Relazione Progetto Sistemi Operativi 2023/2024

Emanuele Di Maggio (883368) [emanuele.dimaggio@edu.unito.it]

1 Introduzione

Il progetto di quest'anno riguarda la simulazione di atomi che producono energia innescando una reazione a catena. Lo scopo del progetto è ideare una simulazione dove:

- Ogni atomo può produrre energia;
- Gli atomi sono influenzati da altri strumenti (processi);
- Le informazioni e le statistiche vengono stampate per mostrare ciò che sta avvenendo.

*Tutti i nomi all'interno del progetto (fatta eccezione per le variabili in maiuscolo del progetto) sono in **inglese**, anche quelli dei processi, ma i commenti e le stampe sono in **italiano**.*
Nella cartella sono presenti anche i file di default di Visual Studio Code.

2 Avviare il progetto

Il progetto è diviso in quattro cartelle che si trovano all'interno della cartella Progetto SO:

- bin → contenente i file eseguibili;
- build → contenente i file oggetto .o;
- data → contenente file utili alla simulazione;
- src → contenente il codice del progetto.

Per eseguire la simulazione bisogna seguire questi passi dal terminale:

1. Eseguire (nella cartella Progetto SO) **make clean** per eliminare precedenti file di compilazione e successivamente **make all** per compilare il progetto;
2. Spostarsi all'interno della cartella **bin**;
3. Eseguire **./starter**.

Una volta eseguito **Starter** verranno mostrate varie opzioni di scelta per le possibili simulazioni e la scelta se si vuole il processo **inhibitor** oppure no.
Io ho inserito quattro simulazioni per ogni possibile terminazione e una quinta in aggiunta con valori liberi. La quinta possibile simulazione legge i dati dal file contenuto in **data** chiamato **filedata.txt**. Al suo interno si possono modificare i valori numerici per scegliere i dati per la simulazione senza bisogno di ricompilare ogni volta tutto il codice.

3 Descrizione progetto

Una volta scelta la simulazione tutti i dati verranno passati al processo master che inizializzerà tutte le strutture utili e avvierà la simulazione.

3.0.1 Funzioni di Supporto

Nel codice del progetto sono presenti due file chiamati **functions.h** e **functions.c** usati per implementare alcune funzioni utili e per rendere il codice più leggibile, inoltre il codice è commentato.

3.0.2 Gestione Casualità

La casualità degli eventi è gestita tramite la funzione **rand()**, della libreria **stdlib.h**, in tutto il codice. Viene usata per:

- Calcolare il numero atomico di un processo (Master e Feeder);
- Calcolare il numero di messaggi che activator dovrà inviare (Activator);
- Calcolare il nuovo numero atomico dell'atomo padre dopo la scissione (Atom);
- Calcolare la probabilità che un'attivazione non venga eseguita (Inhibitor);
- Calcolare la quantità di energia da sottrarre (Inhibitor)

3.1 MASTER

Il processo master ricevuti i dati injzializza le memorie condivise, i semafori e le code di messaggi utili per la simulazione. Successivamente si occupa di generare i figli:

- **master-child** è il processo di supporto a master;
- **atom** sono i processi Atomo;
- **activator** è il processo Attivatore;
- **feeder** è il processo Alimentazione;
- **inhbitor** è il processo Inibitore.

Una volta generati i figli, ognuno di loro (tranne master-child) fa una **execve**, master avvia la simulazione ed entra in un ciclo while dove incomincia a stampare le statistiche ogni secondo.

3.1.1 Master-child

Il processo master-child è un figlio di supporto a master che si occupa di aggiornare le statistiche contenute in memoria condivisa. Inizializzate le sue strutture entra in un ciclo while in attesa di messaggi da una coda. I messaggi che riceve contengono i dati con cui aggiornare le statistiche.

3.1.2 Note Master

Tra le varie memorie condivise generate da master, la memoria condivisa **shm-sim-data** contiene tutte le impostazioni della simulazione e il suo **Id** e il suo **semaforo** vengono passati come parametri a ogni processo eseguito tramite **execve**.

3.2 ATOM

I processi atom all'inizio della simulazione vengono generati da master e successivamente verranno generati dal processo feeder e dagli atom stessi. Ogni processo atom notifica tramite coda di messaggi a master-child la loro creazione ed entra in un ciclo while attendendo un messaggio da parte di activator che lo faccia "scindere". Ricevuto il messaggio, atom entra nella funzione **split** dove genera un nuovo processo atom. Prima che il figlio generato possa eseguire la execve, calcola un nuovo **numero atomico** da assegnargli e glielo invia tramite **pipe**. Una volta calcolato il nuovo numero atomico del figlio, il padre calcola l'**energia** prodotta e la invia a master-child o a inhbitor (se attivo) tramite coda di messaggi. Infine si rimette in attesa di nuovi messaggi di "scissione".

3.2.1 Note Atom

- Se la **fork** del figlio dovesse **fallire** invia un segnale a master che terminerà in **meltdown**;
- Se il processo atomo riceve un messaggio di "scissione", ma il suo numero atomico è minore di **MIN-N-ATOMICO**, il processo invia un messaggio a master-child e termina.

3.3 ACTIVATOR

Il processo activator inizializza le sue strutture ed entra in un ciclo while. All'interno del ciclo si occupa di fare diverse cose:

1. Ottiene il **numero totale** di processi atom attivi dalla memoria condivisa aggiornata da master-child;

2. Calcola un **numero casuale** (saranno le attivazioni) da 1 al numero totale di atom attivi e fa un ciclo for inviando un messaggio nella coda di messaggi alla quale hanno accesso i processi atom. L'atom che riceve il messaggio fa una "scissione";

3. Invia il numero di **attivazioni** a master-child in modo che possa aggiornare le sue statistiche.

4. Infine fa una **nanosleep** per **STEP-ATTIVATORE** e rincomincia dall'inizio.

3.3.1 Note Activator

Se il processo inhibitor è stato richiesto all'inizio della simulazione, tra l'invio di un messaggio agli atom e l'altro, verifica se il processo inhibitor è attivo. Se lo è, controlla una flag in memoria condivisa, aggiornata da inhibitor, che indica se inviare o meno il messaggio di "scissione".

3.4 FEEDER

Il processo feeder inizializza le sua strutture ed entra in un ciclo while. All'interno del ciclo ogni **STEP-ALIMENTAZIONE** genera **N-NUOVI-ATOMI** processi atom tramite fork calcolando per ognuno di essi il numero atomico.

3.4.1 Note Feeder

Se la fork di un processo atom dovesse fallire invia un segnale al processo master che terminerà in **meltdown**.

3.5 INHIBITOR

Il processo inhibitor inizializza le sue strutture e successivamente fa la fork di un nuovo processo figlio ausiliario **inhibitor-child**.
Il figlio si occupa di aggiornare la flag **split-probability** contenuta in memoria condivisa che decide se activator debba inviare o meno il messaggio di "scissione" agli atom. Il padre, invece, entra in un ciclo e si mette in lettura in una coda di messaggi ricevendo l'energia prodotta dagli atom. Una volta prelevata una quantità casuale dall'energia ricevuta, invia sia la nuova energia sia quella sottratta, tramite coda di messaggi, a master-child che aggiornerà le statistiche.

3.5.1 Note Inhibitor

Il meccanismo di pausa e ripresa del processo inhibitor (con annesso figlio) funziona tramite il segnale **SIGTSTP** che si invia al terminale usando la combinazione di tasti **Ctrl + z**. Il processo Inhibitor gestisce questo segnale usando la **sigaction** e funziona in questo modo:

1. Inhibitor riceve il segnale **SIGTSTP** entra nell'handler **inhibitor-pause-and-stop-handler**;

2. Verifica la flag **flag-start-and-stop**:

    - Se vale 1 la imposta a 0, mette in pausa il figlio, rilascia eventuali semafori ottenuti prima della ricezione del segnale, aggiorna la flag che indica che non è attivo e va in **pause()** attendendo il prossimo segnale;
    - se vale 0 la imposta a 1, fa ripartire il figlio, aggiorna la flag che indica che è attivo e riprende da dove era rimasto.

4 Gestione degli errori, segnali, pulizia risorse e terminazione

Nel file **functions.h** è contenuta una macro che gestisce gli errori in base al contenuto di **errno**, indicando il file e la riga nella quale è avvenuto l'errore. Siccome **errno** in alcuni casi è usato attivamente per la simulazione, la macro non viene impiegata per ogni tipo di struttura che può causare errore.
Alcuni segnali vengono gestiti dagli handler. Tutti i processi implementano una **sigaction** e gestiscono il segnale **SIGTERM** aspettando la terminazione di tutti i processi figli e terminando poi loro stessi. Il master gestisce altri segnali utili per la simulazione nel suo handler e pulisce tutte le risorse allocate prima di terminare. Le possibili terminazioni vengono gestite dal master e verificate in questo modo:

- **Timeout** → viene impostato un **alarm** con il valore di **SIM-DURATION** all'inizio della simulazione. Terminato il tempo, il master riceve il segnale **SIGALRM** e termina la simulazione;
- **Explode e Blackout** → durante ogni stampa delle statistiche il master verifica se l'energia totale prodotta supera **ENERGY-EXPLODE-THRESHOLD** o diventa negativa. Se avviene termina la simulazione.
- **Meltdown** → ogni volta che viene fatta una fork se questa fallisce, viene inviato un segnale a master che terminerà la simulazione.