Università degli Studi dell'Insubria
Dipartimento di Scienze Teoriche e Applicate

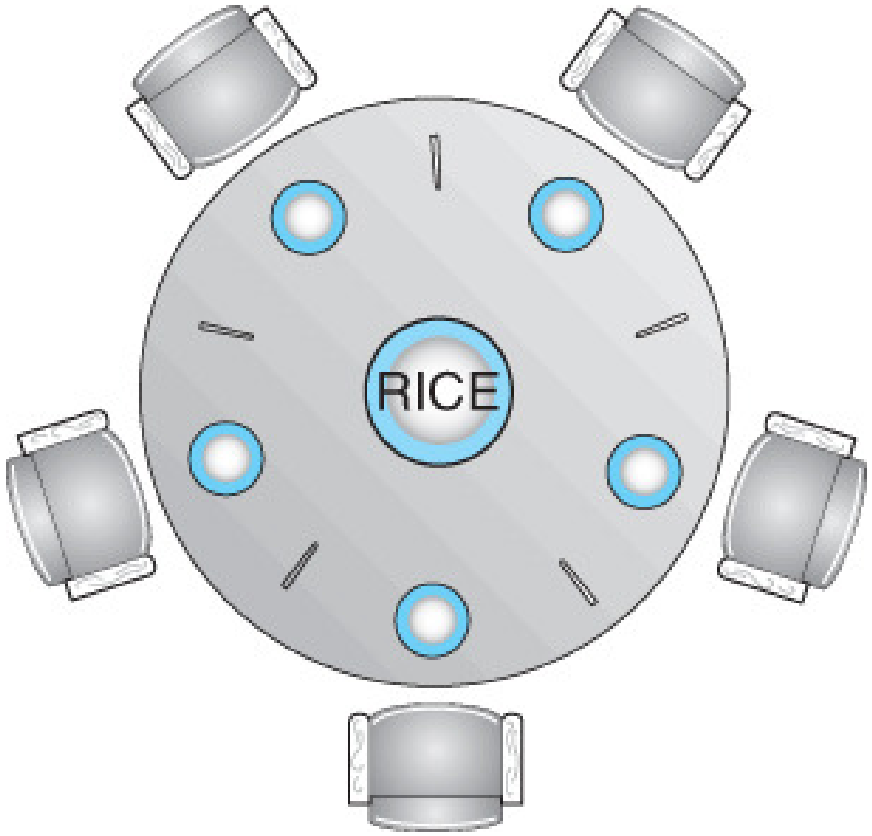
Programmazione Concorrente e Distribuita
Il problema dei 5 filosofi

Luigi Lavazza
Dipartimento di Scienze Teoriche e Applicate
luigi.lavazza@uninsubria.it

Obiettivi della lezione

- In questa lezione verrà introdotto un altro problema di programmazione concorrente.
- Al termine della lezione, si dovrebbe essere in grado di:
  - Conoscere quali sono i problemi di sincronizzazione tipici che si possono verificare in un'applicazione concorrente
  - Saper risolvere questi problemi in diversi modi
  - Riconoscere il problema esemplificato dai 5 filosofi nei contesti reali

Il problema dei filosofi a cena



- Ogni filosofo ha bisogno di due bastoncini per mangiare.
- Ci sono solo 5 bastoncini (tanti quanti i filosofi)
- Ogni bastoncino è a disposizione di (cioè condiviso tra) due filosofi

Il problema dei filosofi a cena

- Ogni filosofo opera secondo il seguente ciclo:
  - Pensa per un po', dopo di che gli viene fame.
  - Cerca di procurarsi i bastoncini per mangiare
  - Dopo essere riuscito a prendere i due bastoncini il filosofo mangia per un po', poi lascia i bastoncini 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 un bastoncino e attende di prendere l'altro.
- La situazione di starvation può verificarsi se uno dei filosofi non riesce mai a prendere entrambi i bastoncini.

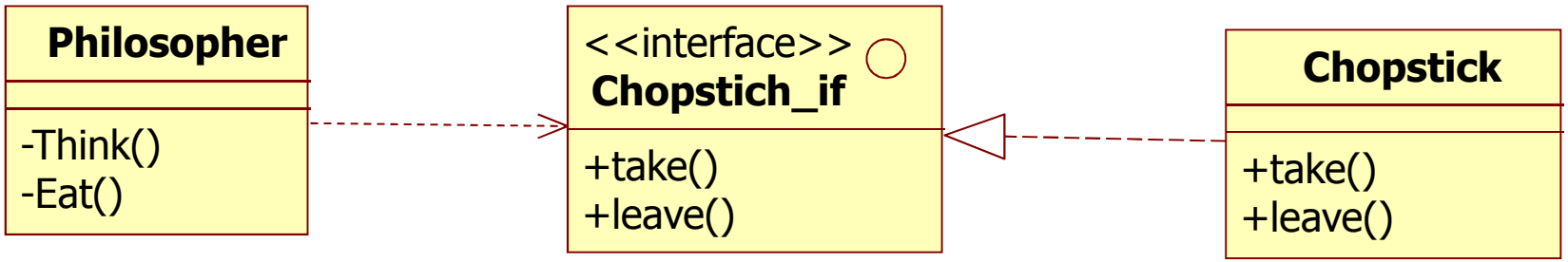
Metafora

- Il problema –formulato nel 1965 da Edsger Dijkstra– faceva riferimento a processori che entravano in competizione per avere l'uso esclusivo (ancorché temporaneo) di risorse (periferiche condivise).
- Tony Hoare ha riformulato il problema nei termini descritti (5 filosofi).

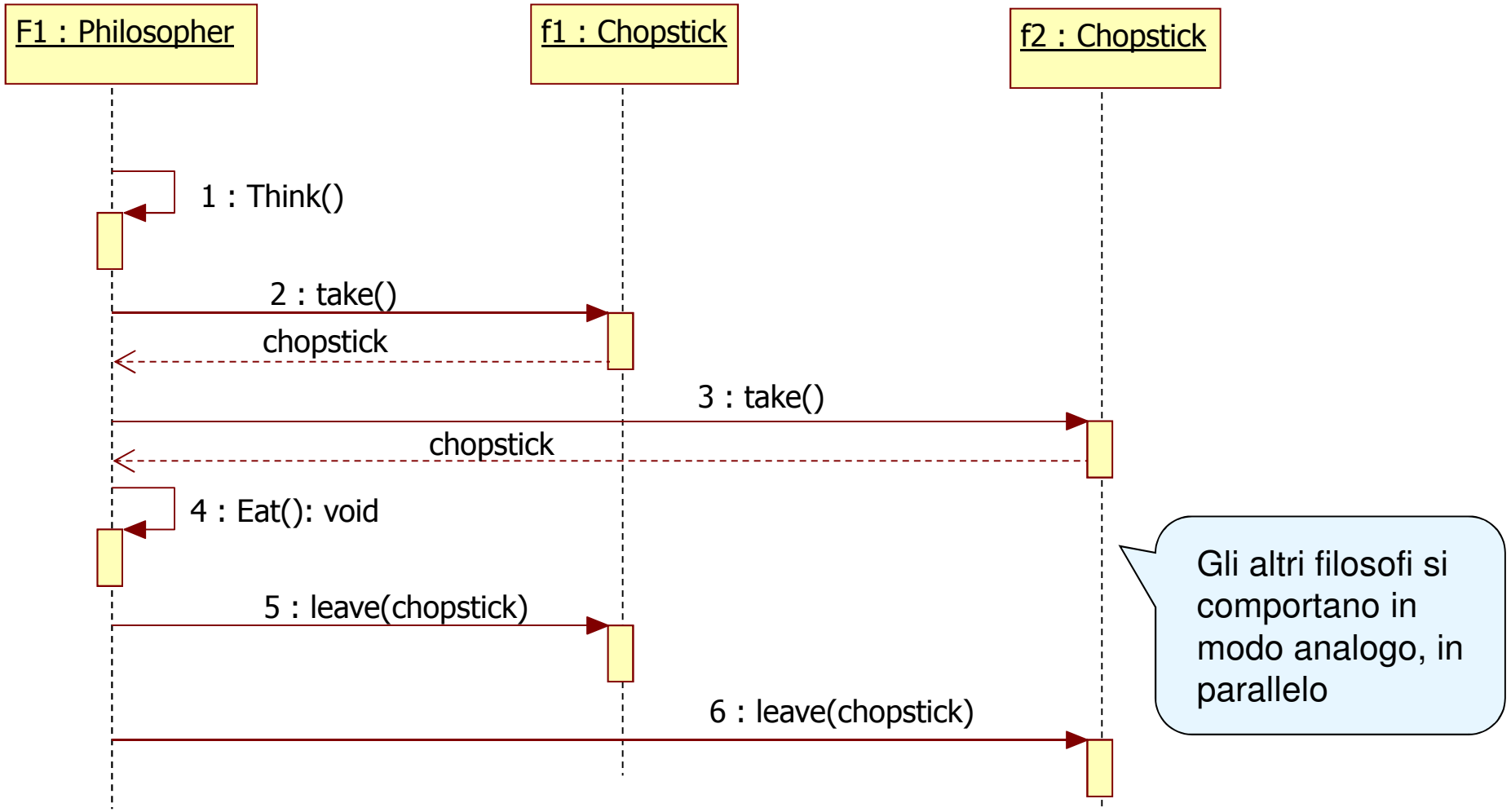
- Caratteristiche fondamentali del problema
  - Accesso concorrente a risorse condividile
  - Non c'è un'autorità centrale che gestisce le risorse
  - I processi (o thread) concorrenti non comunicano tra loro (e non conoscono la situazione globale)

APPLICAZIONE DEL METODO DI DESIGN

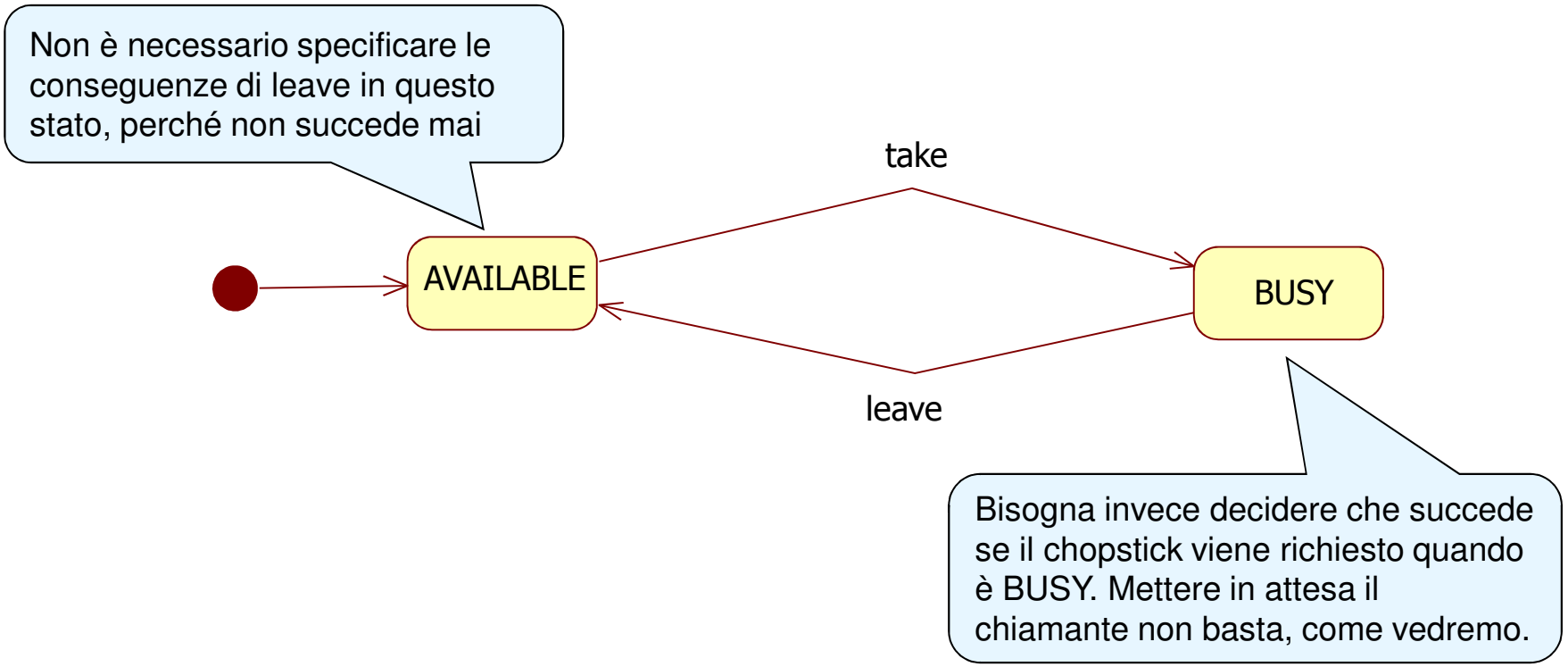
Class diagram



Sequence diagram



State diagram chopstick



Design

- I filosofi vanno implementati come thread
- I bastoncini sono oggetti passivi, implementabili come monitor
  - Devono risultare bloccanti quando necessario (ma non basta...)

Una soluzione scorretta

- Vediamo prima una implementazione che non si pone il problema del deadlock
- Ogni filosofo cerca di impossessarsi dei bastoncini che gli servono nell'ordine che gli torna comodo.

Class Chopstick

```
public class Chopstick {
  public enum State {AVAILABLE, BUSY}
  private State state;
  private int id;
  public Chopstick(int id) {
    this.id = id;
    this.state=Chopstick.State.AVAILABLE;
  }
  public synchronized void take( ) throws InterruptedException {
    while(state==Chopstick.State.BUSY) {
      wait();
    }
    this.state=Chopstick.State.BUSY;
  }
  public synchronized void leave() {
    this.state=Chopstick.State.AVAILABLE;
    notify();
  }
  public String getName() { return "f"+id; }
  public int getId() { return this.id; }
}
```

Class Philosopher

```
public class Philosopher extends Thread {
  private Chopstick right, left;
  private String name ;
  public Philosopher(String id,
                     Chopstick left, Chopstick right) {
    this.name=id;
    this.left=left;
    this.right=right;
  }
  private void writeState(String action, String stickName) {
    System.out.println("Phil "+name+action+stickName);
  }
```

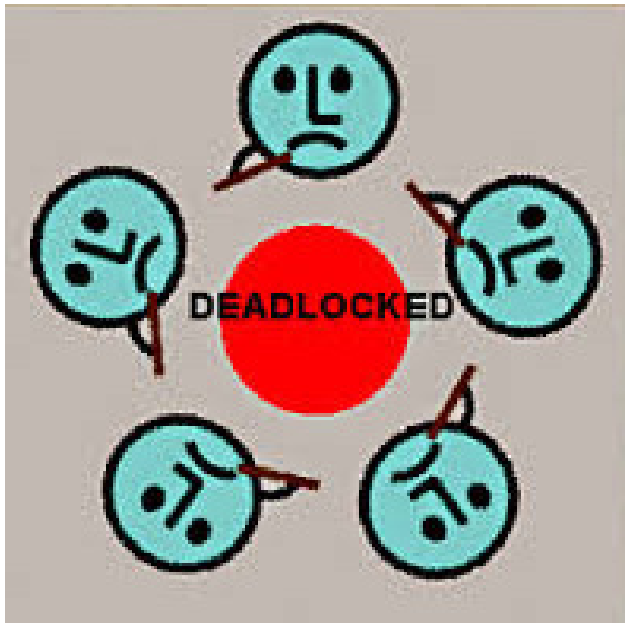
Class Philosopher

```
public void run() {
    while(true) {
      try {
        writeState(": thinking", "");
        Thread.sleep(30);
        writeState(": hungry", "");
        left.take();
        writeState(" picked up ", left.getName());
        right.take();
        writeState(" picked up ", right.getName());
        writeState(": eating", "");
        Thread.sleep(40);
        left.leave();
        writeState(" dropped ", left.getName());
        right.leave();
        writeState(" dropped ", right.getName());
      } catch (InterruptedException e) {return ; }
    }
  }
}
```

Class Table (main)

```
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++)
       new Philosopher("F"+(i+1), sticks[i],
                       sticks[(i+1)%NUM_PHIL]).start();
  }
}
```

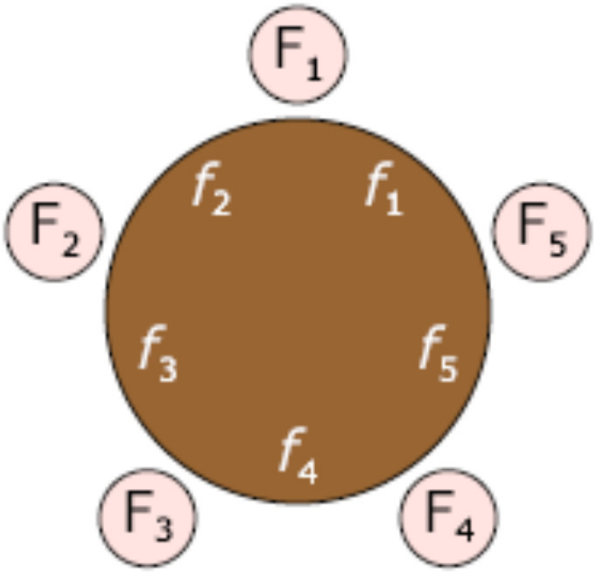
Un possibile risultato




```
Phil F2 thinking
Phil F5 thinking
Phil F4 thinking
Phil F3 thinking
Phil F1 thinking
Phil F2 hungry
Phil F5 hungry
Phil F4 hungry
Phil F3 hungry
Phil F1 hungry
Phil F2 picked up f2
Phil F1 picked up f1
Phil F3 picked up f3
Phil F5 picked up f5
Phil F4 picked up f4
DEADLOCK!
```

Soluzione 1

- Avevamo detto che per evitare il deadlock una possibile soluzione consiste nell'ordinare l'accesso alle risorse.
- Nel nostro caso, i bastoncini devo essere presi in ordine numerico crescente
  - Ogni filosofo prende sempre il bastoncino f_i prima del bastoncino f_j con i < j
- I filosofi F1, F2, F3 e F4 già lo fanno.
- Il filosofo F5 deve prendere prima f1 e poi f5.



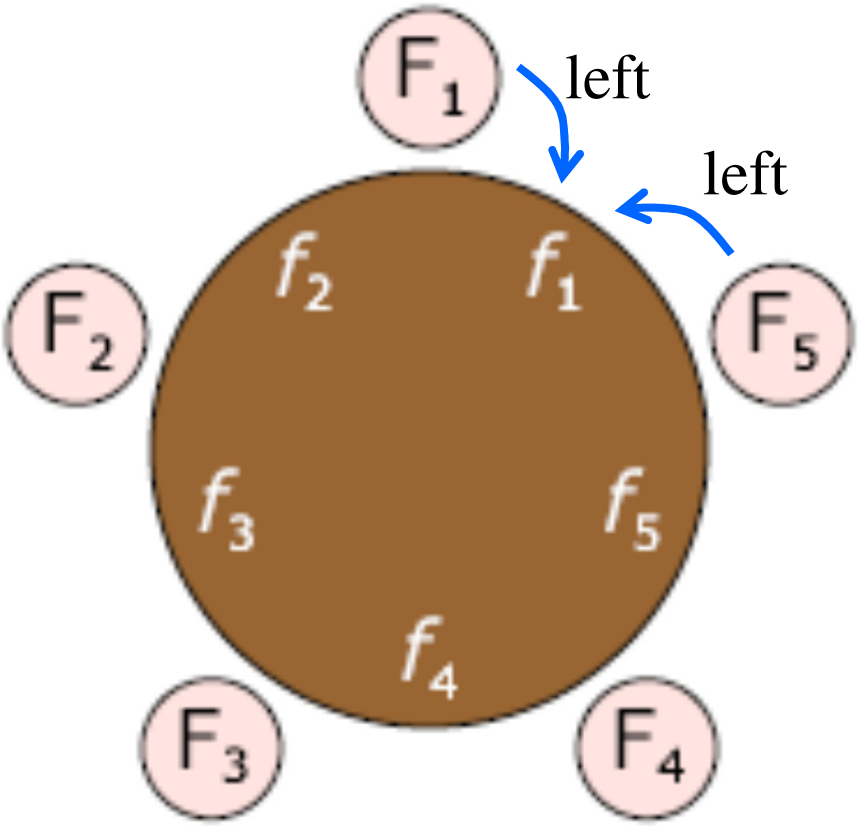
Class Philosopher

```
public class Philosopher extends Thread {
  private Chopstick right, left;
  private String name ;
  public Philosopher(String id,
                     Chopstick left, Chopstick right) {
    this.name=id;
    this.left=(left.getId()<right.getId())?left:right;
    this.right=(left.getId()<right.getId())?right:left;
  }
```

Ogni filosofo prende sempre prima il bastoncino sinistro e poi il destro.
Facciamo in modo che siano ordinati.
NB: il bastoncino «sinistro» non sarà più quello fisicamente a sinistra, ma quello che va preso per primo. Idem per il «destro».

Soluzione 1: risultato

- Il deadlock non è più possibile.
- Se i filosofi F1, F2, F3 e F4 prendono il bastoncino sinistro, F5 trova il bastoncino f1 occupato e quindi non prende f5, evitando così il deadlock
- Infatti f5 resta libero per F4, che potrà prenderlo e mangiare
- Quando F4 finisce, libera f4, che viene preso da F3, ecc.

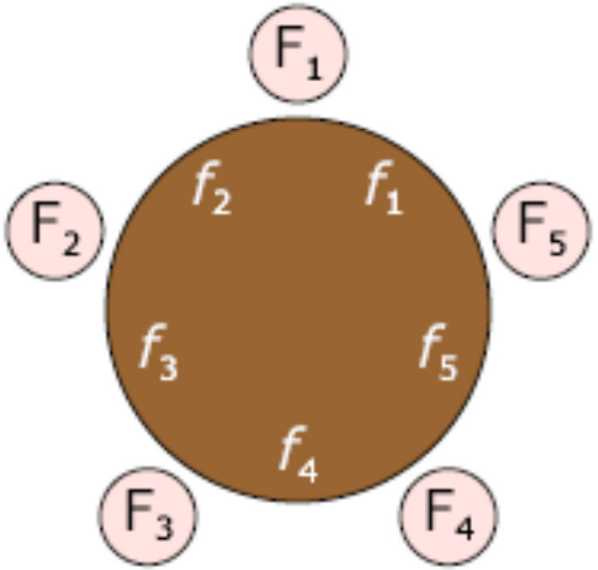


Osservazioni sulla soluzione 1

- La soluzione 1 ha dei limiti, soprattutto quando le risorse richieste non sono note a priori.
- Ad es., se un elaboratore detiene le risorse 3 e 5 e si rende conto di avere bisogno della risorsa 2, dovrebbe rilasciare 3 e 5, e rimettersi in attesa di acquisire la risorsa 2 e poi ancora 3 e 5.
- In diversi casi pratici questo è inefficiente.

Soluzione 2

- Altra possibilità consiste nel evitare la condizione “hold and wait”
- Basta che ogni filosofo prenda (con una operazione atomica) entrambi i bastoncini se disponibili, e aspetti se invece non sono disponibili.
- Ci vuole un mediatore che osservi lo stato dei bastoncini e agisca di conseguenza.
  - A questo scopo introduciamo la classe Waiter (cameriere)



Class Waiter

```
public class Waiter {
  public synchronized void takeTwo(Chopstick i, Chopstick j) {
    while(!(i.isAvailable() && j.isAvailable())) {
      try {
        wait() ;
      } catch (InterruptedException e) { }
    }
    i.take();
    j.take();
  }
  public synchronized void leaveTwo(Chopstick i, Chopstick j) {
    i.leave();
    j.leave();
    notifyAll();
  }
}
```

Class Chopstick

```
public class Chopstick {
  public enum State {AVAILABLE, BUSY}
  private State state;
  private int id;
  public Chopstick(int id) {
    this.id = id;
    this.state=Chopstick.State.AVAILABLE;
  }
  public void take( ) {
    this.state=Chopstick.State.BUSY;
  }
  public boolean isAvailable( ) {
    return (this.state==Chopstick.State.AVAILABLE);
  }
  public void leave() {
    this.state=Chopstick.State.AVAILABLE;
  }
  public String getName() { return "f"+id; }
  public int getId() { return this.id; }
}
```

Poiché Waiter è bloccante, non deve più esserlo Chopstick! Chopstick è una classe normale, senza metodi **synchronized**

Class Philosopher

```
public class Philosopher extends Thread {
  private Chopstick right, left;
  private String name ;
  private Waiter myWaiter;
  public Philosopher(String id, Waiter w,
      Chopstick left, Chopstick right) {
    this.name=id;
    this.left=left;
    this.right=right;
    this.myWaiter=w;
  }
  private void writeState(String action, String stickName) {
    System.out.println("Phil "+name+action+stickName);
  }
```

Class Philosopher

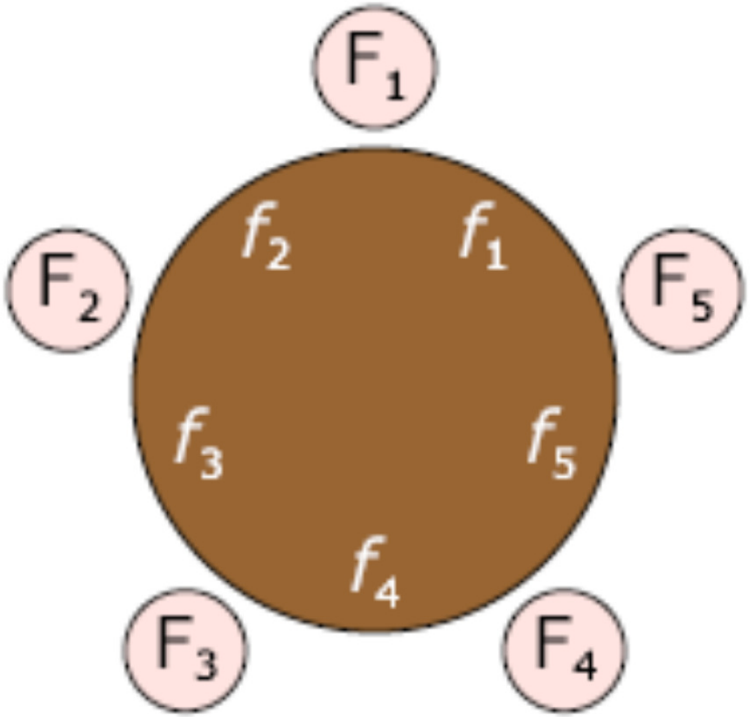
```
public void run() {
  while(true) {
    try {
      Thread.sleep(30);
      Thread.sleep(ThreadLocalRandom.current().nextInt(20,50));
      myWaiter.takeTwo(left, right);
      writeState(": eating", "");
      Thread.sleep(10,30);
      myWaiter.leaveTwo(left, right);
    } catch (InterruptedException e) {return ; }
  }
 }
}
```

Class Table

```
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 w = new Waiter();
       for(int i=0; i<NUM_PHIL; i++)
          new Philosopher("F"+(i+1), w, sticks[i],
                          sticks[(i+1)%NUM_PHIL]).start();
  }
}
```

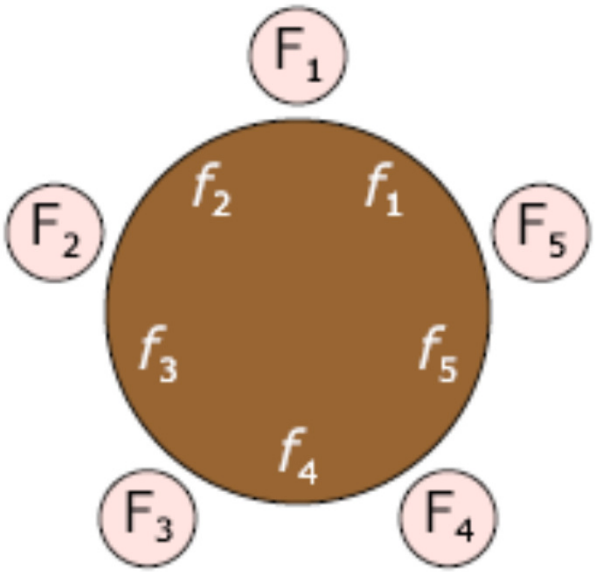
Soluzione 2: risultato

- Il deadlock non è possibile, perché non c'è hold&wait.



La soluzione 2 è migliorabile

- Si consideri la situazione seguente:
  - F1, F2 e F3 hanno fame, quindi richiedono i bastoncini
  - f1, f2, f3 e f4 (cioè tutti i bastoncini che servono a F1, F2 e F3) sono disponibili
- Possibili situazioni risultanti:
  - F2 mangia (con f2 e f3), mentre F1 e F3 aspettano
  - F1 e F3 mangiano (F1 usa f1 e f2, e F3 usa f3 e f4), mentre F2 aspetta
- La seconda situazione è preferibile (aumenta il parallelismo e ottimizza l'uso delle risorse)



Come migliorare la soluzione 2

- Le richieste di risorse non devono essere necessariamente chiamate di metodo bloccanti. Potrebbero essere messaggi che vengono accodati in una struttura (ad es. una coda)
- Un thread Waiter esamina le richieste presenti e le serve in modo intelligente.
  - Praticamente il Waiter può diventare una sorta di scheduler.
  - Può gestire priorità, anzianità delle richieste, ecc.
- Come:
  - Ad esempio usando interrupt
  - I thread
    - mandano le richieste e fanno sleep; gestiscono InterruptedException semplicemente svegliandosi e accedendo alla risorsa richiesta.
    - Alternativamente, mandano la richiesta e continuano l'elaborazione; ogni tanto controllano se le risorse sono disponibili. Situazione delicata da programmare: se il thread non controlla, le risorse gli sono allocate per nulla, impedendo ad altri di usarle.

Altre soluzioni

- Attesa random
- Chandy/Misra
- Algoritmo del banchiere

Soluzione basata sull'attesa casuale

- Se un filosofo ha acquisito un bastoncino e dopo un po' di tempo non è riuscito a procurarsi anche il secondo, può ipotizzare che si sia creato un deadlock.
- Per romperlo, rilascia il bastoncino in suo possesso e attende un po' di tempo prima di riprovare a impossessarsi dei bastoncini.
- Questo non basta: se tutti i filosofi si comportano nello stesso modo, tutti rilasciano il loro bastoncino contemporaneamente e poi tutti riprendono contemporaneamente il bastoncino sinistro, riportandosi nella situazione di deadlock potenziale.
- Perché il procedimento funzioni, occorre che l'attesa sia casuale.
  - Così diventa altamente improbabile che i filosofi prendano i bastoncini contemporaneamente.

- NB: questo è il modo con cui si risolvono i conflitti nelle reti Ethernet.

Class Chopstick

```
public class Chopstick {
  public enum State {AVAILABLE, BUSY}
  private State state;
  private int id;
  public Chopstick(int id) {
    this.id = id;
    this.state=Chopstick.State.AVAILABLE;
  }
  public synchronized void leave() {
    this.state=Chopstick.State.AVAILABLE;
    notify();
  }
  public String getName() {
    return "f"+id;
  }
  public int getId() {
    return this.id;
  }
```

Class Chopstick

```
public synchronized boolean take(long t)
    throws InterruptedException {
  if(t==0){
    while(state==Chopstick.State.BUSY) {
      wait();
    }
    this.state=Chopstick.State.BUSY;
    return (true);
  } else {
    while(state==Chopstick.State.BUSY) {
      wait(t);
      if(state!=Chopstick.State.BUSY){
        break;
      } else {
        notifyAll();
        return (false);
      }
    }
    this.state=Chopstick.State.BUSY;
    return (true);
}}}
```

Class Philosopher

```
import java.util.concurrent.ThreadLocalRandom;
public class Philosopher extends Thread {
  private Chopstick right, left;
  private String name;
  public Philosopher(String n,
      Chopstick left, Chopstick right) {
    this.name=n;
    this.left=left;
    this.right=right;
  }
  private void writeState(String action, String stickName) {
    System.out.println("Phil "+name+action+stickName);
  }
```

Class Philosopher

```
public void run() {
  boolean b, gotSticks;
  while(true) {
    try {
      Thread.sleep(100);  // thinking
      writeState(": hungry", ""); gotSticks=false;
      while(!gotSticks){
        b=left.take(0);
        if(b) {
          gotSticks=right.take(1);
          if(!gotSticks) {
            writeState(": leaving ", left.getName());
            left.leave();
            Thread.sleep(ThreadLocalRandom.current().nextInt(3,30));
          }
        }
      }
      writeState(": eating", ""); Thread.sleep(200);
      left.leave(); right.leave();
    } catch (InterruptedException e) {return ; }
} }
```

Class Table

```
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++)
      new Philosopher("F"+(i+1), sticks[i],
          sticks[(i+1)%NUM_PHIL]).start();
  }
}
```

Osservazioni sulla soluzione

- I tempi di attesa vanno tarati adeguatamente
- Altrimenti il thread che attende i bastoncini fa un sacco di lavoro per niente (prendi un bastoncino, aspetta, lascialo, aspetta ancora, ecc.)
  - Diventa un overhead per il sistema complessivo

Soluzione di Chandy & Misra

- Adatta ad un numero qualunque di processori e di risorse
- Non richiede un elemento centrale (Waiter)
- Richiede che i filosofi si parlino (cosa che era esclusa nella formulazione del problema di Dijkstra)

Regole

- Per ogni coppia di filosofi che si contende una risorsa, crea un bastoncino e dallo al filosofo con l'ID più basso.
  - Quindi ogni filosofo avrà esattamente un bastoncino.
- I bastoncini possono essere sporchi o puliti. Inizialmente sono tutti sporchi.
- Quando un filosofo vuole magiare, deve ottenere le risorse che gli mancano dai vicini. A questo scopo manda una richiesta esplicita.
- Quando un filosofo che detiene un bastoncino riceve una richiesta, la ignora se il bastoncino è pulito; se invece è sporco, lo pulisce e lo cede.
- Quando un filosofo finisce di mangiare, i suoi bastoncini sono sporchi. Se un altro filosofo aveva richiesto un bastoncino, il filosofo che ha finito di mangiare lo pulisce e glielo cede.

Osservazioni sulla soluzione di Chandy & Misra

- Permette un grado elevato di concorrenza ed è applicabile a problemi grandi a piacere
- Risolve il problema della starvation. Lo stato sporco/pulito favorisce i processi più affamati e sfavorisce quelli che hanno appena mangiato (che devono cedere i loro bastoncini sporchi).

- Chandy & Misra hanno dimostrato che con il loro algoritmo non si possono creare attese circolari.
  - A meno che questa non sia creata all'inizio: se inizialmente tutti i filosofi hanno un bastoncino pulito, il sistema è in deadlock
  - Inizializzare il sistema in modo che i filosofi abbiano bastoncini sporchi assicura la mancanza di attesa ciclica.

Algoritmo del banchiere

- Utile se le risorse non sono distinguibili
- Nel caso dei filosofi, in mezzo al tavolo ci sono un insieme di bastoncini da cui pescarne due
- La soluzione col Waiter (un filosofo chiede tutti i bastoncini che gli servono in un colpo solo) funziona anche in questo caso
- Se invece le richieste sono fatte separatamente (un filosofo chiede i bastoncini che gli servono uno alla volta)?

Come evitare il deadlock con l'algoritmo del banchiere

- Il Massimo numero di risorse necessarie sia noto a priori
- Le risorse sono allocate dinamicamente quando richiesto
  - Se concedere una risorsa porta al deadlock, si aspetta
    - Deadlock=tutti i filosofi in hold&wait, quindi si va in deadlock se si concede la risorsa a un filosofo che con quella risorsa non esaurisce le sue necessità
  - Le richieste sono soddisfatte se c'è qualche sequenza di thread deadlock-free
- La somma delle richieste massime può essere maggiore delle risorse disponibili
  - Come nel caso dei filosofi: max richieste = 5×2, risorse disponibili = 5
- Bisogna che ci sia un modo con cui tutti i thread finiscono (o avanzano, se sono potenzialmente infiniti).
- Per esempio: si può permettere a un thread di procedure se (il totale delle risorse disponibili – il numero di risorse allocate) ≥ necessità massime rimanenti dei thread

Algoritmo del banchiere per il problema dei filosofi

- I bastoncini sono in mezzo al tavolo, accessibili da tutti
- Regole:
  - Se non è l'ultimo bastoncino, puoi prenderlo
  - Se è l'ultimo e ti basta per mangiare (perché ne hai già uno) prendilo pure
  - In tutti gli altri casi, aspetta
- Regole alternative (che richiedono la conoscenza non solo dello stato delle risorse, ma anche dello stato degli altri filosofi):
  - Se non è l'ultimo bastoncino, puoi prenderlo
  - Se è l'ultimo e ti basta per mangiare (perché ne hai già uno) prendilo pure
  - Se è l'ultimo e c'è un filosofo che sta mangiando, prendilo pure
    - Perché sicuramente prima o poi il filosofo smette di mangiare e libera altri due bastoncini
  - In tutti gli altri casi, aspetta

Bibliografia

- Molte delle informazioni presenti su questa presentazione sono state estratte dal capitolo 3 di Concurrent and Distributed Programming in Java by Vijay K. Garg