Tesina Progettazione di Sistema Operativi

Federico Canali

23/11/2022

Indice

Introduzione

All'interno di questo elaborato, si analizzano e implementano in linguaggio C quattro problemi di sincronizzazione citati nel libro *"The Little Book of Semaphores"*.
In particolare, i problemi che veranno trattati sono i seguenti:

- *The Santa Claus problem* (pagine 137 - 142);
- *The roller coaster problem* (pagine 153 - 158);
- *The unisex bathroom problem* (pagine 170 - 177);
- *The sushi bar problem* (pagine 183 - 194).

Capitolo 1

The Santa Claus problem

1.1 Testo del problema

Babbo Natale dorme nel suo negozio al Polo Nord e può essere svegliato solo dal ritorno di tutte e nove le renne dalle loro vacanze nel Pacifico meridionale, o da alcuni elfi che hanno difficoltà a fabbricare giocattoli; per permettere a Babbo Natale di dormire un po' , gli elfi possono svegliarlo solo quando tre di loro hanno problemi. Quando tre elfi stanno risolvendo i loro problemi, tutti gli altri elfi che desiderano visitare Babbo Natale devono aspettare che quegli elfi tornino. Se Babbo Natale si sveglia e trova tre elfi in attesa davanti alla porta del suo negozio, insieme all'ultima renna tornata dai tropici, Babbo Natale decide che gli elfi possono aspettare fino a dopo Natale, perché è più importante preparare la sua slitta. (Si presume che le renne non vogliano lasciare i tropici, e quindi vi rimangono fino all'ultimo momento possibile.) L'ultima renna ad arrivare deve prendere Babbo Natale mentre gli altri aspettano in una capanna riscaldata prima di essere imbrigliati alla slitta.

1.2 Soluzione

I threads coinvolti sono:

- 1 per Santa Claus;
- 9 per le renne di Babbo Natale;
- N per gli elfi della fabbrica del Polo Nord.

Il numero degli elfi, infatti, verrà passato come parametro dall'utente.
I parametri che condividono i threads del programma sono i seguenti:

- ***elves***: il contatore degli elfi che hanno riscontrato problemi;
- ***reindeer***: il contatore delle renne che sono tornate dalla vacanza ai tropici;
- ***santaSem***: il semaforo che rappresenta lo stato di Santa Claus (sveglio o addormentato);
- ***reindeerSem***: il semaforo che imbriglia le renne alla slitta;

- ***elfTex***: il mutex che protegge la visita a Santa Claus di altri elfi, mentre lui è occupato con altri di loro;

- ***mutex***: il mutex che protegge l'accesso ai due contatori precedentemente enunciati.

Il **thread di Babbo Natale** eseguirà un ciclo in cui dormirà e attenderà di essere svegliato dagli altri threads. Quando tutti gli altri threads avranno concluso la loro esecuzione, il main thread "morirà" e con lui quello di Babbo Natale.
Una volta sveglio, controllerà in ordine di priorità i motivi per cui è stato sottratto dalla suo riposo. La prima verifica controlla se tutte le sue renne sono tornate dalle vacanze, mentre la seconda controlla il numero di elfi che hanno richiesto il suo aiuto. L'ordine di condizione non è casuale; infatti, come specificato nel testo del problema, se l'ultima renne arriva in contemporanea con la richiesta di aiuto del terzo elfo, essa avrà la priorità.
Il **thread della renna** incrementerà il contatore delle renne che tornano dalle vacanze e se questa risulta essere l'ultima delle renne, prenderà con sè Babbo Natale. Altrimenti attenderà il semaforo che imbriglia le renne alla slitta con le altre renne nella capanna.
Il **thread dell'elfo** incrementerà il contatore di elfi in richiesta di aiuto. Se il numero di elfi arriva a 3, potranno svegliare Babbo Natale e richiedere supporto. Altrimenti attenderanno l'arrivo di altri compagni. Se il numero N di elfi inseriti dall'utente non è un multiplo di 3 (in C: *N % 3 != 0*) allora esisterà almeno un elfo che non riceverà mai aiuto da Babbo Natale.

1.3 Invocazione programma

Il programma del relativo problema, prende il seguente nome: *santa-claus-problem.c*. Esso prende in input un solo parametro, il quale rappresenta il numero di elfi che si desidera creare. Tuttavia, non si accetta una quantità di elfi inferiore al numero minimo necessario per chiedere aiuto a Babbo Natale; dunque, 3.
Per invocare il programma eseguire i comandi seguenti, dove ELVES rappresenta il numero di elfi:

```
make
./santa-claus-problem ELVES
```

Capitolo 2

The roller coaster problem

2.1 Testo del problema

Supponiamo che ci siano n thread per passeggeri e un thread per auto. I passeggeri aspettano ripetutamente di fare un giro in macchina, che può contenere C passeggeri, dove $C < n$. L'auto gira sui binari solo quando è piena.

2.2 Soluzione

I threads coinvolti sono:

- 1 per l'auto;
- N per i passeggeri.

Il numero dei passeggeri, infatti, verrà passato come parametro dall'utente.
I parametri che condividono i threads del programma sono i seguenti:

- ***boarders***: il contatore dei passeggeri a bordo dell'auto;
- ***unboarders***: il contatore dei passeggeri che sono a terra e aspettano di salire in auto;
- ***boardQueue***: il semaforo su cui i passeggeri attendono di salir sull'auto;
- ***unboardQueue***: il semaforo su cui i passeggeri attendono di scendere dall'auto;
- ***allAboard***: il semaforo che indica se l'auto è piena (capienza dell'auto raggiunta);
- ***allAshore***: il mutex che indica se l'auto è vuota (nessun passeggero a bordo);
- ***mutex***: il mutex che protegge il contatore dei passeggeri a bordo;
- ***mutex2***: il mutex che protegge il contatore dei passeggeri a terra.

Il **thread dell'auto** esegue un ciclo infinito in cui carica e scarica i passeggeri dopo ogni giro di pista. A causa di questo loop, non si attende la conclusione del thread e terminerà assieme al main thread.
L'auto inizia a caricare i passeggeri, in attesa del semaforo *boardQueue*, garantendo la sua piena capacità. Successivamente attende che tutti i passeggeri siano saliti, fermandosi al semaforo *allAboard*. Quando è tutto pronto, esegue un giro di pista. Lo scarico dei passeggeri avviene allo stesso modo del carico. L'unica differenza sono i semafori coinvolti: *unboardQueue* e *allAshore*.
Il **thread del passeggero** attende l'imbarco al semaforo *boardQueue*. Una volta che sale a bordo incrementa il numero di passeggeri e controlla se è l'ultimo di essi. In questo caso, notifica al conducente di procedere al giro di pista tramite il semaforo *allAboard* e azzera il numero di passeggeri. Lo stesso procedimento avviene per lo sbarco, coinvolgendo semafori differenti: *unboarderQueue* e *allAshore*.
Il giro di pista avviene solo se l'automobile è completamente piena. Se il numero di passeggeri immesso dall'utente non fosse un multiplo della capacità dell'auto, si incombe in STARVATION. Dunque, ci sarà almeno un passeggero che si troverà a bordo del veicolo e sarà in attesa di altri suoi compagni che non arriveranno mai.

2.3 Invocazione programma

Il programma del relativo problema, prende il seguente nome: *roller-coaster-problem.c*.
Esso prende in input un solo parametro, il quale rappresenta il numero di passeggeri che si desidera creare. Tuttavia, non si accetta una quantità di passeggeri inferiore alla capacità del veicolo.
Per invocare il programma eseguire i comandi seguenti, dove PASSENGERS rappresenta il numero di passeggeri:

```
make
./roller-coaster-problem PASSENGERS
```

Capitolo 3

The unisex bathroom problem

3.1 Testo del problema

Stava lavorando in un cubicolo nel seminterrato di un monolite di cemento, e il bagno delle donne più vicino era due piani più in alto. Ha proposto all'Uberboss di convertire il bagno degli uomini sul suo piano in un bagno unisex, un po' come in Ally McBeal. L'Uberboss ha acconsentito, a condizione che vengano mantenuti i seguenti vincoli di sincronizzazione:

- *Non possono esserci uomini e donne in bagno contemporaneamente.*
- *Non dovrebbero mai esserci più di 3 dipendenti che sprecano tempo in azienda in bagno.*

3.2 Soluzione

I threads coinvolti sono:

- N per gli uomini;
- M per le donne.

I numeri N e M vengono decisi casualmente, in base all'input passato dall'utente (si veda la sezione 3.3 Invocazione programma).
I parametri che condividono i threads del programma sono i seguenti:

- ***femaleInside***: il contatore delle donne in bagno;
- ***maleInside***: il contatore degli uomini in bagno;
- ***femaleSwitch***: il semaforo che permette di escludere gli uomini dalla stanza quando vi sono delle donne all'interno;
- ***maleSwitch***: il semaforo che permette di escludere le donne dalla stanza quando vi sono degli uomini all'interno;
- ***empty***: il semaforo che indica se la stanza è vuota o meno;
- ***femaleMultiplex***: il semaforo che assicura che la capienza della stanza non venga superata dal numero di donne all'interno;

- ***maleMultiplex***: il semaforo che assicura che la capienza della stanza non venga superata dal numero di uomini all'interno;

Il **thread della donna** non appena viene creato, cerca di entrare nella stanza. Attende al semaforo *femaleSwitch* se altre donne stanno entrando. Una volta al suo interno incrementa il numero di donne all'interno della stanza e se il numero è pari ad uno (prima donna a presentarsi all'ingresso del bagno), segnala al primo uomo che cerca di entrare in bagno, che deve attendere al semaforo *empty* (stanza non più vuota). Successivamente libera le altre donne ferme sul *femaleSwitch* e si entra effettivamente nella stanza passando il semaforo *femaleMultiplex*. Una volta conclusi i propri bisogni all'interno della toilette, per uscire decrementa il contatore delle donne e se anche l'ultima donna ha lasciato la stanza, allora si liberano gli uomini fermi in attesa del semaforo *empty* (stanza vuota).
Il **thread dell'uomo** funziona esattamente come quello della donna, coinvolgendo i semafori e i contatori relativi agli uomini.
Tuttavia, questa soluzione presenta un problema di STARVATION. Il primo sesso che guadagna il primo accesso al bagno, avrà la precedenza su l'altro. Ovvero, finchè ci sarà quel sesso all'interno della stanza, gli altri colleghi dello stesso avranno la precedenza sull'altro sesso. Non viene rispettata la coda di arrivo.
Questo accade poichè una volta superato il semaforo "switch" di ingresso, tutti gli impiegati del sesso all'interno attenderanno al semaforo di ingresso effettivo nella stanza ("multiplex").
Per far rispettare l'ordine di arrivo degli impiegati, si introduce un altro semaforo:

- ***turnstile***: il semaforo che permette il rispetto della coda;

Quando un impiegato si presenta all'ingresso del bagno, anzichè attendere sullo "switch" del proprio sesso, attende al semaforo *turnstile*. Grazie ad esso si risolve il problema di STARVATION.

3.3 Invocazione programma

I programmi del relativo problema, prendono i seguenti nomi: *unisex-bathroom-problem.c* e *unisex-bathroom-problem-without-starvation.c*. Essi prendono in input un solo parametro, il quale rappresenta il numero di impiegati che si desidera creare. Tuttavia, non si accetta una quantità di impiegati inferiore o uguale a zero.
Per rendere l'ordine di arrivo all'ingresso del bagno di un thread uomo o donna casuale, si genera un numero casuale [0, 1] che decreterà il thread da istanziare.
Per invocare il programma *unisex-bathroom-problem.c* eseguire i comandi seguenti, dove EMPLOYEES rappresenta il numero di impiegati:

```
make
./unisex-bathroom-problem EMPLOYEES
```

Per invocare il programma *unisex-bathroom-problem-without-starvation.c* eseguire i comandi seguenti, dove EMPLOYEES rappresenta il numero di impiegati:

```
make
./unisex-bathroom-problem-without-starvation EMPLOYEES
```

Capitolo 4

The sushi bar problem

4.1 Testo del problema

Immagina un sushi bar con 5 posti a sedere. Se arrivi mentre c'è un posto vuoto, puoi prendere posto immediatamente. Ma se arrivi quando tutti i posti sono pieni, significa che stanno cenando tutti insieme e devi aspettare che l'intero gruppo se ne vada prima di sederti.

4.2 Soluzione

I threads coinvolti sono:

- N per i clienti;

Il numero N viene definito dall'utente, nella fase di invocazione del programma.
I parametri che condividono i threads del programma sono i seguenti:

- ***eating***: il contatore dei clienti che sono seduti al bar e stanno mangiando;
- ***waiting***: il contatore dei clienti che attendono di sedersi per mangiare;
- ***mutex***: il mutex che protegge i due contatori;
- ***block***: il semaforo dove i clienti che entrano nel bar devono attendere per sedersi;
- ***must_wait***: la variabile booleana che indica se il bar è al completo oppure no.

Per questo problema vengono proposte due soluzioni e una "non-soluzione".

4.2.1 Non-soluzione

Il **thread cliente** acquisisce il mutex e controlla se deve attendere prima di sedersi. Se non deve attendere: aggiorna il numero di persone sedute a mangiare e imposta la variabile di attesa a true se il bar è pieno. Dunque, mangia e rilascia il mutex. Se deve attendere: aggiorna il numero di persone in attesa, rilascia il mutex e attende al semaforo *block*. Una volta conclusa l'attesa, acquisisce il mutex e aggiorna

il contatore delle persone in attesa.
Una volta concluso il pasto, il thread acquisisce un'ultima volta il mutex e aggiorna il contatore di persone al tavolo. Se è l'ultimo a concludere, allora libererà i threads in attesa su *block* e imposterà *must_wait* a false.
Tuttavia, questa soluzione presenta un problema che viola i vincoli di sincronizzazione. Se esistono dei clienti in attesa al semaforo *block* e altri che attendono l'acquisizione del mutex, allora nel momento in cui vengono liberati i threads su *block* devono competere per l'acquisizione del mutex. Se dovessero perdere la competizione, ci potrebbero essere più di 5 threads nella sezione critica.

4.2.2 Soluzione 1

Questa soluzione risolve il problema critico, reso noto nella "non-soluzione". Ovvero, rimuove la riacquisizione del mutex e delega l'aggiornamento dei contatori *waiting* e *eating* all'ultimo cliente che lascia il party.
Poichè l'aggiornamento dello stato del contatore *eating* viene svolto dall'ultimo cliente che si alza dal tavolo, i nuovi threads vedono lo stato corretto e si bloccano se devono attendere prima di sedere.
Tuttavia, questo approccio di aggiornamento dello stato risulta difficile da verificare.

4.2.3 Soluzione 2

La seconda, e ultima, soluzione implementata si basa su una nozione controintuitiva in cui possiamo trasferire un mutex da un thread all'altro. Ovvero, un thread esegue il lock sul mutex e un altro esegue l'unlock su di esso. In questo modo, quando un thread bloccato in attesa del semaforo *block* si risveglia, non deve riacquisire il mutex (come nella "non-soluzione") ma possiede già il mutex in quanto gli viene passato dal thread che lo sblocca.
L'ultimo cliente aggiorna lo stato della variabile *must_wait*.

4.3 Invocazione programma

I programmi del relativo problema, prendono i seguenti nomi: *sushi-bar-problem-non-solution.c*, *sushi-bar-problem-solution-1.c* e *sushi-bar-problem-solution-2.c*. Essi prendono in input un solo parametro, il quale rappresenta il numero di clienti che si desidera creare. Tuttavia, non si accetta una quantità di clienti inferiore o uguale a zero.
Per invocare il programma *sushi-bar-problem-non-solution.c* eseguire i comandi seguenti, dove CUSTOMERS rappresenta il numero di clienti:

```
make
./sushi-bar-problem-non-solution CUSTOMERS
```

Per invocare il programma *sushi-bar-problem-solution-1.c* eseguire i comandi seguenti, dove CUSTOMERS rappresenta il numero di clienti:

```
make
./sushi-bar-problem-solution-1 CUSTOMERS
```

Per invocare il programma *sushi-bar-problem-solution-2.c* eseguire i comandi seguenti, dove CUSTOMERS rappresenta il numero di clienti:

```
make
./sushi-bar-problem-solution-2 CUSTOMERS
```

Conclusione

L'obiettivo posto all'inizio della tesina riguarda l'implementazione dei quattro esercizi proposti dal libro "The Little Book of Semaphores" in linguaggio C. Tale implementazione è stata svolta seguendo le specifiche del testo. Inoltre, ho raggruppato le informazioni di un thread all'interno di una struct e ho creato una funzione di creazione di thread. Entrambe sono state inserite all'interno del file *my-thread.h*.
La struct *ThreadInfo* contiene il thread, il suo id, la *start_routine* e la sua etichetta (massimo 25 caratteri). Quest'ultima, in particolare, risulta molto utile per identificare il thread nel ciclo in cui si esegue *pthread_join()* per l'attesa della sua terminazione.
La funzione *createThread()* si occupa di istanziare il thread e di informare l'utente a console del risultato di essa. Grazie a questa funzione è possibile ridurre il codice ridondante.

Bibliografia

[1] Allen B. Downey, Franklin W. Olin College of Engineering *The Little Book of Semaphores*, Green Tea Press.