Progetto sistemi operativi: CORE WARS

1. Introduzione

Core Wars è un gioco di programmazione in cui due programmi assembly cercano di distruggersi uno con l'altro nella memoria di un computer simulato. I programmi (chiamati *warriors*) sono scritti in un linguaggio assembly speciale chiamato *Redcode*, e sono eseguiti da un processore simulato chiamato MARS. Il progetto consiste nello scrivere un sistema operativo per una macchina dotata di processore MARS. Per semplificare il lavoro, il sistema operativo e il codice delle applicazioni sono fisicamente separati e scritti in linguaggi diversi. Scriveremo il codice del sistema operativo in un linguaggio di alto livello, dotato di primitive di programmazione concorente (ad esempio, Java o altri linguaggi), mentre i programmi che "vivono" all'interno della memoria saranno scritti in Redcode. Ovviamente, questo non rispecchia il normale funzionamento di un sistema operativo, che convive con il codice nella memoria della macchina e deve implementare degli opportuni meccanismi di protezione; tuttavia, questo "espediente" permetterà di limitare la complessità del problema.

Esistono dei veri e propri tornei basati su Core Wars; per semplicità, e per meglio aderire allo spirto dell'esercitazione, non utilizzeremo il linguaggio assembly previsto da Redcode, ma una sua variante.

Gli obiettivi del progetto sono due: da un lato, provare a scrivere un "micro" sistema operativo per una macchina virtuale. Dall'altro, utilizzare i meccanismi di sincronizzazione per gestire l'esistenza di thread multipli, quali uno o più processori e un meccanismo di DMA.

2. Architettura

La macchina che prenderemo in considerazione è una macchina così costituita: un processore MARS, una memoria circolare di dimensione fissa e un meccanismo di DMA in grado di leggere "programmi" da uno storage esterno. Compito del sistema operativo è quello di gestire il processore, il timer, il sistema DMA e il disco. In particolare, la comunicazione tra sistema operativo e processore verrà gestito tramite un meccanismo di interrupt.

2.1 Memoria

La memoria di una macchina MARS (il core) è costituita da un numero finito di celle organizzate in un array circolare: una lunga catena di celle in cui l'ultima è seguita dalla prima cella. Nei programmi in Redcode, tutti i riferimenti alla memoria sono relativi (mai assoluti). Ovvero, un'istruzione può fare riferimento alle celle che si trovano *x* posizioni in avanti o indietro rispetto alla cella in cui si trova l'istruzione, mai alla cella nella posizione *x*. Essendo la memoria circolare, se stiamo eseguendo un'istruzione nell'ultima cella e questa istruzione fa riferimento alla cella successiva, in realtà fa riferimento alla prima cella della memoria. Se la dimensione della memoria è n ed un'istruzione fa riferimento ad una cella che è n celle avanti o indietro, in realtà fa riferimento a se stesso.

Ogni cella è suddivisa in tre campi:

- opcode: codice dell'istruzione
- operando A: operando A
- operando B: operando B

2.2 Processore

Il processore preso in considerazione contiene 32 registri di uso generale a 32 bit ed un program counter. Il processore è in grado di eseguire le seguenti istruzioni:

- Opcode: 0 - **IMM Rx, valore**

  Carica il valore immediato "valore" nel registro Rx

- Opcode: 1 - **COPY (Rx), (Ry)**

  Copia il contenuto della cella indirizzata (in modo relativo) da Rx nel contenuto della cella indirizzata (in modo relativo) da Ry

- Opcode: 2 - **LOADA Rx, (Ry)**

  Carica il contenuto dell'operando A della cella indirizzata (in modo relativo) da Ry nel registro Rx

- Opcode: 3 – **LOADB Rx, (Ry)**

  Carica il contenuto dell'operando B della cella indirizzata (in modo relativo) da Ry nel registro Rx

- Opcode: 4 - **STOREA Rx, (Ry)**

  Salva il registro Rx nell'operando A della cella indirizzata (in modo relativo) da Ry

- Opcode: 5 - **STOREB Rx, (Ry)**

  Salva il registro Rx nell'operando B della cella indirizzata (in modo relativo) da Ry

- Opcode: 6 - **MOVE Rx, Ry**

  Sposta il contenuto del registro Rx nel registro Ry

- Opcode: 7 - **ADD Rx, Ry**

  Esegue l'operazione Rx+Ry e mette il risultato in Ry

- Opcode: 8 - **SUB Rx, Ry**

  Esegue l'operazione Rx-Ry e mette il risultato in Ry

- Opcode: 9 - **AND Rx, Ry**

  Esegue l'operazione Rx AND Ry e mette il risultato in Ry

- Opcode: 10 - **OR Rx, Ry**

  Esegue l'operazione Rx OR Ry e mette il risultato in Ry

- Opcode: 11 - **NOT Rx, Ry**

  Esegue l'operazione Rx OR Ry e mette il risultato in Ry

- Opcode: 12 - **JUMP Ry**

  Salta all'indirizzo (relativo) contenuto in Ry

- Opcode: 13 – **BGT Rx, Ry**

  Se Rx > 0, salta all'indirizzo (relativo) contenuto in Ry

- Opcode: 14 – **BLT Rx, Ry**

Se Rx < 0, salta all'indirizzo (relativo) contenuto in Ry

- Opcode: 15 – **BEQ Rx, Ry**

  Se Rx = 0, salta all'indirizzo (relativo) contenuto in Ry

- Opcode: 16 – **BNE Rx, Ry**

  Se Rx = 0, salta all'indirizzo (relativo) contenuto in Ry

2.3 Interruzioni

Il processore dispone di un'ultima istruzione, la più importante, che genera un interrupt software; ovvero, una system call:

- Opcode: 32 – **SYSCALL**

  Genera un interrupt il cui IRQN è 0

Un interrupt deve essere gestito da un'opportuno interrupt handler per conto del sistema operativo. Le convenzioni per la selezione della system call opportuna e per il passaggio di parametri fra i warrior e il sistema operativo sono lasciati ai gruppi. Per esempio, una system call di tipo "starthread" può aver bisogno di un parametro, ovvero l'indirizzo (relativo) di partenza del nuovo thread. Per esempio, questo indirizzo potrebbe essere collocato nel registro 25 (per convenzione del sistema operativo). L'indice della system call, ad esempio 0, può essere collocato nel registro 24.

Ulteriori interrupt che possono essere generati:

- IRQN 1: Generato dal timer allo scadere di un quanto temporale
- IRQN 2: Generato dal processore in occasione di una istruzione non valida (opcode diverso da quelli presenti nel sistema).
- IRQN 3: Generato dal meccanismo di DMA quando ha completato un'operazione di lettura/scrittura

Per ognuno di questi interrupt, il sistema operativo dovrà fornire un interrupt handler opportuno.

2.3 Timer

Il timer è il meccanismo che gestisce la sincronizzazione tra i vari elementi del sistema. Compito del timer è quello di emettere tick a tutti i componenti del sistema, per fare in modo che siano sincronizzati (tramite un metodo tick, contenuto nell'interfaccia TimerUser). In particolare, emette un tick al processore e al DMA. I componenti del sistema completano il task che devono gestire durante il tick (ad esempio: il processore esegue un'istruzione, il DMA effettua la copia delle informazioni nella memoria, etc.). Quando il task è terminato, i componenti notificano il timer di questo fatto tramite un'istruzione tickCompleted. Quando il timer riceve un invocazione tickCompleted per ognuno dei componenti nel sistema, emette il tick successivo.

Inoltre, tramite il timer è possibile settare un timeout (normalmente corrispondente al quanto temporale del meccanismo di schedulazione) e fare in modo che un interrupt venga generato allo scadere di questo quanto temporale, in modo che il sistema operativo possa prendere di nuovo il controllo della macchina e schedulare un altro processo.

2.4 Storage

La macchina MARS è dotato di uno storage, da cui è possibile leggere file contenenti programmi e/o dati. Lo storage è organizzato in un certo numero di file, identificati da un indice numerico. Ogni file è associato ad un process id; solo i processi caratterizzati dal process ID corretto possono leggere i propri file. Per leggere/scrivere un file dallo storage, è necessario identificare il numero del file, l'offset (la cella di partenza da cui iniziare a leggere) e il numero di celle da leggere/scrivere. Nel caso della lettura, le celle lette verranno restituite come array. Nel caso di scrittura, le celle scritte verranno passate come array di celle.

2.5 Il DMA

Il DMA opera in concorrenza con il processore e permette al sistema di copiare informazioni dallo storage alla memoria e viceversa. Il DMA è comandato dal sistema operativo, che invoca un metodo indicando il file da cui leggere, il numero di celle da leggere e l'indirizzo dove mettere le celle lette. Il DMA utilizzerà lo storage per leggere i dati del file.

3. Caratteristiche del sistema operativo

3.1 Meccanismo di scheduling

La parte più importante del sistema operativo è l'algoritmo di scheduling. Il processore viene condiviso da due o più processi, a seconda del numero di warrior; ogni warrior è associato ad uno ed un solo processo. Ogni processo contiene almeno un thread di controllo; ogni warrior può creare nuovi thread durante la propria vita. Ad ogni processo viene assegnato un quanto temporale, che viene utilizzato per eseguire uno o più thread del processo. Il meccanismo di scheduling fra processi è di tipo round-robin; i due (o più) processi contenuti nel sistema si alternano strettamente utilizzando il loro quanto temporale. Il meccanismo di scheduling per i thread di un processo è di tipo preemptive e basato su priorità. Esistono 5 livelli di priorità; ogni livello di priorità contiene una coda round-robin di thread.

Ogni thread può essere in tre differenti stati: *runnable*, *not runnable* e *dead*. Se un thread è in stato runnable, può eseguire senza problemi. Se un thread è in stato not-runnable, è in attesa di un evento (completamento di un'operazione di lettura o scrittura dallo storage). Un thread è in stato dead quando esegue un'istruzione non valida o una system call STOP. In questo caso, viene rimosso dal sistema.

Ogni processo può essere in tre differenti stati: *runnable*, *not runnable* e *dead.* Un processo è in stato runnable se almeno uno dei suoi thread è in stato runnable. Un processo è in stato not runnable se tutti i suoi thread non sono in stato runnable. Un processo è in stato dead se tutti i suoi thread sono in stato dead. Quando tutti i processi tranne uno sono in stato dead, il processo rimanente è dichiarato vincitore.

Il meccanismo di scheduling è il seguente. Il sistema operativo seleziona un processo, seguendo un meccanismo round-robin, tra i processi in stato runnable e gli assegna un quanto di tempo. Il processo continuerà ad eseguire fino a quando (i) il suo quanto temporale scade, o (ii) passa in stato not runnable. All'interno di un processo, il sistema seleziona la coda di priorità più alta fra quelle contenenti thread in stato runnable. All'interno di questa coda, seleziona in modo round-robin un thread ed inizia ad eseguirlo. L'esecuzione continuerà finchè: (i) il quanto temporale del processo scade, nel qual caso si seleziona un altro processo; (ii) il thread esegue una system call YIELD, ed

esiste un altro thread in stato runnable con la stessa priorità; (iii) il thread passa in stato not runnable, ovvero esegue una system call di lettura o scrittura nello storage; (iv) un altro thread dello stesso processo passa dallo stato not runnable allo stato runnable ed ha priorità più alta.

3.1 System Call

Il sistema operativo mette a disposizione dei warrior un certo numero di system call. L'invocazione di una system call causa la preemption del thread e l'intervento del sistema operativo. Tutte le system call ritornano immediatamente, a parte READ e WRITE che sono bloccanti.

- **STARTTHREAD address, priority**
  Aggiunge un thread all'elenco del Warrior, il cui indirizzo di partenza è dato da address e la cui priorità è dato da priority. Per far partire un thread con codice uguale a se stesso (comportamento simile ad una fork, sebbene stiamo parlando di thread e non di processi), un warrior può copiare il proprio codice in qualche altra zona di memoria a partire da un indirizzo x, e poi far partire il thread con starting address x. Per far partire un thread con codice diverso, un warrior può caricare il codice dallo storage, per poi seguire il procedimento precedente. La chiamata non è bloccante.
- **YIELD**
  Rilascia il processore ad un altro thread del processo, se questo esiste. Altrimenti continua. Il thread resta nello stato runnable; semplicemente
- **STOP**
  Questa chiamata interrompe un thread in esecuzione. Non ritorna nulla. Ovviamente, una chiamata del genere non dovrebbe mai essere chiamata volontariamente da un thread, visto che la morte di tutti i thread decreta la vittoria dell'avversario. Quindi questa system call può essere utilizzata per cercare di far terminare un thread avversario, coprendo una delle sue istruzioni con questa chiamata.
- **fd = OPEN fileIndex, mode**
  Apre un file contenuto nello storage identificato da fileIndex (un numero intero; non utilizziamo nomi complessi per semplicità). La modalità di apertura prevede due modalità principali: READ e WRITE (di significato intuitivo); Ritorna un file descriptor maggiore di zero se il file esiste e può essere aperto; ritorna –1 nel caso il file non esista; ritorna –2 nel caso si sia superato il numero massimo di file aperti per warrior.
- **READ fd, address, size**
  Legge size celle nel file identificato da file descriptor a partire dalla posizione corrente, collocandole all'indirizzo di memoria address (relativo alla chiamata di sistema, e quindi al program counter). Il numero massimo di celle che possono essere lette consecutivamente è determinato dalla variabile MaxReadWrite (dettagli in seguito). Ritorna il numero di celle lette, o un valore negativo nel caso di problemi.
- **WRITE fd, address, size**
  Scrive size celle nel file identificato da file descriptor a partire dalla posizione corrente nel file, prelevandole dall'indirizzo di memoria address (relativo alla chiamata di sistema, e quindi al program counter). Il numero massimo di celle che possono essere lette consecutivamente è determinato dalla variabile MaxReadWrite (dettagli in seguito). Ritorna il numero di celle scritte, o –1 nel caso vi siano problemi.
- **CLOSE fd** Chiude il file identificato dal file descriptor fd. Ritorna –1 nel caso il file descriptor non sia utilizzato.
- **SYSCONF paramId** Legge una delle variabili runtime elencate in seguito, che possono essere utilizzate da un warrior per variare le proprie strategie.

3.3 Variabili Run-Time

Il sistema operativo dispone di un certo numero di variabili implementative, che possono essere lette tramite la chiamata di system SYSCONF.

- **CoreSize**: Dimensione del core in numero di celle.
- **ContextSwitchBeforeTie**: In ogni ciclo, un certo numero di istruzioni (Quantum) per ognuno dei warriors presenti viene eseguito. Questa variabile dichiara il numero di cicli che devono essere eseguiti prima che il sistema operativo dichiari la parità tra processi.
- **Quantum**: numero di istruzioni eseguite prima del prossimo context switch. Rappresenta il numero di istruzioni eseguite dal sistema operativo prima di interrompersi per eseguire un'altro warrior.
- **MaximumNumberOfFile**: ogni warrior può aprire un certo numero di file, limitato superiormente da questa variabile runtime. Thread multipli appartenenti allo stesso warrior condividono la tabella dei file.
- **MaxReadWrite** Numero massimo di celle che possono essere lette o scritte da una chiamata MaxReadWrite.
- **MaximumNumberOfTasks**: ogni warrior può lanciare un certo numero di task (thread) addizionali. Questa variabile rappresenta il numero massimo di task che possono essere presenti nel sistema.
- **MinimumSeparation**: inizialmente, ogni warrior viene caricato in memoria in una posizione casuale. Questa variabile rappresenta la minima distanza che deve esistere tra i diversi warrior presenti.
- **WarriorsNumber**: numero di Warriors presenti nel sistema.

Valori che possono essere adottati nelle vostre prove:

- CoreSize: 8192 celle
- ContextSwitchBeforeTie: 20000
- Quantum: 10
- MaximumNumberOfFiles: 16
- MaximumNumberOfTasks: 8
- MinimumSeparation: 2048 celle
- WarriorsNumber: 2

3.4 Meccanismo di bootstrap

Alla partenza, il sistema operativo legge un mini-programma (detto di bootstrap) dallo storage per ogni warrior presente nel sistema e lo colloca nella memoria; il programma di bootstrap del warrior i-esimo, identificato dal process id i, verrà caricato a partire dal file i. Ovviamente, il file *i* deve appartenere al warrior i-esimo. La lunghezza del mini programma è limitata a 16 celle; se un programmatore vuole realizzare programmi più complessi, dovrà fare in modo che il programma di bootstrap carichi altri file e li collochi in memoria.

4. Implementazione

Il timer, il processore (o i processori, nel caso di un'architettura multi-processore) e il DMA devono essere realizzati come thread o processi distinti nel linguaggio di programmazione scelto. Questo significa che essi devono sincronizzarsi e competere per l'accesso alla struttura dati condivisa, ovvero la memoria. All'inizio, il vostro programma dovrà contenere un metodo main il quale lancia

i vari thread rappresentanti elementi del sistema, inizializza uno storage, copia i programmi di bootstrap in memoria e inizia il meccanismo di scheduling.

Per quanto riguarda l'output, i programmi originali di Corewars visualizzano l'arena di gioco in modo grafico e mostrano i vari warrior che si rincorrono nella memoria. Niente di tutto ciò. E' richiesto invece l'inserimento di stampe di debug ogni qualvolta che viene invocato un interrupt handler (system call, interrupt del timer, istruzioni non valide, interrupt del DMA). Queste stampe di debug dovranno contenere qualche informazione sull'interrupt e sull'azione eseguita; ad esempio, il numero di system call e gli argomenti nel caso di una system call, il processo schedulato nel caso di un interrupt del timer o di una system call YIELD, etc.

5. Organizzazione e consegna

Il progetto verrà affrontato da gruppi di 3-4 persone. Non sono ammessi gruppi di dimensione diversa. La consegna avverrà tramite posta elettronica (montreso@phd.cs.unibo.it), rispettando le regole seguenti:

- il subject del messaggio sarà costituito dalla stringa LSO-PROGETTO
- il body conterrà un attachment, in formato compresso (ZIP o GZ), contenente
    - i file che implementano il programma
    - uno o più file contenenti una relazione sul progetto, la quale discuterà delle scelte progettuali effettuate
    - uno storage (codificato nel modo preferito; ad esempio, come file di programma contenente un'inizializzatore per l'array; oppure, come file normale letto tramite un qualche meccanismo)
    - un file readme, contenente poche righe di spiegazione su come lanciare il vostro programma

La deadline per la consegna sarà indicativamente 7 giorni prima della prova pratica. La data esatta sarà indicata sul newsgroup; l'intervallo fra la consegna e l'orale potrà essere occasionalmente inferiore, qualora impegni da parte mia mi impediscano di svolgere regolare ricevimento nelle settimane precedenti.

A. Varianti

La specifica del problema presentata in questo documento racchiude gli elementi basilari che dovrebbero essere presenti nel progetto. Non deve essere intesa come una specifica fissa, da soddisfare in ogni caso. Modifiche e miglioramenti sono benvenuti, a patto che siano motivati e discussi nella relazione finale.

Alcuni esempi di elementi aggiuntivi che potrebbero essere considerati:

- estendere l'architettura in modo da considerare la presenza nel core di un numero variabile di warrior (non limitato a due)
- estendere l'architettura in modo tale che siano presenti più di un processore, in cui ogni processore viene dedicato ad un singolo warrior (ovvero, a tutti i task del warrior). E' interessante il caso in cui il numero di warrior e il numero di processori non coincide. In questo caso, ogni processore corrisponderà ad un thread nel linguaggio di programmazione scelto (Java, C, C++) e competerà con gli altri processori e il sistema DMA per il controllo della memoria.
- estendere l'architettura in modo da considerare l'esecuzione batch di coppie (insiemi, se si considera l'estensione precedente) di warrior; ogni coppia (insieme) viene caricato in memoria, dopo di che i warrior vengono fatti partire; quando la competizione termina (per morte di tutti i warrior tranne uno o per pareggio), si passa alla coppia (insieme) successivo.