Relazione Progetto Sistemi Operativi

Ferrando Damillano Filippo, Nicosia Francesco, Noto Nicola

Gennaio 2024

Indice

1 Membri del Gruppo

Ferrando Damillano Filippo

- Matricola: 1043397
- Turno: T1
- Email: filippo.ferrandodami@edu.unito.it

Nicosia Francesco

- Matricola: 1030308
- Turno: T4
- Email: francesco.nicosia@edu.unito.it

Noto Nicola

- Matricola: 1055839
- Turno: T3
- Email: nicola.noto@edu.unito.it

2 Scelte progettuali generali

Alla base dello sviluppo del progetto vi è la conoscenza di dover preparare un programma nel quale diversi processi devono sincronizzarsi e comunicare tra loro. Abbiamo utilizzato diversi metodi come i semfafori per gestire la sincronizzazione, ma anche per la gestione di letture e scritture sulla shared memory; Per quanto riguarda invece la comunicazione tra processi abbiamo utilizzato sia code che messaggi che i semafori stessi. Per non cadere nella banalità e settare le variabili necessarie all'esecuzione della simulazione con un file, abbiamo scelto le variabili d'ambiente, impostate attraverso un'apposito script. Grazie a questo metodo siamo in grado di impostare delle soglie di meltdown adhoc per ogni computer su cui viene fatto eseguire.

3 Descrizione funzionalità dei vari processi

3.1 Master

Il processo Master gestisce diversi handler per controllare i segnali provenienti da **alimentatore** e **atomo** in caso di timeout, blackout, explode o meltdown e gestire la terminazione della simulazione sia con una killall che attraverso un script che gestisce anche la rimozione delle risorse IPC.

Successivamente, nella funzione principale, andrà a recuperare tutte le variabili necessarie a far funzionare il progetto tramite funzioni definite nel file *generalLib.h*

Master si occupa dell'allocazione delle varie struct:

- L'allocazione in memoria condivisa della struct stats, in cui scrivere i parametri letti precedentemente da environment, così da poter condividere i parametri generali con tutti gli altri processi.
- La struct per la **nanosleep**, che servirà a impostare dei delay in ordine dei nanosecondi per evitare problemi di partenza con i processi e gestre i tempi di proessi come attivatore o alimentatore
- Una struct per la coda di messaggi usata da **inibitore** e **atomo**
- l'allocazione in memoria condivisa di una struct che useranno solamente gli atomi.

Finite le struct, passerà a creare i vari processi come Attivatore, Alimentatore, Inibitore e i processi Atomo. Una volta controllato di aver tutti i processi correttamente sincronizzati, darà il via alla simulazione. Ogni secondo il master si occuperà delle stampe dei valori dei processi, e in caso di explode (Creazione di più energia rispetto a quella gestibile) o di blackout (in caso di energia insufficiente rispetto a quella che si vuole prelevare), terminerà la simulazione, premurandosi di eliminare i restanti processi zombie se presenti. La gestione delle risorse IPC avviene in modo atipico salvando i vari id di semafori, code di messaggi o segmenti di shared memory in file creati ad-hoc e che lo script .sh *killer.sh* di occuperà di eliminare (deallocando le risorse) un volta terminata l'esecuzione della simulazione in qualsiasi caso.

3.2 Atomo

Il processo Atomo inizierà ricevendo tutte le informazioni che gli servono per capire quando ha il via libera per forkare e tenendosi in contatto con l'inibitore che limiterà il numero di scissioni e assorbirà parte dell'energia rilasciata nella creazione dei nuovi atomi. Se l'atomo viene creato dal master, dovrà aspettare il semaforo di start, in caso contrario, l'alimentatore setterà la variabile "bypass" a un numero diverso da 0 che permetterà all'atomo di ignorare il semaforo e procedere al controllo per la scissione. Dopo il via libera dal semaforo, verrà calcolato il numero atomico, numero che dovrà essere compreso nella soglia impostata precedentemente. Se non risulta, procederà all'eliminazione immediata dell'atomo. Dopo il controllo del numero atomico, proseguirà alla fork dell'atomo, incrementando il contatore degli atomi. Il tutto avrà un apposito caso di meltdown. Dopo la fork, riceveremo il messaggio di controllo dall'inibitore che, se di tipo 1, procederà all'eliminazione dell'atomo.

3.3 Alimentatore

Il processo Alimentatore ogni *STEP-ALIMENTATORE* nanosecondi, verranno creati n atomi, cioè il "combustibile", assegnandoli un numero atomico. Per comunicare al processo Atomo il suo n-atomico utilizzeremo un buffer da passare come argomento alla execve. Atomo prima di continuare il ciclo di esecuzione controlla l'esistenza di zombie, eliminandoli se trovati. Se non trova nulla, passerà alla creazione dei nuovi n atomi, con controllo di un eventuale meltdown.

3.4 Attivatore

L'attivatore ha un compito semplice quanto fondamentale, comunicare agli n atomi la necessità di una scissione. La prima istruzione eseguita, infatti, è la gestione dei file risorse ipc, in modo da ottenere una giusta comunicazione. Successivamente, inizializzerà il numero di attivazioni che deve eseguire, sincronizzandosi con i restanti processi e riservando il semaforo in memoria per trasmettere il numero di attivazioni da eseguire.

3.5 Inibitore

Infine, il processo Inibitore ha il compito di:

- Assorbire parte dell'energia prodotta dalla scissione degli atomi, andando a ridurre la quantità di energia sprigionata.
- Limitare il numero di scissioni rendendo l'operazione probabilistica.

Tramite delle variabili di stato, l'inibitore invierà messaggi all'attivatore, comunicando se deve procedere normalmente oppure dicendogli di fermarsi. Tramite queste variabili di stato, si andrà ad accendere o a spegnere l'inibitore, limitandone il funzionamento. Create le struct necessarie, come quella per la shared memory, andremo a stabilire la coda di messaggi per comunicare con il processo Atomo (tramite id). Dopo aver sincronizzato l'inibitore con i restanti processi, tramite semaforo, andremo a gestire l'explode tramite un while e una serie di if, prelevando e rilasciando energia, controllando di non superare il limite massimo impostato, e gestiremo anche il meltdown che, tramite una serie di if, andrà a "mangiare" un tot di Atomi in base al numero corrente di processi. Se non sono presenti Atomi da mangiare, non invierà messaggi.