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

Giugno 2025

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; preemption che un task può subire.

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 (i.e., Rate Monotonic e Earlist Deadline First) e ed un eventuale protocollo di accesso alle risorse (i.e., Priority Ceiling Protocol). Considerando la gestione dinamica delle priorità da parte di EDF, è previsto il suo utilizzo solo senza risorse condivide.
- Generare un dataset di tracce relative a più simulazioni su un taskset.
- Specificare il tempo desiderato della simulazione.
- Rilevare eventuali deadline miss. Se un task non ha finito di eseguire entro la propria deadline allora la simulazione si arresta; non continua perché ci interessa valutare il primo fallimento, visto che i successivi potrebbero essere in cascata del primo.
- Campionare da una distribuzione e rilevare un additional execution time in un chunk. Questo modella un tempo di computazione di un chunk maggiore (o minore) di quello che ci aspettiamo, cioè maggiore del WCET.
- Introdurre in modo stocastico e rilevare un fault a livello del protocollo di accesso alle risorse (i.e., PCP) tale per cui la priorità dinamica assegnata al task che entra in critical section è quella corretta più un valore campionato da una distribuzione uniforme.
- Introdurre in modo stocastico e rilevare un fault a livello di chunk tale per cui con una certa probabilità il chunk non acquisisce, e quindi non rilascia, il semaforo che gestisce la risorsa associata alla critical section in cui entra.

- Definito uno scheduler con il relativo taskset, eseguire il controllo di feasibility adeguato per la schedulabilità del taskset. Per RM è stato impelementato l'hyperbolic bound; per EDF il controllo necessario e sufficiente sul fattore di utilizzo.

1.2 Utilizzo

Nei prossimi punti vediamo degli accorgimenti che sono utili per far utilizzare il sistema nel modo corretto.

- Gli elementi del sistema, come chunk, task, taskset, scheduler e protocollo di accesso alle risorse, devono essere definiti all'interno del main. Una volta definito tutto si deve chiamare il metodo `schedule` sullo scheduler. Nel caso si voglia generare una sorta di dataset di tracce, è prevista la possibilità di chiamare più volte il medtodo `schedule` all'interno dello stesso main. Per generare un dataset di tracce è invece necessario chiamare il metodo `scheduleDataset` passando come parametro il numero di tracce.

- Un chunk campiona il suo tempo di esecuzione da una distribuzione. Tale distribuzione, passata come parametro del costruttore, è un oggetto di tipo `Sampler` definito nella libreria Sirio. Oltre alle distribuzioni introdotte da Sirio è presente anche l'implementazione `ConstantSampler`, che definisce un campionamento costante.

- I tempi devono essere passati al sistema e letti da esso in milli secondi. Il sistema li gestisce in nanosecondi per avere un'alta precisione.

- Quando si vuole introdurre il fault relativo alla priorità dinamica settata da PCP si deve specificare il valore minimo e massimo entro cui campionare. Questi valori sono intesi come esclusi.

- Quando si vuole introdurre il fault relativo all'acquisizione del semafore associato alle risorse, la soglia specificata deve essere in un intervallo 0.0 e 1.0.

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 Figura 2.1.

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.

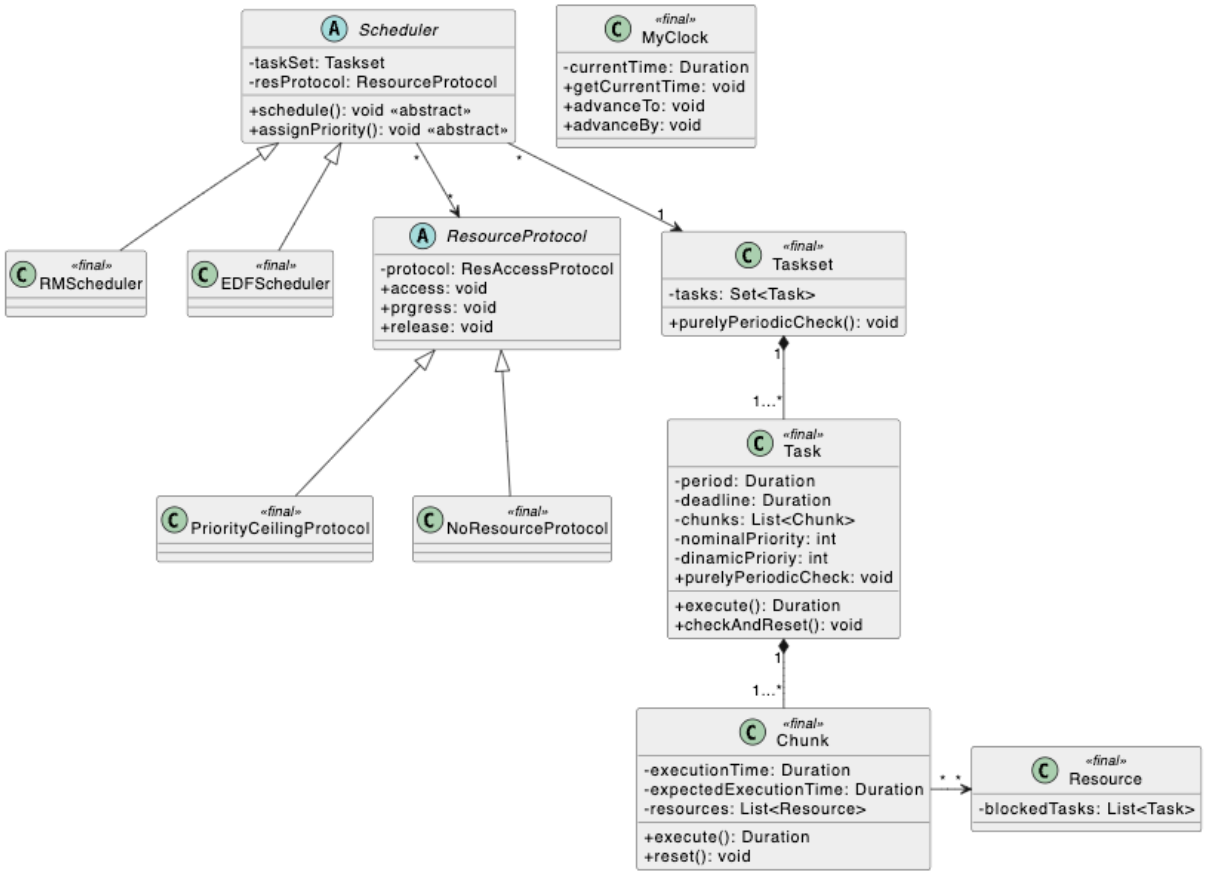


Figura 2.1: Class diagram

3 Implementazione

L'implementazione deve rappresentare una simulazione deterministica. Per questo motivo non sono stati usati tread (oggetti Java `Thread`), la gestione del tempo avviene in modo discreto.

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.

L'implementazione della logica di scheduling è la stessa sia per RM che per EDF: il metodo `schedule` con i metodi healper necessari sono stati, per questo motivo, inseriti all'interno della classe base. Ogni implementazione concreta deve sono definire alcuni aspetti usati poi nella logica. Per modellare questo principio è stato usato il Template Mehod pattern: il metodo `schedule` è dichiaro pubblico e `final` in modo che non possa essere modificato dalle implementazioni, mentre gli hooks chiamati al suo interno sono dichiarati `abstract` e `protected`.

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

Per gestire i tempi rilevanti, cioè quelli in cui lo scheduler deve prendere il controllo del sistema per rivere la propria politica di scheduling, è delegata alla struttura `readyTasks`. 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 rilevanti 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).

Per capire i passaggi che esegue la simulazione consideriamo e analiziamo il sequence diagram di Figura 3.1:

- `assignPriority`
  Assegna le priorità secondo l'implementazione in questione. È dichiarato astratto in `Scheduler` e deve essere implementato dalle classi concrete.

- `Myclock.reset`
  Resettando il clock di sistema prima di iniziare la simulazione, si permette di eseguire più volte il metodo `schedule` all'interno dello stesso main. Questo è necessario perchè, come viene descritto nella Sezione 3.4.2, il clock del sistema è implementato in maniera statica.

- `initStructure`
  Inizializza le strutture dati usate dalla simulazione: la lista di task pronti, e la lista di eventi importanti.

- `checkFeasibility`
  Controlla se è possibile, o meglio se non è possibile (e.g., per RM), schedulare il taskset secondo i test di schedulabilità implementati: per RM utilizza l'hyperbolic bound; per EDF valuta se il fattore di utilizzo è maggiore o minore dell'unità.

- `Task.execute`
  È il metodo che manda in esecuzione sistema per un tempo limitato. Questo tempo va dal tempo attuale fino al prossimo evento significativo.

- `access-progress-release`
  Sono i metodi che definsicono il protocollo di accesso alle risorse. I dettalgi implementativi sono specificati nel Paragrafo 3.2.

- `Chunk.execute`
  Definsice l'esecuzione del singolo chunk. In particolare si occupa della fase di logging.

- `relasePeriodTasks`
  Si occupa di rilasciare i task nel momento in cui scocca il suo periodo. Inoltre controlla che quei task abbiano finito di eseguire ed eventualmente sollava una `DeadlineMissExecption`.

- `reset`
  Durante l'esecuzione lo stato dei task e dei chunk cambia per riflettere informazioni relative all'esecuzione. Con questo metodo si vuole riportare i task e chunk al loro stato iniziale.

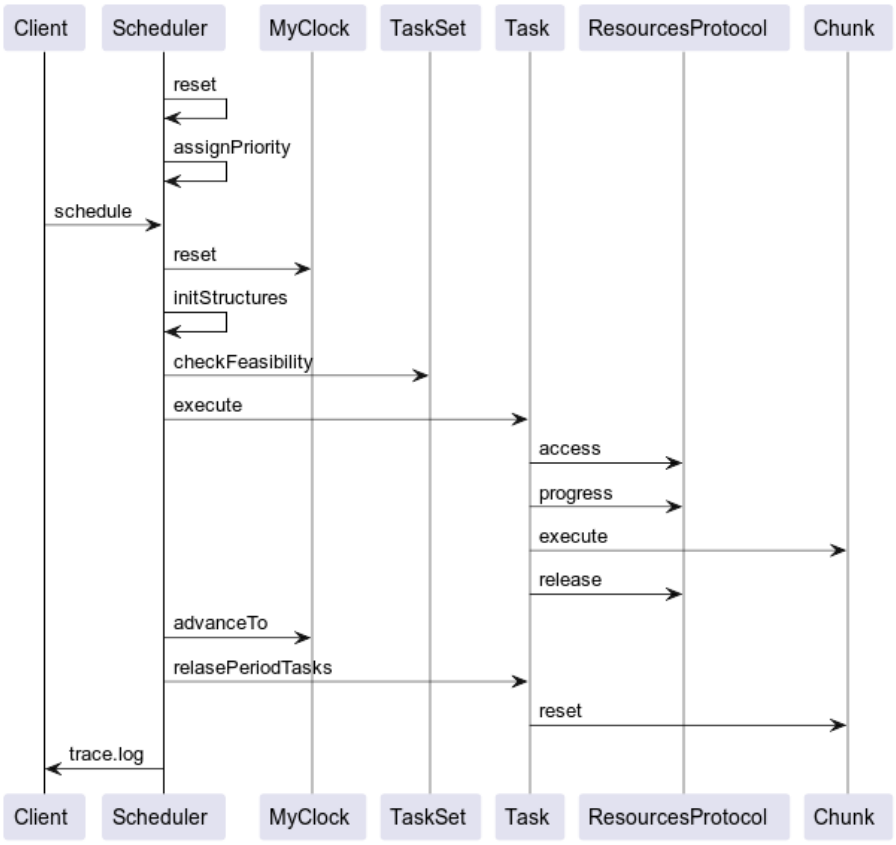


Figura 3.1: Sequence Diagram scheduling

3.1.1 Rate Monotonic

Vediamo come sono implementati i metodi astratti della classe `Scheduler` in RM, cioè quelli che definsicono il suo comportamento rispetto agli altri scheduler.

- `checkFeasibility`
  Viene controllato l'hyperbolic bound. È stato deciso di non cnsiderare il test di Lui & Layland viesto che questo non è tight: se un taskset non risulta schedulabile seconda questo test, l'hyperbolic bound potrebbe definire che è schedulabile.

- `assignPriority`
  Assegna la priorità in modo inverso rispetto alla durata del periodo.

- `addReadyTask`
  Siccome la lsita dei task pronti per l'esecuzione è oridnata in modo dinamica (visto che è implementata come un $TreeSet < Task >$ con un comparatore) secondo la priorità dinamica, aggiunge semplicemente il task a tale lista.

3.1.2 Earlist Deadline First

Vediamo come sono implementati i metodi astratti della classe `Scheduler` in EDF, cioè quelli che definsicono il suo comportamento rispetto agli altri scheduler.

- `checkFeasibility`
  Viene controllato che il fattore di utilizzo del taskset sia minore o uguale a 1.
- `assignPriority`
  Assegna la priorità in modo inverso rispetto alla durata della dealine.
- `addReadyTask`
  Aggiunge il task alla lista dei task printi e poi la riordina secondo la prossima deadline. Il metodo che stabilisce la prossima deadline è `Task.nextDeadline`.

3.1.3 Test

Per quanto riguarda i test su RM sono stati eseguiti con e senza risorse condivise. In entrambi i casi oltre a garantire che non si arrivi a un deadline miss quando il taskset è schedulabile e ci si arrivi invece quando il taskset non lo è, è stato osservato il file di log generato per capire se esso era compliance alla politica di scheduling con e senza PCP.

Per eseguire i test su taskset non schedulabili è stato preso come SUT (System Under Test) con fattore di utilizzo maggiore di 1. In questo modo siamo sicuri che il taskset non sia schedulabile.

Per EDF valgono gli stessi concetti di cui sopra. Ovviamente siccome non è stato implementato un protocollo di accesso alle risorse da usare con EDF, sono stati testati sono taskset i cui chunk non utilizzano risorse condivise.

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

3.3.1 Additional execution time

Per implementare il fault 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.

3.3.2 Priority Ceiling Protocol

Sono state aggiunte due classi che introducono due tipi di fault: il primo setta una priorità dinamica errata quando un task entra nella critical section; il secondo non fa acquisire una risorsa quando al chunk che la utilizza.

Le classi che implementano questi comportamenti sono:

- `PriorityCeilingProtocolFaultSetPriority`
  L'idea è campionare un valore delta, da aggiungere poi alla priorità dinamica corretta, da una distribuzione uniforme. Tale distribuzione è un'istanza di `UniformSampler` della libreria Sirio. I due parametri necessari alla distribuzione sono presi tramite costruttore.

- `PriorityCeilingProtocolFaultAquireResource`
  Si campiona da una distribuzione uniforme (i.e., `Math.random()`) un valore compreso tra 0 e 1. Se il valore passato come soglia è minore di questo valore campionato allora si salta il blocco di codice responsabile dall'acquisizione del semaforo.

Per non rilasciarlo al termine dell'esecuzione del chunk, questo viene inserito in una lista.

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 staticamente, e quindi a livello globale. 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.

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 Dataset

Per concludere analiziamo come il sistema si comporta sulla generazione di dataset. Si procederà in modo incrementale, per capire come affrontare lo scheduling di taskset più semplici e poi quelli più complessi.

Questa parte può essere anche utile per capire come struttrare il main ed utilizzare il simulatore.

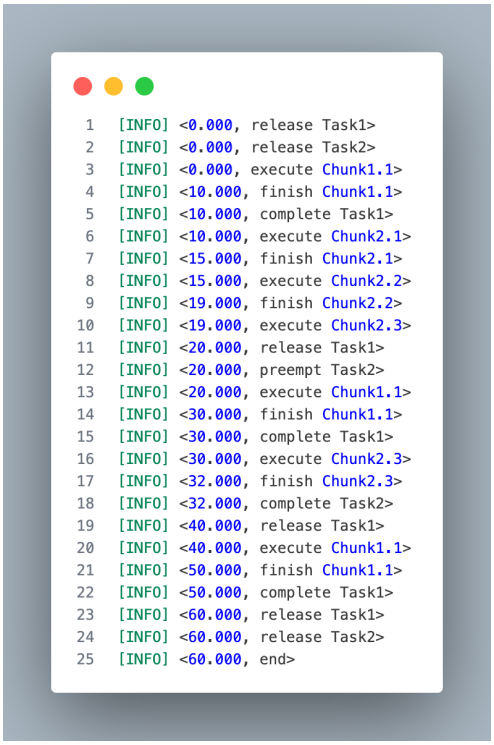
4.1 Baseline

Vediamo come funziona una simulazione tramite RM con un taskset molto semplice. Il taskset in questione, come si può notare dal codice in Figura 4.1a, è molto semplice in modo da poter osservare le differenze tra le due simulazioni. Si tratta di un task con due task puramenti periodici: il primo ha periodo e deadline relativo pari a 20ms e un chunk con un exectuion time di 10ms; il secondo ha periodo e deadline di 60ms e tre chunk con execution time rispettivamente di 5ms, 4ms e 3ms.

La traccia in Figura 4.1b è l'output della simulazione.

```
Task task1 = new Task(
    20,
    20,
    List.of(
        new Chunk(1, new ConstantSampler(new BigDecimal(10)))));
Task task2 = new Task(
    60,
    60,
    List.of(
        new Chunk(1, new ConstantSampler(new BigDecimal(5))),
        new Chunk(2, new ConstantSampler(new BigDecimal(4))),
        new Chunk(3, new ConstantSampler(new BigDecimal(3)))));
TaskSet taskSet = new TaskSet(Set.of(task1, task2));
Scheduler rm = new RMScheduler(taskSet, 60);
rm.schedule();
```

(a) Taskset baseline



```
[INFO] <0.000, release Task1>
[INFO] <0.000, release Task2>
[INFO] <0.000, execute Chunk1.1>
[INFO] <10.000, finish Chunk1.1>
[INFO] <10.000, complete Task1>
[INFO] <10.000, execute Chunk2.1>
[INFO] <15.000, finish Chunk2.1>
[INFO] <15.000, execute Chunk2.2>
[INFO] <19.000, finish Chunk2.2>
[INFO] <19.000, execute Chunk2.3>
[INFO] <20.000, release Task1>
[INFO] <20.000, preempt Task2>
[INFO] <20.000, execute Chunk1.1>
[INFO] <30.000, finish Chunk1.1>
[INFO] <30.000, complete Task1>
[INFO] <30.000, execute Chunk2.3>
[INFO] <32.000, finish Chunk2.3>
[INFO] <32.000, complete Task2>
[INFO] <40.000, release Task1>
[INFO] <40.000, execute Chunk1.1>
[INFO] <50.000, finish Chunk1.1>
[INFO] <50.000, complete Task1>
[INFO] <60.000, release Task1>
[INFO] <60.000, release Task2>
[INFO] <60.000, end>
```

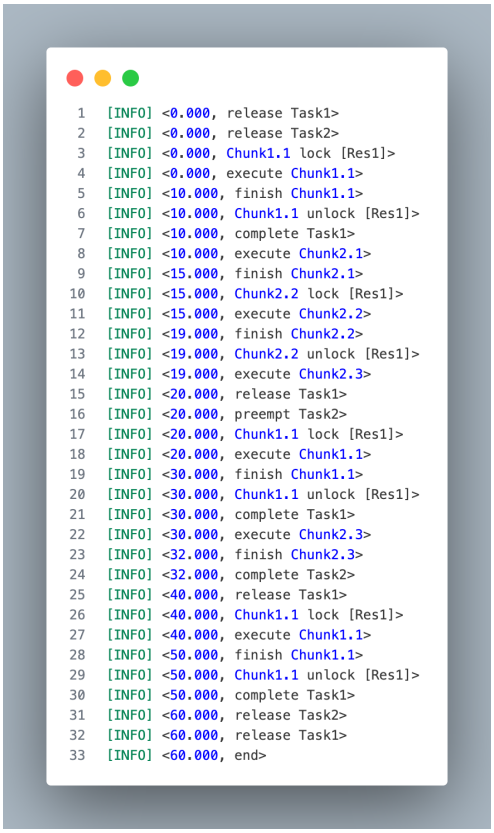
(b) Traccia del task baseline

Figura 4.1: Baseline.

Di seguito osserviamo invece lo stesso taskset schedulato sempre con RM, ma in cui due chunk (uno per task) condividono una risorsa. In Figura 4.2a possiamo osserve come configurarlo, mentre in Figura 4.2b possiamo osservare la traccia della simulazione.

```
Resource resource = new Resource();
Task task1 = new Task(
    20,
    20,
    List.of(
        new Chunk(1, new ConstantSampler(new BigDecimal(10)), List.of(resource))));
Task task2 = new Task(
    60,
    60,
    List.of(
        new Chunk(1, new ConstantSampler(new BigDecimal(5))),
        new Chunk(2, new ConstantSampler(new BigDecimal(4)), List.of(resource)),
        new Chunk(3, new ConstantSampler(new BigDecimal(3)))));
TaskSet taskSet = new TaskSet(Set.of(task1, task2));
PriorityCeilingProtocol pcp = new PriorityCeilingProtocol();
Scheduler rm = new RMScheduler(taskSet, pcp, 60);
rm.schedule();
```

(a) Taskset baseline con risorsa condivisa



```
[INFO] <0.000, release Task1>
[INFO] <0.000, release Task2>
[INFO] <0.000, Chunk1.1 lock [Res1]>
[INFO] <0.000, execute Chunk1.1>
[INFO] <10.000, finish Chunk1.1>
[INFO] <10.000, Chunk1.1 unlock [Res1]>
[INFO] <10.000, complete Task1>
[INFO] <10.000, execute Chunk2.1>
[INFO] <15.000, finish Chunk2.1>
[INFO] <15.000, Chunk2.2 lock [Res1]>
[INFO] <15.000, execute Chunk2.2>
[INFO] <19.000, finish Chunk2.2>
[INFO] <19.000, Chunk2.2 unlock [Res1]>
[INFO] <19.000, execute Chunk2.3>
[INFO] <20.000, release Task1>
[INFO] <20.000, preempt Task2>
[INFO] <20.000, Chunk1.1 lock [Res1]>
[INFO] <20.000, execute Chunk1.1>
[INFO] <30.000, finish Chunk1.1>
[INFO] <30.000, Chunk1.1 unlock [Res1]>
[INFO] <30.000, complete Task1>
[INFO] <30.000, execute Chunk2.3>
[INFO] <32.000, finish Chunk2.3>
[INFO] <32.000, complete Task2>
[INFO] <40.000, release Task1>
[INFO] <40.000, Chunk1.1 lock [Res1]>
[INFO] <40.000, execute Chunk1.1>
[INFO] <50.000, finish Chunk1.1>
[INFO] <50.000, Chunk1.1 unlock [Res1]>
[INFO] <50.000, complete Task1>
[INFO] <60.000, release Task2>
[INFO] <60.000, release Task1>
[INFO] <60.000, end>
```

(b) Traccia del task baseline con risorsa condivisa

Figura 4.2: Baseline con risorsa condivisa

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.