

Liveness

Tuesday, 21 March 2023 10:33

Distinguiamo 2 concetti:

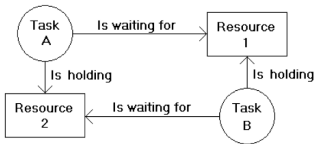
- **Safety:** non si raggiunge uno stato incoerente aka no sezioni critiche non gestite
- **Liveness:** Gli agenti riescono a progredire nella loro elaborazione, aka no problemi

Cioè non devono accadere:

- **Deadlock**

Diversi agenti sono in attesa di un altro l'un con l'altro

Ed essendo tutti in attesa nessuno può generare l'evento di sblocco



Condizioni che possono farlo verificare:

- **Mutua esclusione:** 1 risorsa non è condivisibile contemporaneamente tra 2 agenti
- **Hold and wait/Accumulazione incrementale:** Gli agenti possono richiedere un'altra senza rilasciare la prima
- **No preemption sulle risorse condivise:** una risorsa può essere rilasciata solo volontariamente da un'attività concorrente
- **Attesa circolare:** Immagine precedente
Nota: se c'è un'attesa circolare ma nessuno mantiene una risorsa per l'altro, si allora
- Per ogni risorsa condivisa esiste 1 unica istanza

- **Starvation**

Noi siamo in attesa di una risorsa che però, anche se viene liberata non ci viene mai assegnata, e quindi non la riceviamo mai.

Può accadere es

nelle code a priorità senza olding (aka la priorità sale più sei dentro alla coda)



- **Livelock**

Simile al deadlock, non siamo bloccati però non riusciamo a continuare

Es. abbiamo due handshake, uno invia "ciao" e continua dopo "Ehyla!", l'altro con "hello" e continua dopo "hello back!"

Quindi non abbiamo un coordinamento tra 2 sistemi

Significato:

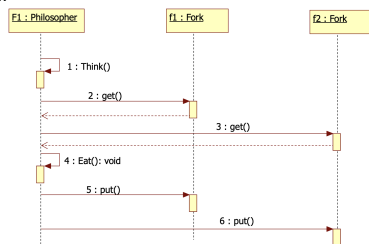
1. Non avviene mai un deadlock, e quindi il processo raggiungerà sempre a raggiungere sezione critica e quindi progredire
2. Siamo liberi sia da deadlock che starvation

Esempi:

- 5 filosofi con 5 piatti 5 bastoncini
- Ogni filosofo ha bisogno di 2 bastoncini per mangiare
Che però non sono condivisibili
- Il piatto diventa pieno una volta svuotato
- Iniziamo:
- Le soddisfazioni della deadlock sono vere:
 - **Mutua esclusione:** Le bacchette non sono condivisibili
 - **Hold and wait/Accumulazione incrementale:** Si siccome mangiare aspetta
 - **No preemption sulle risorse condivise:** Si siccome mangia
 - **Attesa circolare:** Si siccome uno aspetta sempre con 1 bacchetta in mano
 - Esiste 1 sola risorsa condivisa: si

- Quindi può succedere un deadlock, quando? Quando tutti i filosofi hanno 1 bacchetta

Diagramma:



- **Codice:**

```
class Philosopher extends Thread {
    int identity;
    Fork left, right;
    Philosopher(int id, Fork left, Fork right) {
        this.identity = id; this.left = left;
        this.right = right;
    }
    public void run() {
        while (!isInterrupted()) {
            try {
                sleep(100*Math.random());
                right.get();
                sleep(500);
                left.get();
                sleep(50*Math.random());
            } catch (InterruptedException e) {}
        }
    }
}
```

```
class Fork {
    private boolean taken=false;
    private int identity;
    Fork(int identity) {
        this.identity = identity;
    }
    synchronized void put() {
        taken=false;
        notify();
    }
    synchronized void get() throws InterruptedException {
        while (taken && this.identity == Thread.currentThread().getId()) {}
        taken=true;
    }
}
```

```

        right.put();
        left.put();
    } catch (InterruptedException e)
    {break;}
}

}

while (taken)
    wait();
    taken=true;
}

}

https://replit.com/@miclav/Filosof-a-cena

public class Main {
    public static void main(String args[]){
        Fork forks[] = new Fork(5);
        Philosopher phils[] = new Philosopher(5);

        for(int c=0; c<5; forks[c] = new Fork(c), c++);

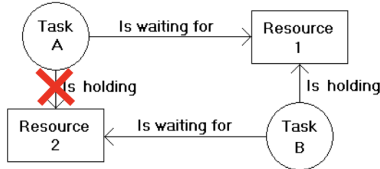
        for(int c=0; c<5; c++){
            phils[c] = new Philosopher(c, forks[c], forks[(c+1)%5]);
        }

        for(int c=0; c<5; phils[c].start(), c++);
        try {
            Thread.sleep(5000);
        }
        catch (InterruptedException e) {}

        for(int c=0; c<5; phils[c].interrupt(),c++);
    }
}

```

- Risoluzione:
 - Rompere simmetria
- Anziché prendere la bacchetta sinistra e destra
 Noi prendiamo la bacchetta più piccola
 Facendo così il filosofo 5 dovrà decidere tra
 Bacchetta 4 e bacchetta 0, e sceglierà bacchetta 0

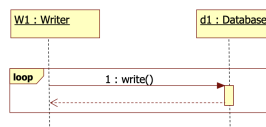
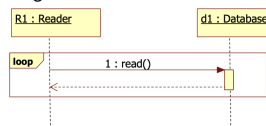


Esempio:

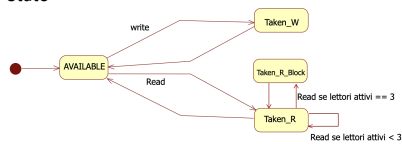
- Abbiamo una risorsa condivisa che possono leggere quella risorsa
- Accesso scrittura è esclusivo -> Nessuno può leggere mentre scrivo

Soluzione:

- Diagramma



- Stato



- Codice

```

public class Database {
    private int content, readers, writeInt, writers;
    Database(int content){
        this.content = content; readers = 0; writers = 0;
    }
    public synchronized void prepareToRead(){
        while(readers>3 || writers>0){
            try {
                wait();
            } catch (InterruptedException e){}
            readers++;
            System.out.println("Readers = " + readers);
        }
    }
    public int read(){
        prepareToRead();
        // Do the reading...
        try{
            Thread.sleep(50);
        } catch (InterruptedException e){}
        int contSnapshot = content;
        doneReading();
        return contSnapshot;
    }
}

class Reader extends Thread {
    private final Database d;
    private int numReads = 0;
    public Reader(Database d) {
        this.d = d;
    }
    public void run() {
        while (numReads < 10) {
            d.read();
            numReads++;
            try{
                Thread.sleep(50);
            } catch (InterruptedException e){}
        }
    }
}
// Writer is analogous...

public synchronized void doneReading(){
    readers--;
    System.out.println("Readers = " + readers);
    notifyAll();
}
public synchronized void write(int content){
    while(readers>0){
        try {
            wait();
        } catch (InterruptedException e){}
        writers++;
        System.out.println("Writers = " + writers);
        this.content = content;
        try{
            Thread.sleep(100);
        } catch (InterruptedException e){}
        writers--;
        writeInt++;
        System.out.println("Writers = " + writers);
        notifyAll();
    }
}

public class Main {
    public static void main(String[] args) {
        Database database = new Database(0);
        Reader r1, r2;
        r1 = new Reader(database);
        r2 = new Reader(database);
        Writer w1, w2;
        w1 = new Writer(database, 1);
        w2 = new Writer(database, 2);
        r1.start();
        r2.start();
        w1.start();
        w2.start();
        try {
            r1.join();
            r2.join();
            w1.join();
            w2.join();
        } catch (InterruptedException e) {}
        System.out.println("End of Program");
    }
}

```

