UNIVERSITÀ
DEGLI STUDI
FIRENZE

Scuola di Ingegneria

Corso di Laurea in Ingegneria Informatica

Software Engineering for Embedded Systems
Project work

Real-time Scheduling Simulator

Edoardo Sarri

7173337

Data

Indice

Elenco delle figure

1 Introduzione

Il progetto vuole modellare e implementare un sistema in Java che permetta di generare tracce di un'esecuzione a partire dalla definizione di un taskset con o senza risorse da usare in mutua esclusione. Le eventuali risorse sono gestite da un protocollo di accesso alle risorse.

Ogni traccia è definita come una sequenza di coppie $< tempo, evento >$, dove un *evento* può essere: rilascio di un job di un task; acquisizione/rilascio di un semaforo da parte di un job di un task; completamento di un chunk; completamento di un job di un task.

1.1 Capacità

A partire da un taskset, il sistema ha le capacità di:

- Generare la traccia di esecuzione del taskset schedulato tramite un dato algoritmo di scheduling e ed un eventuale protocollo di accesso alle risorse.
- Rilevare eventuali deadline miss. Se al termine del prorpio periodo, un task non ha completato un tutti i chunk di cui è composto allora il sistema lo evidenzia e si arresta.
- Introdurre e rilevare un additional execution time in un chunk. Questo modella un tempo di computazione di un chunk maggiore di quello che ci aspettiamo, cioè maggiore del WCET.

2 Analisi

In questo capitolo analiziamo la struttura del progetto, partendo dai suoi componenti e definendo la loro relazione.

2.1 Componenti

2.1.1 Task

Un task è definito da: un un insieme di Chunk; una deadline; una priorità nominale e dinamica; un pattern di rilascio.

Non ci interessa definire un activation time perché vogliamo considerare il caso pessimo: l'activation time sarà uguale per tutti i task e coinciderà con l'istante inziale.

2.1.2 Chunk

Un chunk, cioè una computazione atomica del task. È definito da: una distribuzione del tempo di esecuzione previsto e un tempo di esecuzione effettivo; un'eventuale richiesta di risorse da usare in mutua esclusione (da acquisire prima dell'esecuzione e rilascaire subito dopo).

2.1.3 Taskset

È un insieme di task. È l'oggetto principale gestito dallo scheduler.

2.1.4 Risorse

Sono le risorse da utilizzare in mutua esclusione. Ogni risorsa è gestita da un semaforo binario, quindi può essere posseduta da un solo task alla volta.

2.1.5 CPU

È l'unità di elaborazione. Supponiamo essere unica.

2.1.6 Scheduler

È il componente che assegna un task al processore. Sono stati implementati Rate Monotonic (RM) e Earlist Deadline First (EDF).

2.1.7 Protocollo di accesso alle risorse

È il meccanismo che garantisce la mutua esclusione di una risorsa. È stato implementato Priority Ceiling Protocol (PCP).

2.2 Class diagram

Per capire meglio la struttura del progetto, analizziamo il diagramma delle classi.

In questo modello sono definiti solo i campi che definiscono i vari componenti; per chiarezza del sistema non sono stati infatti introdotti gli oggetti necessari all'implementazione.

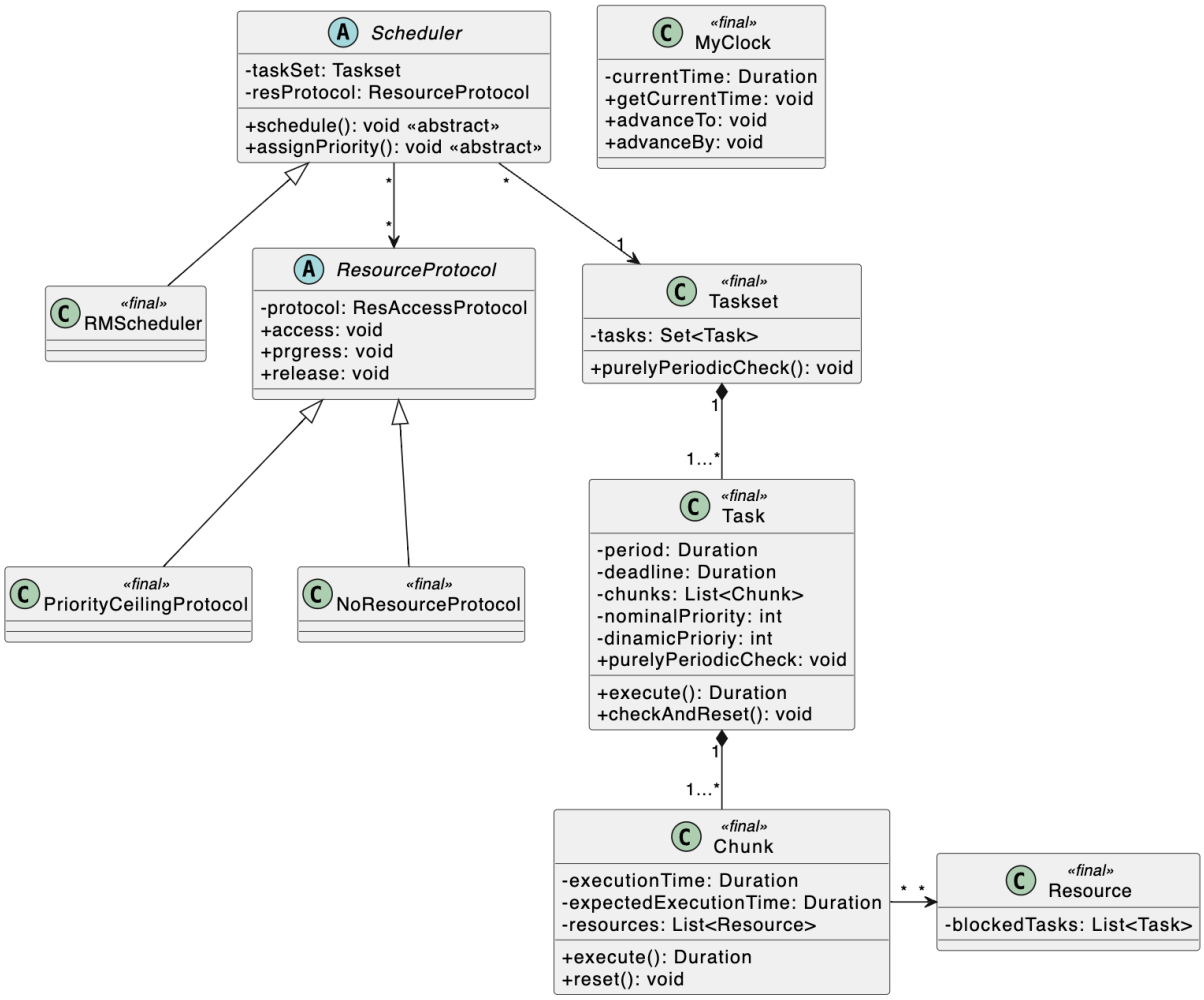

Figura 2.1: Class diagram.

3 Implementazione

L'implementazione deve rappresentare una simulazione deterministica. Per questo motivo non sono stati usati tread (oggetti Java `Thread`).

3.1 Scheduler

Ogni possibile implementazione di un algoritmo di scheduling è un'estensione della classe `Scheduler`. Questo permette di definire i comportamenti comuni a tutti gli scheduler e di astrarre future implementazioni.

Per l'implementazione, oltre a ciò che definisce uno scheduler (i.e., taskSet e l'eventuale protocollo di accesso alle risorse), è stato necessario mantenere una lista di task pronti `readyTaskse` una di task bloccati `blockedTask`. Inoltre è necessario mantenere un riferimento all'ultimo task che è andato in esecuzione.

Nella classe base sono presenti, oltre che ai classici getter e setter, anche i metodi fondamentali che ogni sotto classe deve implementare: `schedule` che definisce la politica di scheduling; `assignPriority` che definisce come assegnare la priorità ai task.

Quando uno scheduler viene creato oltre a assegnare il taskset e il protocollo di accesso alle risorse, viene chiamato `assignPriority`, e vengono inizializzate le strutture relative al protocollo. La scelta di fare questo assegnamento qua e non nella classe dedicata al protocollo è dovuta alle dipendenze: per come è stato implementato l'oggetto principale (e anche l'ultimo che deve essere istanziato) è lo scheduler, e quindi è il protocollo si basa su di esso.

3.1.1 Rate Monotonic

Descriviamo brevemente l'idea di implementazione di RM. Per semplificare l'ordine con cui i vari componenti interagiscono è stato definito il sequence diagram in Figura 3.1.

Dopo aver chiamato il costruttore della superclasse, RM valuta se tutti i task che compongono il taskset sono puramente periodici, cioè se hanno stesso periodo e deadline relativa.

La simulazione di scheduling si basa su due strutture principali:

- `taskReady`
  Ospita i task che si contendono l'accesso alla CPU. È stata implementa tramite un

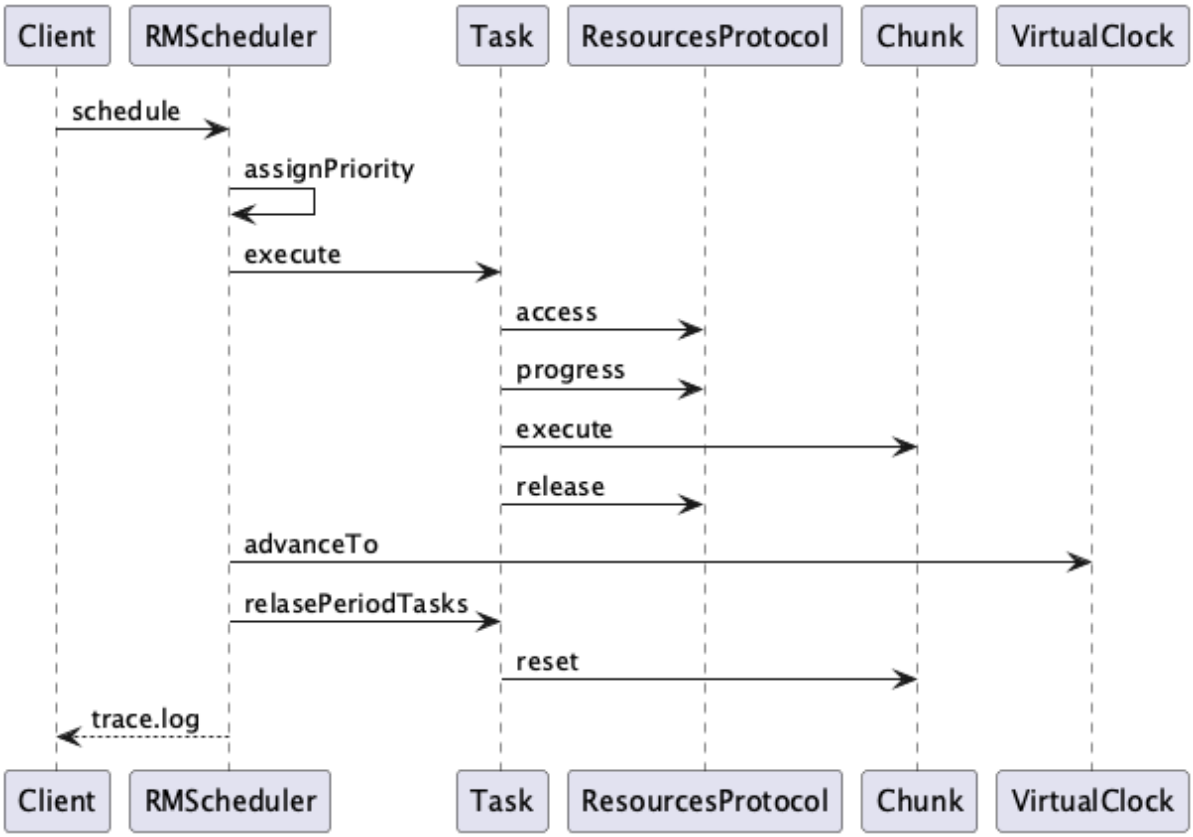

Figura 3.1: Sequence Siagram RM

`Treeset` e al suo interno si mantiene un'ordinamento inverso rispetto alla durata del periodo dei vari task: task con periodo inferiore ha una priorità maggiore.

- `events`
  Gestisce gli eventi rilevanti per lo scheduler, cioè i momenti in cui deve valutare se i task che hanno terminato il periodo hanno superato la deadline oppure se devono essere rilasciati nuovamente. Nell'intervallo tra un perido e il successivo infatti lo scheduler non fa altro che mandare in esecuzione uno dopo l'altro il task a priorità maggiore. Gli eventi sono l'unione ordinata dei multipli di ciascun periodo fino al minimo comune multiplo dei periodi oppure fino a 10 volte il periodo maggiore (per semplicità del caso sia molto oneroso generare questa lista).

Il metodo `schedule` poi delega la gestione dei chunk di ogni task alla classe `Task`, la quale a sua volta rimanda alla calsse `Chunk` la loro esecuzione (i.e. il logging).

3.2 Resource Access Protocol

Ogni implementazione di un protocollo di accesso alle risorse deve estendere la classe `ResourceProtocol`.

l'idea iniziale era di implementare il concetto astratto di protocollo di accesso tramite un'interfaccia, ma vista la necessità di mantenere un riferimento allora scheduler nel protocollo si è introddo questo campo nella classe base.

I metodi definiti da questa classe astratta sono le operazioni che devono essere svolte da un procotollo di questo tipo: deve gestire la fase di accesso, progresso e rilascio. Definisce anche il metodo `initStructures` che ha il compito di inizializzare le strutture dati usate dal protocollo.

Oltre a questi metodi è dichiarato un metodo `initStructures` che inizializza le strutture necessarie al protocollo.

3.2.1 Priority Ceiling Protocol

Tralasciando quello che fanno i metodi di accesso, progresso e rilascio, che riflettono quanto ci dice la teoria, in questo classe le strutture usate sono prevalentemente due:

- `ceiling`
  È una mappa che associata ad ogni risorsa il suo ceiling, cioè la massima priorità nominale dei task che usano quella risorsa.

- `busyResources`
  È una lista delle risorse che sono occupate da un qualche task.

3.3 Fault injection

Per implementare l'injection di un additional execution time in un `chunk` è stato previsto un altro costruttore che prendesse, oltre ai parametri previsti, anche un `overheadExectutionTime`.

Il motivo per non introdurre una classe che eseguisse l'injection è stato che un chunk nella realtà nasce con il suo execution time e non ci sono altre entità che aumentano quello campionato; il costrutture vuole modellare questa idea.

3.4 Utilità

Vediamo la scelta su alcuni componenti di utilità.

3.4.1 Loggin

Per il logging è stato implementato un semplice logging su un file e viene rappresnetato come una sequenza di coppie $< evento, tempo >$.

Il file di destinazione delle tracce loggate è `trace.log`.

3.4.2 Clock

Il tempo all'interno del sistema è gestito globalmente: ad ogni esecuzione del `main` il tempo viene resettato. Questa scelta è dovuta al fatto che praticamente tutti gli ogegtti devono accedere al clock del sistema; in questo modo si evita di passarlo ogni volta nei vari metodi chiamati a cascata. L'implementazione è stata fatta tramite il pattern Singleton.

A causa della staticità si consiglia di usare eseguire una sola simulazione per esecuzione. C'è la possibilità di resettarlo manualmente (tramite `MyClock.reset()`); questa possibilità è stata introdotto per i test.

Il clock del sistema è rappresnetato dalla classe `MyClock`. Questa non fa altro che mantenere il tempo assoluto ed esporre due metodi che permettono di avanzare di un dato intervallo temporale e avanzare fino a un determinato tempo.

Il tempo è gestito tramite oggetti di tipo `Duration`, classe di `java.time` che implementa oggetti immutabili e che permetto una facile gestione del tempo.

In particolare il tempo deve essere consdierato dall'utente (i.e., passato in input e restituito poi in output) in millesecondi, ma il sistema lo gestisce tramite i nanosecondi. Questo permette di lavorare con millisecondi frazionari quando si campiona dalle distribuzioni di Sirio.

La stampa all'interno delf ile di log `trace.log` nel formato corretto è implementa nel metodo `Utils.printCurrentTime`.

3.4.3 Sampling dei tempi

Quando si deve definire i tempi che definiscono i vari componenti del sistema, cioè come il periodo, la dealine, l'execution time di un chunk, si usa un campionamento da un data distribuzione.

Le distribuzioni sono implementate dalla libreria `Sirio`; oltre a quelle definite dalla libreria è stata implementata la classe `ConstantSampler`, che permette di gestire tempi costanti, mantenendo l'astrazione della libreria Sirio.

I sampler di Sirio restiuiscono oggetti di tipo `BigDecimal`. Come detto nel paragrafo sopra il sistema gestisce il tempo come oggetti di tipo `Duration`. Per implementare questo meccanismo è stata implementata la classe `SampleDuration`, che preso un `Sampler` restituisce il rispettivo tempo in nanosecondi.

3.4.4 Eccezioni

Durante l'esecuzione si possono verificare dei problemi, più o meno previsti. Questi sono gestiti tramite eccezioni; questo permette in futuro di cambiare o aggiungere un comportamente del sistema quando si verificano determinate situazioni.

Le eccezioni implementate è utili sono:

- `DeadlineMissedExeption`
  Viene sollevata quando un task non rispetta la deadline. Il sistema non la gestisce, ma la propaga fino al main: in questo modo se e quando si verifica questo problema, viene stamapta in `trace.log` e la simulazione si arresta.

- `AccessResourceProtocolExeption`
  Viene sollevata quando un task viene bloccato dal metodo `access` del protocollo di accesso.

4 Dubbi

4.1 Domande

- Nei fallimenti osservati che vuol dire valutare la violazione del tempo di computazione di un chunk (troppo basso o troppo alto)? Se non viola la deadline allora esegue per il suo execution time altrimenti di più.
- Manca EDF e la possibilità di iniettare fault.
- Generazione lista eventi con multipli. vedi test generateMultiplesUpToLCM4.

Bibliografia

[1] Laura Carnevali. *Appunti slides Software Engineering for Embedded System*.

[2] *chatGTP*.

[3] *documentazione Java, oracle*. https://docs.oracle.com/javase/8/docs/api/overview-summary.html.