Programmazione e Architetture (Modulo B)

Lezione 16
Deadlock e problemi dell’esecuzione concorrente

Luca Manzoni

Il problema dei filosofi a cena

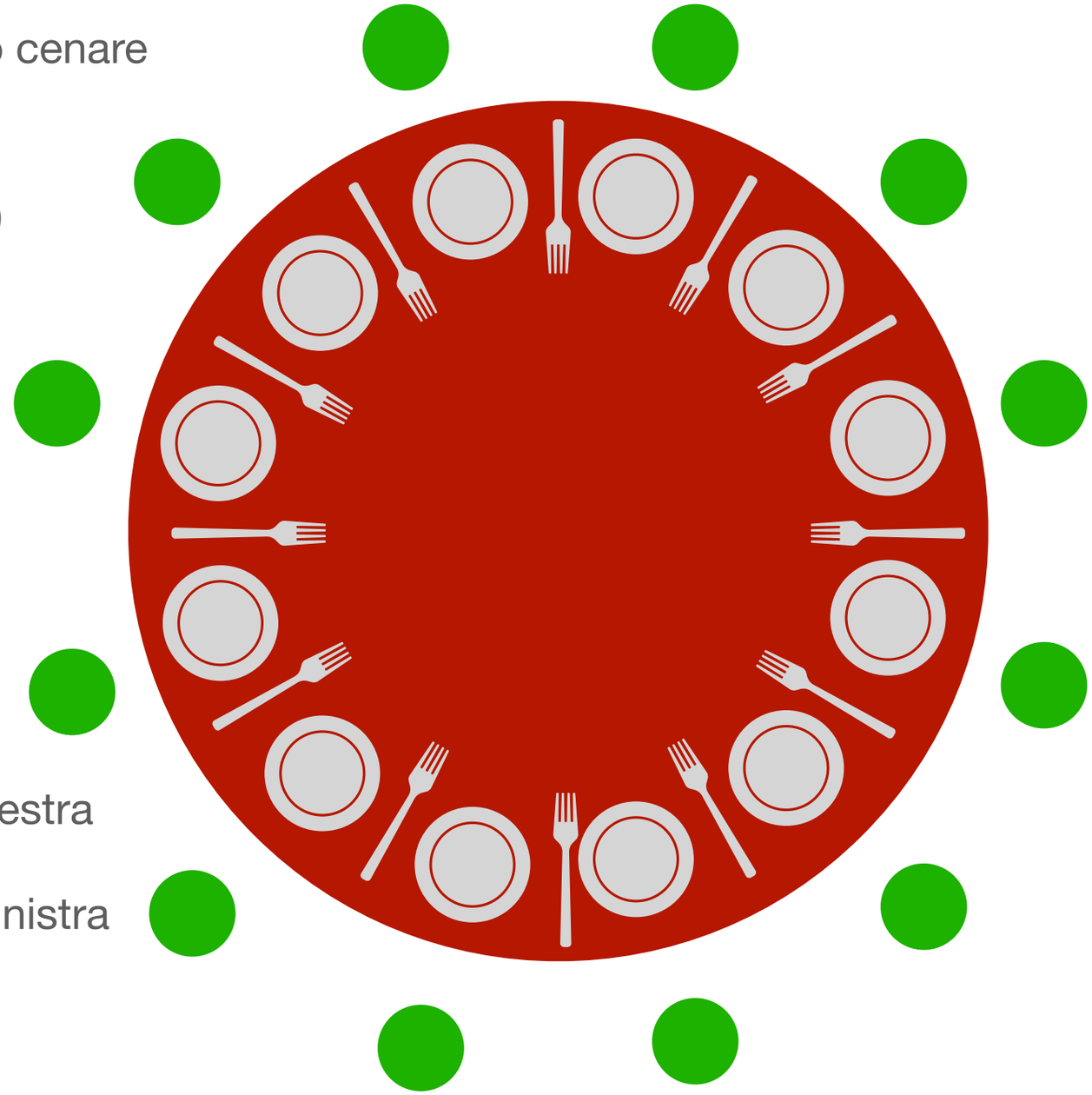
Deadlock al tavolo

C’è un tavolo rotondo con n filosofi che vogliono cenare

Ogni filosofo ha davanti un piatto (quindi n piatti)

Tra ogni coppia di piatti vi è esattamente una forchetta (quindi n forchette)

Per mangiare ogni filosofo ha bisogno di **due** forchette



L’algoritmo che usano i filosofi è il seguente:

1. Se è disponibile, prendi la forchetta alla tua destra altrimenti attendi che si liberi
2. Se è disponibile, prendi la forchetta alla tua sinistra altrimenti attendi che si liberi
3. Mangia
4. Finito di mangiare libera le due forchette

Il problema dei filosofi a cena

Deadlock al tavolo

Una possibile sequenza di operazioni è la seguente:

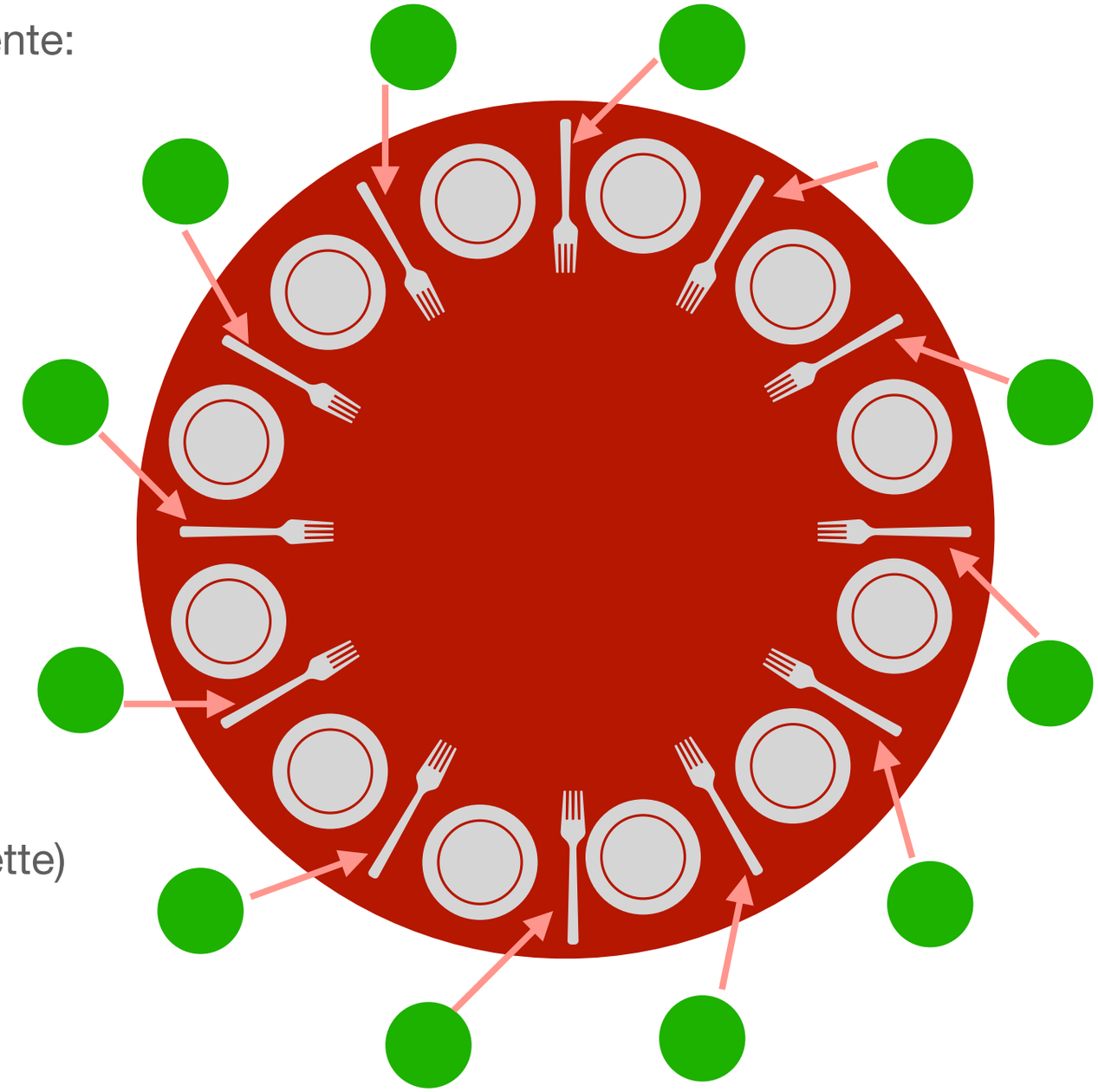
Ogni filosofo prende la forchetta alla sua destra

Ogni filosofo prova a prendere la forchetta alla sua sinistra

La forchetta alla sua sinistra è occupata, quindi ogni filosofo attende che si liberi

Nessuno delle forchette si libererà mai, dato che tutti i filosofi sono in attesa e nessuno può finire di mangiare (liberando quindi le forchette)

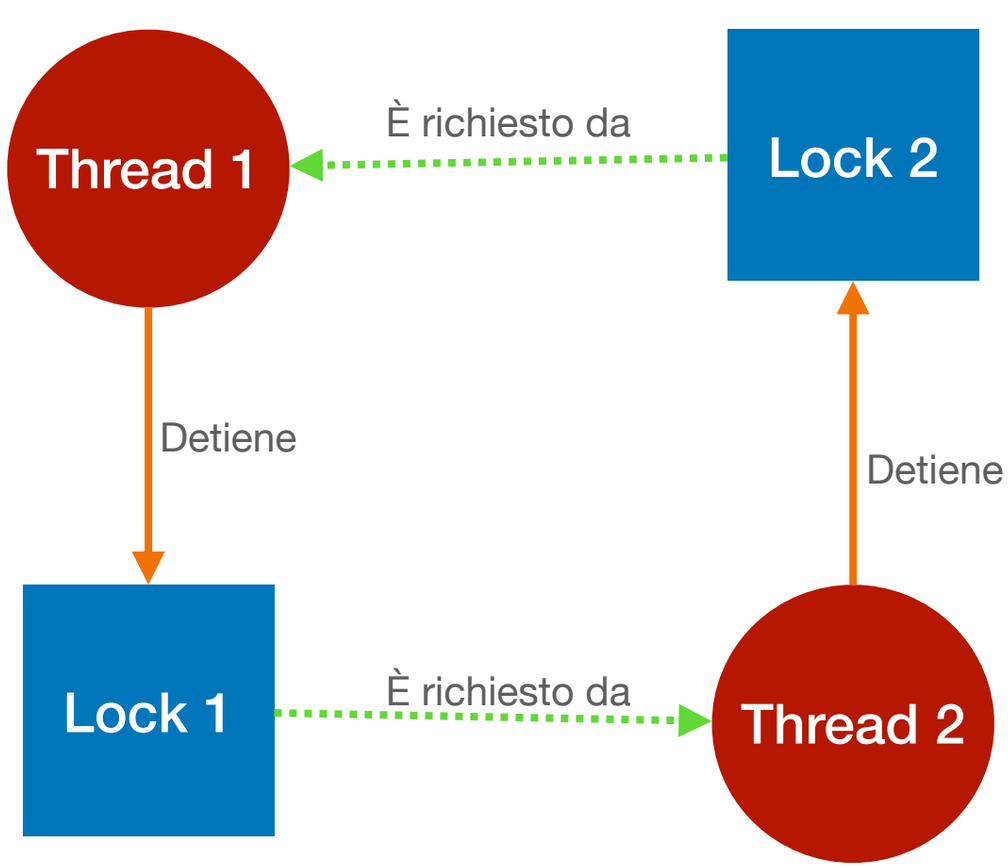
Deadlock!



Deadlock

Cosa è?

- Un deadlock avviene quando vi è un gruppo di thread in cui ogni thread attende che un altro thread (incluso sé stesso) compia una azione.



Requisiti per un dealock

Condizioni di Coffman (1971)

- **Mutua esclusione**. I thread hanno un uso esclusivo delle risorse che acquisiscono (gli altri thread non possono usarle).

- **Hold-and-wait**. Quando un thread ha accesso esclusivo a una risorsa può attendere in attesa che un'altra risorsa si liberi.

- **No preemption**. Non è possibile togliere forzatamente una risorsa a un thread. Da non confondere con la preemptive scheduling, sono due cose diverse (in questo caso non possiamo togliere forzatamente un lock a un thread).

- **Attesa circolare**. Esiste una sequenza di thread in cui ogni thread detiene una o più risorse del thread successivo nella sequenza.

Evitare i deadlock

Strategie per non avere deadlock

- Notate che non è certo che il deadlock si verifichi a ogni esecuzione. Per esempio se un filosofo riesce a prendere due forchette lui può proseguire e non si verificano le condizioni per il deadlock

- Questo non è positivo! Diventa difficile riprodurre le condizioni che hanno portato al deadlock e fare il debug.

- In generale ci basta rompere una delle quattro condizioni di Coffman per evitare che si verifichino deadlock…

- …oppure sperare che non i deadlock siano abbastanza infrequenti che possiamo non occuparcene.

Strategia 1

Evitare la mutua esclusione

- I deadlock avvengono perché abbiamo dei lock…
- …se eliminiamo i lock non possiamo avere deadlock!
- Però abbiamo introdotto i lock per evitare le race condition
- Con adeguato supporto hardware è possibile implementare delle strutture dati lock-free che non hanno problemi di race condition
- Con l'esclusione di questa tipologia di strutture, in generale non possiamo fare a meno dei lock

Strategia 2

Evitare hold-and-wait

- Un’altra possibilità è quella di evitare che un thread possa rimanere ad attendere che una risorsa si liberi mentre possiede dei lock

- Supponiamo che un thread possa aver necessità dei lock $L_1, L_2, \ldots, L_k$ durante la sua esecuzione

- Se li acquisisce tutti nello stesso momento (anche *prima* che gli servano) non può mai ritrovarsi ad attendere che un lock sia rilasciato

- **Problema**: dobbiamo prendere anche per lungo tempo dei lock che, in realtà, proteggono risorse che usiamo solo per brevi intervalli.

Strategia 3

Consentire la preemption

- Un’altra possibilità è permette ai lock di essere rilasciati.
- Esiste la funzione `pthread_mutex_trylock`, che ritorna non zero se non è stato possibile acquisire il lock.
- In quel caso possiamo rilasciare tutti i lock che il thread aveva e ricominciare da capo.
- Questo permette di “cedere” i lock quando non è possibile proseguire.
- Queste genera un nuovo problema: il **livelock**, in cui i thread fanno qualcosa ma non riescono mai ad acquisire tutti i lock che gli servono.

Strategia 4

Evitare dipendenze circolari

- In assenza di dipendenze circolari non è possibile ottenere deadlock.
- Una possibile strategia è quindi quella di imporre un ordine sui lock (e.g., $L_1 < L_2 < \ldots < L_k$) e se ci servono più lock (e.g., L_7, L_2, L_3 nell'ordine in cui ci servono) dobbiamo necessariamente acquisirli nell'ordine imposto:
  - E.g., prima acquisiamo L_2, poi L_3 e infine L_7, anche se la prima risorsa che ci serve è quella protetta da L_7.
- Per esempio il codice che si occupa del memory mapping in Linux ha ben documentato l'ordine in cui acquisire i diversi lock

Strategia 5

Evitare i deadlock tramite lo scheduler

- Una possibilità è quella di non scrivere il nostro codice in modo che i deadlock non possano verificarsi…
- …ma di avere uno scheduler che evita il loro verificarsi
- Per fare questo lo scheduler deve sapere che lock deve acquisire un thread
- Potremmo però essere troppo conservativi e limitare fortemente le performance per evitare scenari poco probabili di deadlock

Strategia 6

Rilevare e correggere le condizioni di deadlock

- Una ultima possibilità è quella di ispezionare mentre un programma sta eseguendo quale sia lo stato dei lock (chi li possiede) e dei thread (cosa stanno attendendo)

- Nel caso sia rilevata una dipendenza circolare è possibile svolgere alcune operazioni per “rimettere in moto” il sistema

- Una di queste possibilità è quella di far ripartire l’intera computazione (sperando che la situazione di deadlock non si ripeta)

- Un approccio molto pragmatico e utile in casi particolari (alcuni database)

Esercizio sui thread

Thread, parallelismo

E performance

- Proviamo a sommare 10^8 volte il valore 1 (versione semplificata del problema “vogliamo sommare tutti i valori di un array di 10^8 elementi)

- Possiamo farlo in un singolo thread oppure dividere il lavoro in n thread e ognuno di loro effettua solo $10^8/n$ somme

- In un sistema in cui i thread possono essere eseguiti in parallelo possiamo velocizzare con questo metodo?

- In un sistema con due core ci aspettiamo quasi un dimezzamento del tempo di esecuzione.

Legge di Amdahl

Quale è il massimo speedup grazie al parallelismo?

- Sia F la frazione del calcolo che è parallelizzabile
- Di conseguenza $(1 - F)$ è la frazione del calcolo che è parallelizzabile
- Supponiamo di avere a disposizione N processori
- Lo **speedup**, definito come $\frac{\text{tempo sequenziale}}{\text{tempo parallelo con } N \text{ processori}}$ è dato da $\frac{1}{(1 - F) + \frac{F}{N}}$

Legge di Amdahl

Quale è il massimo speedup grazie al parallelismo?

- Se lo speedup con N processori è $\frac{1}{(1-F)+\frac{F}{N}}$, per $N \to +\infty$ abbiamo che lo speedup è $\frac{1}{1-F}$, ovvero dipende dalla frazione del codice eseguibile solo in modo sequenziale

- $50\,\%$ del codice eseguibile in modo parallelo implica uno speedup massimo di $1/(1-0.5) = 2$

- $99\,\%$ del codice eseguibile in modo parallelo implica uno speedup massimo di $1/(1-0.99) = 100$