.
Questo, soprattutto in alcune esecuzioni, causava la saturazione delle code globali e delle transaction pool, con conseguente blocco dell'esecuzione di nodi ed utente.
Per risolvere questo problema abbiamo dovuto rallentare l'esecuzione dell'utente, inserendo una sleep, per sospendere l'esecuzione dell'utente per un secondo.

Code di messaggi: evitare la race condition

Per evitare che i processi si sottraggano vicendevolmente le transazioni spedite sulle code globali abbiamo deciso di sfruttare la relazione di parentela esistente tra tutti i processi della simulazione e l'univocità del PID, garantita dal sistema operativo
Abbiamo infatti utilizzato come tipo del messaggio il PID del nodo destinatario, sicuramente noto perché reso disponibile dal master, che conosce, essendo il padre, tutti i processi della simulazione, oppure perché prelevato tramite la system call getppid.