Azzolini Riccardo

Problema dei 5 filosofi

1 Problema

Nel problema dei **5 filosofi**, detto anche dei **filosofi a cena**, ci sono (appunto) 5 filosofi che cenano intorno a un tavolo rotondo. Ogni filosofo ha bisogno di due bacchette per mangiare, ma ci sono solo 5 bacchette (tante quante i filosofi), ciascuna a disposizione di (cioè condivisa tra) due filosofi.

Ogni filosofo opera secondo il seguente ciclo:

1. pensa per un po', poi gli viene fame;
2. cerca di procurarsi le bacchette per mangiare;
3. dopo essere riuscito a prendere le due bacchette, il filosofo mangia per un po', poi lascia le bacchette e ricomincia a pensare.

Il problema consiste nello sviluppo di un algoritmo che impedisca situazioni di **deadlock** o **starvation**:

- il deadlock può verificarsi se ciascuno dei filosofi ha in mano una sola bacchetta e attende di prendere l'altra (quindi si ha un'attesa circolare);
- la starvation può avvenire se almeno uno dei filosofi non riesce mai a prendere entrambe le bacchette.

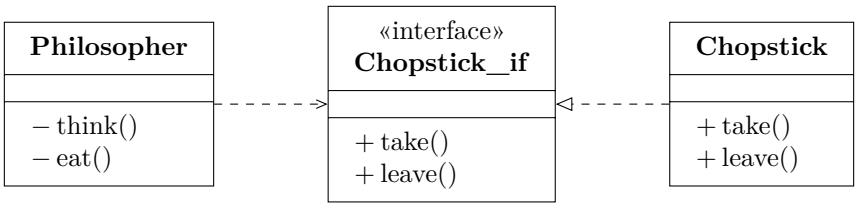
Questo problema rappresenta, sotto forma di metafora, una situazione in cui dei thread / processi concorrenti (i "filosofi") competono per l'uso di risorse limitate, e ciascuno di essi ha bisogno di ottenere (temporaneamente) l'accesso esclusivo a due risorse ("bacchette") contemporaneamente.

Nella sua versione originale, formulata da Edsger Dijkstra nel 1965, il problema faceva riferimento a processori che entravano in competizione per avere l'uso esclusivo di periferiche (risorse) condivise. La metafora dei filosofi è stata successivamente introdotta da Tony Hoare.

2 Applicazione del metodo di progettazione

Per (iniziare a) formulare una possibile soluzione, si applica il metodo di progettazione introdotto in precedenza.

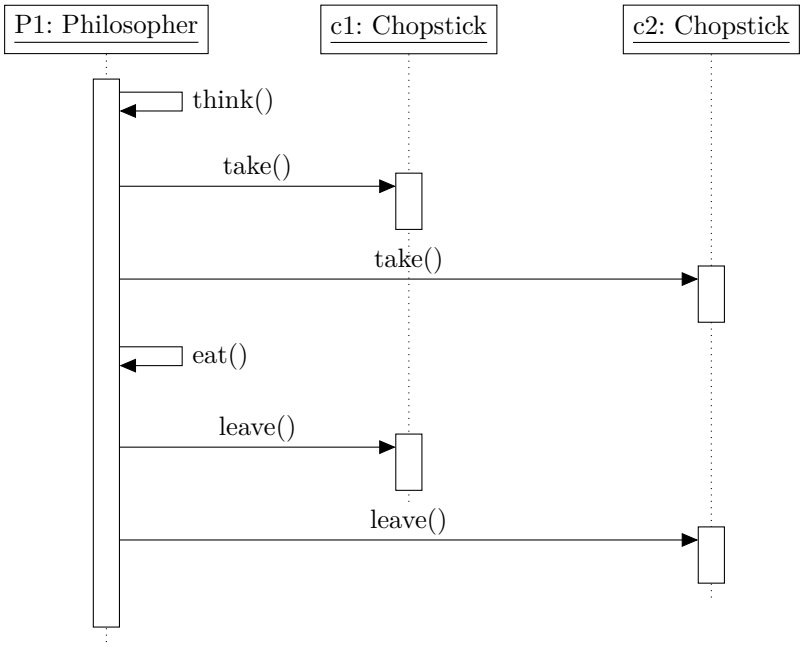
2.1 Class diagram



- La classe `Chopstick`, che rappresenta una singola bacchetta, ha due metodi: `take` per prenderla, e `leave` per rilasciarla.
- I due metodi privati di `Philosopher` corrispondono alle attività che esso svolge: pensare (`think`) e mangiare (`eat`).
- Per mangiare, i filosofi interagiscono con le bacchette mediante l'interfaccia fornita da queste ultime.

2.2 Sequence diagram

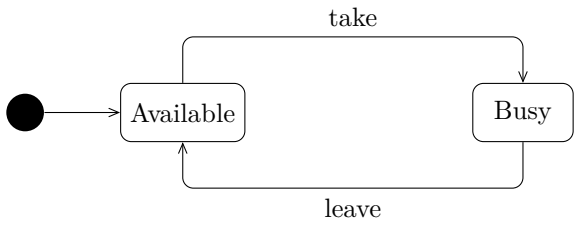
`Philosopher` è un oggetto attivo, quindi il suo comportamento viene descritto mediante un sequence diagram, che si ricava immediatamente dalla descrizione testuale del problema:



In parallelo, gli altri filosofi si comportano in modo analogo (ma ciascuno interagisce con una coppia diversa di bacchette: le due a sua disposizione).

2.3 State diagram

`Chopstick` è un oggetto passivo. Lo state diagram che ne rappresenta il comportamento prevede due stati: libero e occupato.



Non è necessario specificare cosa accadrebbe se si cercasse di lasciare una bacchetta già libera, dato che ciò non succede mai (nessun filosofo più rilasciare una bacchetta se non l'ha prima acquisita).

Al contrario, bisogna decidere cosa succede quando la bacchetta viene richiesta, ma è già occupata. Questa è una delle decisioni fondamentali della soluzione, e sarà quindi discussa in seguito; in particolare, non basta mettere in attesa il chiamante: ciò non risolve i problemi di deadlock e starvation.

2.4 Oggetti da implementare

I filosofi (oggetti attivi) verranno implementati come thread.

Invece, le bacchette (oggetti passivi) sono implementabili come monitor, che devono risultare bloccanti quando necessario (ma, come già detto, ciò non è sufficiente).

3 Non soluzione

La seguente implementazione non si pone il problema del deadlock: ogni filosofo cerca di impossessarsi delle bacchette nell'ordine che gli torna comodo.

```
public class Chopstick {
    private enum State { AVAILABLE, BUSY }

    private final int id;
    private State state = State.AVAILABLE;

    public Chopstick(int id) {
        this.id = id;
    }
```

```
    public synchronized void take() throws InterruptedException {
        while (state == State.BUSY) {
            wait();
        }
        state = State.BUSY;
    }

    public synchronized void leave() {
        state = State.AVAILABLE;
        notify();
        // Non serve notifyAll perché ciascun Chopstick è condiviso
        // tra solo due filosofi, quindi ci sarà al più un filosofo
        // in attesa di prenderlo.
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return "c" + id;
    }
}

public class Philosopher extends Thread {
    private final String name;
    private Chopstick left, right;

    public Philosopher(String name, Chopstick left, Chopstick right) {
        this.name = name;
        this.left = left;
        this.right = right;
    }

    public void run() {
        while (true) {
            try {
                System.out.println("Phil " + name + " thinking");
                Thread.sleep(30); // Tempo passato a pensare
                System.out.println("Phil " + name + " hungry");

                left.take();
                System.out.println(
```

```
                    "Phil " + name + " picked up " + left.getName()
                );
                right.take();
                System.out.println(
                    "Phil " + name + " picked up " + right.getName()
                );

                System.out.println("Phil " + name + " eating");
                Thread.sleep(40); // Tempo passato a mangiare

                left.leave();
                System.out.println(
                    "Phil " + name + " dropped " + left.getName()
                );
                right.leave();
                System.out.println(
                    "Phil " + name + " dropped " + right.getName()
                );
            } catch (InterruptedException e) { return; }
        }
    }
}

public class Table {
    private static final int NUM_PHIL = 5;

    public static void main(String[] args) {
        Chopstick[] sticks = new Chopstick[NUM_PHIL];
        for (int i = 0; i < NUM_PHIL; i++) {
            sticks[i] = new Chopstick(i + 1);
        }
        for (int i = 0; i < NUM_PHIL; i++) {
            Chopstick left = sticks[i];
            Chopstick right = sticks[(i + 1) % NUM_PHIL];
            new Philosopher("P" + (i + 1), left, right).start();
        }
    }
}
```

Il risultato tipico dell'esecuzione di questo programma *è un deadlock*:

```
Phil P1 thinking
Phil P5 thinking
Phil P3 thinking
```

```
Phil P2 thinking
Phil P4 thinking
Phil P5 hungry
Phil P3 hungry
Phil P1 hungry
Phil P4 hungry
Phil P2 hungry
Phil P4 picked up c4
Phil P5 picked up c5
Phil P3 picked up c3
Phil P2 picked up c2
Phil P1 picked up c1
<deadlock>
```

Infatti, ciascun filosofo prende la bacchetta alla propria sinistra: di conseguenza, nessuno trova la bacchetta a destra libera, e tutti i filosofi si bloccano.

4 Soluzione: ordinare le bacchette

Come già detto, una possibile soluzione per evitare il deadlock consiste nell'ordinare l'accesso alle risorse. In questo caso, ciò significa (ad esempio) prendere le bacchette in ordine numerico crescente: se $i < j$, un filosofo deve prendere la bacchetta c_i prima della bacchetta c_j. Nel tentativo di soluzione precedente, i filosofi $P1$, $P2$, $P3$ e $P4$ già lo fanno, mentre $P5$ prende prima c_5 e poi c_1.

Filosofo	1ª bacchetta	2ª bacchetta
P_1	c_1	c_2
P_2	c_2	c_3
P_3	c_3	c_4
P_4	c_4	c_5
P_5	c_5	c_1

Bisogna allora modificare il codice dei filosofi, in modo che anche P_5 prenda prima c_1 e poi c_5. Un possibile modo sarebbe controllare quale delle due bacchette sia quella di indice minore ogni volta che il filosofo le vuole prendere, o, in alternativa, si può eseguire il controllo una volta sola, nel costruttore:

```
public class Philosopher extends Thread {
    private final String name;
    private Chopstick first, second;

    public Philosopher(String name, Chopstick left, Chopstick right) {
```

```
        this.name = name;
        if (left.getId() < right.getId()) {
            this.first = left;
            this.second = right;
        } else {
            this.first = right;
            this.second = left;
        }
    }

    public void run() {
        // Analogo a prima
        // Prende prima first, e poi second
    }
}
```

Così, il deadlock non è più possibile. Ad esempio, se i filosofi P_1, P_2, P_3 e P_4 prendono la bacchetta sinistra (c_1, c_2, c_3, c_4), allora P_5 trova la bacchetta c_1 occupata, e quindi non prende neanche c_5, che quindi rimane libero per P_4, il quale potrà prenderlo e mangiare. Poi, quando P_4 finisce, libera c_4, che viene preso da P_3, e così via.

Tuttavia, nelle applicazione reali, questa soluzione ha dei limiti, soprattutto quando le risorse necessarie non sono note a priori. Ad esempio, se un thread detenesse le risorse 3 e 5, e si rendesse poi conto di avere bisogno anche della 2, dovrebbe rilasciare 3 e 5, e rimettersi in attesa di acquisire 2, e poi ancora 3 e 5. Ciò è piuttosto inefficiente, e si cerca quindi di evitarlo.

5 Soluzione: evitare hold and wait

Un'altra possibilità è evitare la condizione *hold and wait*, facendo in modo che ciascun filosofo prenda, con un'*operazione atomica*, tutte e due le bacchette, se disponibili, o aspetti (senza prenderne neanche una) se invece non sono entrambe disponibili. È allora necessario un *mediatore*, che gestisca l'acquisizione delle risorse "per conto" dei filosofi, osservando lo stato delle bacchette e agendo di conseguenza. A tale scopo, si introduce la classe `Waiter` ("cameriere"):

```
public class Waiter {
    public synchronized void takeTwo(Chopstick i, Chopstick j)
            throws InterruptedException {
        while (!(i.isAvailable() && j.isAvailable())) {
            wait();
        }
        i.take();
```

```
            j.take();
        }

        public synchronized void leaveTwo(Chopstick i, Chopstick j) {
            i.leave();
            j.leave();
            // Se ci sono un filosofo in attesa di i
            // e un altro in attesa di j,
            // bisogna svegliare entrambi.
            notifyAll();
        }
}
```

Siccome `Waiter` è bloccante, non deve più esserlo `Chopstick`, che diventa allora una classe normale, senza metodi `synchronized`:

```
public class Chopstick {
    private enum State { AVAILABLE, BUSY }

    private final int id;
    private State state = State.AVAILABLE;

    public Chopstick(int id) {
        this.id = id;
    }

    public void take() {
        state = State.BUSY;
    }

    public void leave() {
        state = State.AVAILABLE;
    }

    public boolean isAvailable() {
        return state == State.AVAILABLE;
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return "c" + id;
```

```
    }
}
```

Infine, si modificano anche le classi `Philosopher` e `Table`, che devono rispettivamente utilizzare e creare un'istanza condivisa di `Waiter`:

```
public class Philosopher extends Thread {
    private final String name;
    private Waiter waiter;
    private Chopstick left, right;

    public Philosopher(
        String name,
        Waiter waiter,
        Chopstick left,
        Chopstick right
    ) {
        this.name = name;
        this.waiter = waiter;
        this.left = left;
        this.right = right;
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(30); // Tempo passato a pensare
                waiter.takeTwo(left, right);
                System.out.println("Phil " + name + " eating");
                Thread.sleep(40); // Tempo passato a mangiare
                waiter.leaveTwo(left, right);
            } catch (InterruptedException e) { return; }
        }
    }
}

public class Table {
    private static final int NUM_PHIL = 5;

    public static void main(String[] args) {
        Chopstick[] sticks = new Chopstick[NUM_PHIL];
        for (int i = 0; i < NUM_PHIL; i++) {
            sticks[i] = new Chopstick(i + 1);
        }
```

```
        Waiter waiter = new Waiter();
        for (int i = 0; i < NUM_PHIL; i++) {
            Chopstick left = sticks[i];
            Chopstick right = sticks[(i + 1) % NUM_PHIL];
            new Philosopher("P" + (i + 1), waiter, left, right).start();
        }
    }
}
```

Nota: Nel metodo `run` di `Philosopher` sono stati tolti tutti i `println`, tranne `"eating"`, perché, siccome ciascuno di essi prende sempre le due bacchette contemporaneamente, non è utile sapere quali abbia preso (saranno sempre le stesse): l'unica cosa che si vuole verificare dall'output è che tutti i filosofi riescano periodicamente a mangiare.

5.1 Come migliorare questa soluzione

Questa soluzione è già piuttosto efficiente, ma è comunque migliorabile.

Ad esempio, se P_1, P_2 e P_3 hanno fame, e tutte le bacchette che servono a loro (c_1, c_2, c_3, c_4) sono disponibili, si possono avere due situazioni risultanti:

- P_2 mangia (usando c_2 e c_3), mentre P_1 e P_3 aspettano;
- P_1 (che usa c_1 e c_2) e P_3 (che usa c_3 e c_4) mangiano, mentre P_2 aspetta.

La seconda situazione è preferibile, in quanto aumenta il parallelismo e ottimizza l'uso delle risorse.

Le richieste di risorse non devono essere necessariamente chiamate a metodi bloccanti. Ad esempio, potrebbero invece essere messaggi che vengono accodati in una struttura (ad esempio una coda). Così, un thread `Waiter` potrebbe esaminare le richieste presenti e servirle in modo "intelligente", diventando, in pratica, una sorta di scheduler, potenzialmente in grado di gestire priorità, anzianità delle richieste, ecc.

Se, però, le richieste di risorse non sono bloccanti, diventa necessario un meccanismo per informare i thread (filosofi) quando gli è stata riservata una risorsa.

- Una possibilità è usare `interrupt`: i thread si bloccano (ad esempio mediante `sleep`) e, quando vengono svegliati da un'`InterruptedException`, la gestiscono accedendo alla risorsa richiesta.
- Un'altra opzione è che, una volta inviata la richiesta di accesso, i thread continuino a eseguire altre elaborazioni. Poi, ogni tanto, essi controllano se gli sono state assegnate le risorse. Questa soluzione potrebbe risultare più efficiente, ma è delicata da programmare: se un thread non controllasse (abbastanza spesso), le risorse gli sarebbero allocate per niente, impedendo ad altri di usarle.

6 Soluzione basata sull'attesa casuale

Se un filosofo ha acquisito una bacchetta, e, dopo un po' di tempo, non è riuscito a procurarsi anche la seconda, può ipotizzare che si sia creato un deadlock. Per cercare di romperlo, potrebbe rilasciare la bacchetta in suo possesso, e attender un po' di tempo prima di riprovare a impossessarsi delle bacchette.

Questo, però, non basta: se si comportassero allo stesso modo, tutti i filosofi potrebbero rilasciare la loro bacchetta contemporaneamente, e poi allora riprenderebbero tutti contemporaneamente la bacchetta sinistra, riportandosi nella situazione di deadlock. Perché il procedimento funzioni, occorre che l'attesa sia casuale: così, diventa altamente improbabile che tutti i filosofi prendano le bacchette contemporaneamente.

Osservazioni:

- Questo è il modo con cui si risolvono i conflitti nelle reti Ethernet.
- Mentre le soluzioni precedenti erano basate su *deadlock prevention* (cioè evitavano a priori il deadlock), questa è un esempio di *deadlock removal*: si lascia che il deadlock si verifichi, e poi si cerca di risolverlo a posteriori.

  Le tecniche di deadlock removal tendono essere più complesse rispetto a quelle di deadlock prevention, in particolare quando, per risolvere un deadlock, è necessario "disfare" qualche cosa che il thread ha già fatto. Fortunatamente, non è questo il caso: quando si blocca, un filosofo non ha ancora iniziato a mangiare, cioè il thread non ha ancora fatto niente con le risorse (perché non le ha ancora potute acquisire).
- I tempi di attesa vanno tarati adeguatamente, altrimenti il thread che attende le bacchette fa molto lavoro per niente (prende una bacchetta, aspetta, la lascia, aspetta ancora, ecc.), e ciò diventa un overhead per il sistema complessivo.

6.1 Implementazione

Il codice della classe `Chopstick` è in gran parte analogo a quello della "non soluzione", ma è stato aggiunta un'altra versione del metodo `take`, che riceve come argomento un tempo di attesa massimo (`timeout`), e restituisce un valore booleano per indicare se l'acquisizione della bacchetta sia riuscita (`true`) o meno (`false`), entro il timeout specificato.

```
public class Chopstick {
    private enum State { AVAILABLE, BUSY }

    private final int id;
    private State state = State.AVAILABLE;
```

```
    public Chopstick(int id) {
        this.id = id;
    }

    public synchronized void take() throws InterruptedException {
        while (state == State.BUSY) {
            wait();
        }
        state = State.BUSY;
    }

    public synchronized boolean take(long timeout)
            throws InterruptedException {
        if (state == State.BUSY) {
            wait(timeout);
            if (state == State.BUSY) {
                return false;
            }
        }
        state = State.BUSY;
        return true;
    }

    public synchronized void leave() {
        state = State.AVAILABLE;
        notify();
    }

    public int getId() {
        return this.id;
    }

    public String getName() {
        return "c" + id;
    }
}
```

Il filosofo usa `take()` (con attesa illimitata) per prendere la bacchetta di sinistra (perché, cercando – per ora – di prendere una sola risorsa, non può ancora provocare un deadlock), poi invece cerca di prendere la destra con un timeout (qui si sceglie 1 millisecondo):

- se ci riesce, mangia e lascia le bacchette;

- altrimenti, lascia la bacchetta sinistra, attende per un tempo casuale (qui tra 1 e 3 millisecondi[1]), e poi riprova ad acquisire le bacchette, finché non ha successo.

```
import java.util.concurrent.ThreadLocalRandom;

public class Philosopher extends Thread {
    private final String name;
    private Chopstick left, right;

    public Philosopher(String name, Chopstick left, Chopstick right) {
        this.name = name;
        this.left = left;
        this.right = right;
    }

    public void run() {
        while (true) {
            try {
                Thread.sleep(100); // Thinking

                boolean gotSticks = false;
                while (!gotSticks) {
                    left.take();
                    gotSticks = right.take(1);
                    if (!gotSticks) {
                        left.leave();
                        Thread.sleep(
                            ThreadLocalRandom.current().nextInt(1, 4)
                        );
                    }
                }

                System.out.println("Phil " + name + " eating");
                Thread.sleep(200);
                left.leave();
                right.leave();
            } catch (InterruptedException e) { return; }
        }
    }
}
```

[1] La chiamata `nextInt(1, 4)` genera un numero casuale compreso tra 1 e 3, perché il minimo (1) è incluso, ma il massimo (4) è escluso.

Il codice di `Table` non richiede invece modifiche (sempre rispetto alla versione originale introdotta nella "non soluzione").

7 Soluzione di Chandy/Misra

La soluzione proposta da Chandy e Misra è adatta a un numero qualunque di thread (filosofi) e risorse (bacchette), e non necessita di un elemento centrale (`Waiter`), ma richiede invece che i filosofi si parlino (cosa che era esclusa nella formulazione del problema di Dijkstra).

Le regole della soluzione sono le seguenti:

- Per ogni coppia di filosofi che si contendono una risorsa, si crea una bacchetta, e la si dà al filosofo con l'ID minore. Quindi, ogni filosofo avrà esattamente una bacchetta.
- Le bacchette possono essere sporche o pulite. Inizialmente, sono tutte sporche.
- Quando un filosofo vuole mangiare, deve ottenere le risorse che gli mancano dai vicini. A tale scopo, manda una richiesta esplicita.
- Quando un filosofo che detiene una bacchetta riceve una richiesta, la ignora se la bacchetta è pulita (perché ciò significa che non l'ha ancora utilizzata per mangiare); se invece è sporca, la pulisce e la cede.
- Quando un filosofo finisce di mangiare, le sue bacchette sono sporche. A questo punto, se nel frattempo un altro filosofo aveva richiesto una bacchetta, il filosofo che ha finito di mangiare la pulisce e gliela cede.

Chandy e Misra hanno dimostrato che, con il loro algoritmo, non si possono creare attese circolari, a meno che questa non esista fin dall'inizio: se, inizialmente, tutti i filosofi avessero una bacchetta pulita, il sistema sarebbe in deadlock (perché nessuno sarebbe disposto a cedere una bacchetta a un altro). Inizializzare il sistema in modo che i filosofi abbiano bacchette sporche assicura l'assenza di attesa ciclica, e quindi di deadlock.

Questa soluzione permette un elevato grado di concorrenza, ed è applicabile a problemi grandi a piacere. Inoltre, essa risolve il problema della starvation: lo stato sporco/pulito favorisce i processi più affamati, e sfavorisce quelli che hanno appena mangiato (i quali possono cedere le bacchette sporche ad altri).

8 Algoritmo del banchiere

L'algoritmo del banchiere è utile se le risorse non sono distinguibili.[2] Nel caso dei filosofi, ciò significa che in mezzo al tavolo ci sono un insieme di bacchette (non per forza 5), da cui pescarne due per mangiare.

In questa situazione, la soluzione con un mediatore (`Waiter`), al quale un filosofo chiede "in un colpo solo" tutte le bacchette che gli servono, continua a funzionare. È invece necessario l'algoritmo del banchiere nel caso in cui le richieste sono fatte separatamente, cioè un filosofo chiede le bacchette che gli servono una alla volta.

8.1 Criteri dell'algoritmo

L'algoritmo del banchiere è basato sui seguenti criteri:

- Il massimo numero di risorse necessarie è noto a priori (ad esempio, nel caso dei filosofi, è noto che ciascuno di esso abbia bisogno di due bacchette).
- Le risorse sono allocate dinamicamente, nel momento in cui vengono richieste, e si riesce a determinare se la concessione di una risorsa porti o meno al deadlock. Nel caso dei filosofi, si ha un deadlock se sono tutti in hold and wait, ovvero si va in deadlock se si concede l'ultima bacchetta disponibile a un filosofo che, con tale risorsa, non esaurisce le sue necessità (non arriva ad avere due bacchette).

  Allora, si evita la concessione di una risorsa qualora questa provocherebbe un deadlock. Le richieste sono invece soddisfatte se c'è almeno una sequenza di thread che riescono a eseguire senza deadlock (permettendo al sistema di avanzare).
- La somma delle richieste massime può essere maggiore delle risorse disponibili (in realtà, questa condizione è banale, altrimenti sarebbe già impossibile un deadlock, e non servirebbero algoritmi particolari). Nel caso dei filosofi, le richieste massime sono in totale $5 \cdot 2 = 10$, mentre le risorse disponibili sono solo 5.
- Bisogna fare in modo che tutti i thread finiscano (o, in generale, avanzino, se sono potenzialmente infiniti). Per esempio, si può permettere a un thread di procedere se

$$\begin{pmatrix}\text{totale delle risorse disponibili} \\ -\ \text{numero di risorse allocate}\end{pmatrix} \geq \text{necessità massime rimanenti dei thread}$$

[2] Questo algoritmo è detto "del banchiere" perché il denaro è, appunto, una risorsa non distinguibile.

8.2 Applicazione al problema dei 5 filosofi

In questa versione del problema, le bacchette sono in mezzo al tavolo, accessibili a tutti.

Un filosofo può prendere una bacchetta

- se non è l'ultima disponibile,
- oppure, se è l'ultima, ma gli basta per mangiare (perché ne ha già un'altra).

Quando, invece, rimane una sola bacchetta disponibile, e questa non basterebbe al filosofo per mangiare, prendendola rischierebbe un deadlock (perché anche tutti gli altri filosofi potrebbero avere in mano una singola bacchetta), quindi in questo caso deve aspettare.

In alternativa, se il filosofo conosce lo stato degli altri filosofi, può prendere l'ultima bacchetta anche quando non gli basta per mangiare, purché ci sia almeno un filosofo che sta attualmente mangiando: prima o poi, quest'ultimo finirà di mangiare, e libererà quindi altre due bacchette, consentendo così l'avanzamento del sistema.