Capitolo 8 : Deadlocks

Table of contents

I deadlock sono un fenomeno abbastanza generale, applicabile anche alla vita reale:

Pensiamo a due persone che hanno bisogno di coltello e forchetta per mangiare; uno ha il coltello e l'altro ha la forchetta, nessuno presta all'altro il suo utensile, e quindi entrambi restano bloccati senza poter mangiare.

Il modello

Abbiamo il generico sistema, al cui interno sono poste delle risorse: Spazio in memoria, I/O devices, ecc. Le risorse sono indicate con R1, R2, ... , Rn.

Il discorso è il seguente: dobbiamo **disciplinare** in qualche modo l'accesso alle risorse, per evitare fenomeni in cui nessuno può andare avanti, ed i processi restano **bloccati** per sempre.

L'idea è quella di utilizzare un protocollo che:

- Faccia richiesta della risorsa.
- Utilizzi la risorsa.
- Rilasci la risorsa non appena ha finito.

Caratterizzazione di Deadlock

Quando è presente un **deadlock**, ci troviamo in un caso in cui queste quattro condizioni sono vere:

- **Mutua esclusione** : un solo processo alla volta può utilizzare una risorsa. Infatti se una risorsa fosse condivisibile, il problema non si porrebbe.
- **Hold and wait** : ogni processo ha almeno una risorsa, e ne vuole acquisire un'altra addizionale. Nell'esempio della forchetta, se una persona avesse entrambe le posate, ed un'altra nemmeno una, il problema non si pone.
  Nei casi di deadlock ognuno ha una risorsa, ma non vuole rinunciarvi.
- **No Preemption (prelazione)** : una risorsa può essere rilasciata solo

volontariamente dal processo che la trattiene, dopo che il dato processo ha completato il suo task.

- **Attesa circolare** : Il caso più semplice di deadlock è quello tra due processi, dove entrambi i processi vogliono una risorsa del corrispettivo, ma nessuno dei due è disposto a rilasciarla. Questa teoria può essere estesa ad N processi, come nel caso dei filosofi.

I Deadlock sono situazioni in cui dei processi restano bloccati **per sempre**, ovvero questi due processi non verranno sbloccati mai dalla situazione corrente; se invece un processo resta bloccato per un dato tempo, anche lunghissimo, non siamo in un caso di **deadlock**.

Grafi dell'allocazione delle risorse

E' evidente che il metodo più semplice per esaminare le dinamiche dei deadlock, è quello di costruire dei **grafi.**

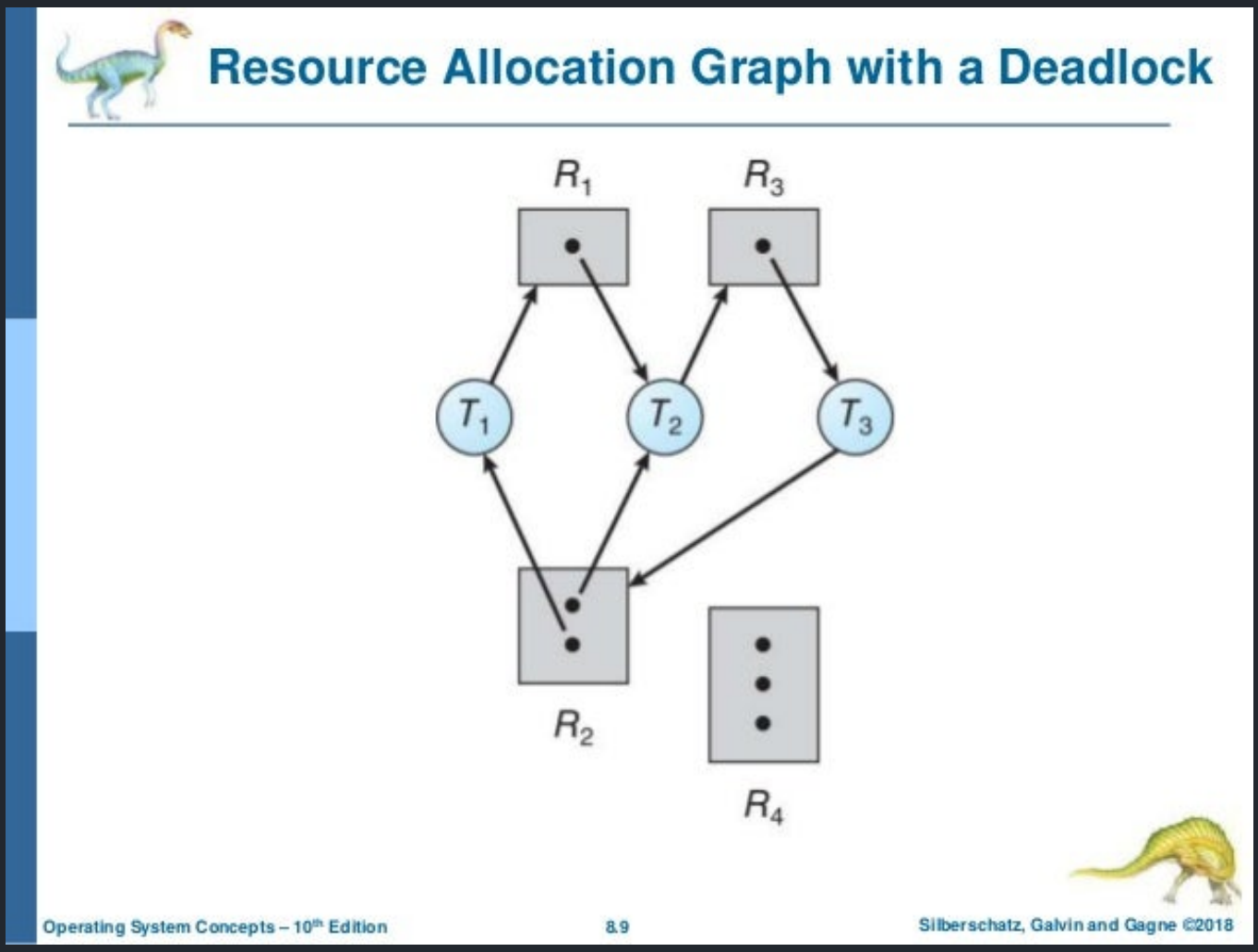
`Reminder:` un grafo è un insieme di Vertici V e di archi E, e possono essere di tipo orientato o non orientato.

L'idea è quella di avere due tipi di vertici:

- Vertici che corrispondono ai processi
- Vertici che corrispondono alle risorse

Anche gli archi sono di due tipi:

- **Processo** ➡️ **Risorsa** se richiedo la risorsa
- **Risorsa** ➡️ **Processo** se la risorsa è stata assegnata ad un processo.



I processi (T1,T2,T3) sono rappresentati da Cerchi.

Le risorse (R1,R2,R3,R4) sono rappresentate con i rettangoli.

I "pallini" all'interno delle risorse rappresentano le istanze in cui le risorse sono disponibili; ad esempio R4 abbiamo 3 istanze.

- La risorsa R1 (una sola istanza) è stata assegnata al processo T2.
- T2 ha anche il possesso di una risorsa R2 che gli è stata assegnata.
- T1 ha una risorsa R2.
- T2, non solo ha le risorse R1 ed R2, ma vorrebbe anche la risorsa R3: T2 ➡️ R3, ma R3 al momento è assegnata a T3.

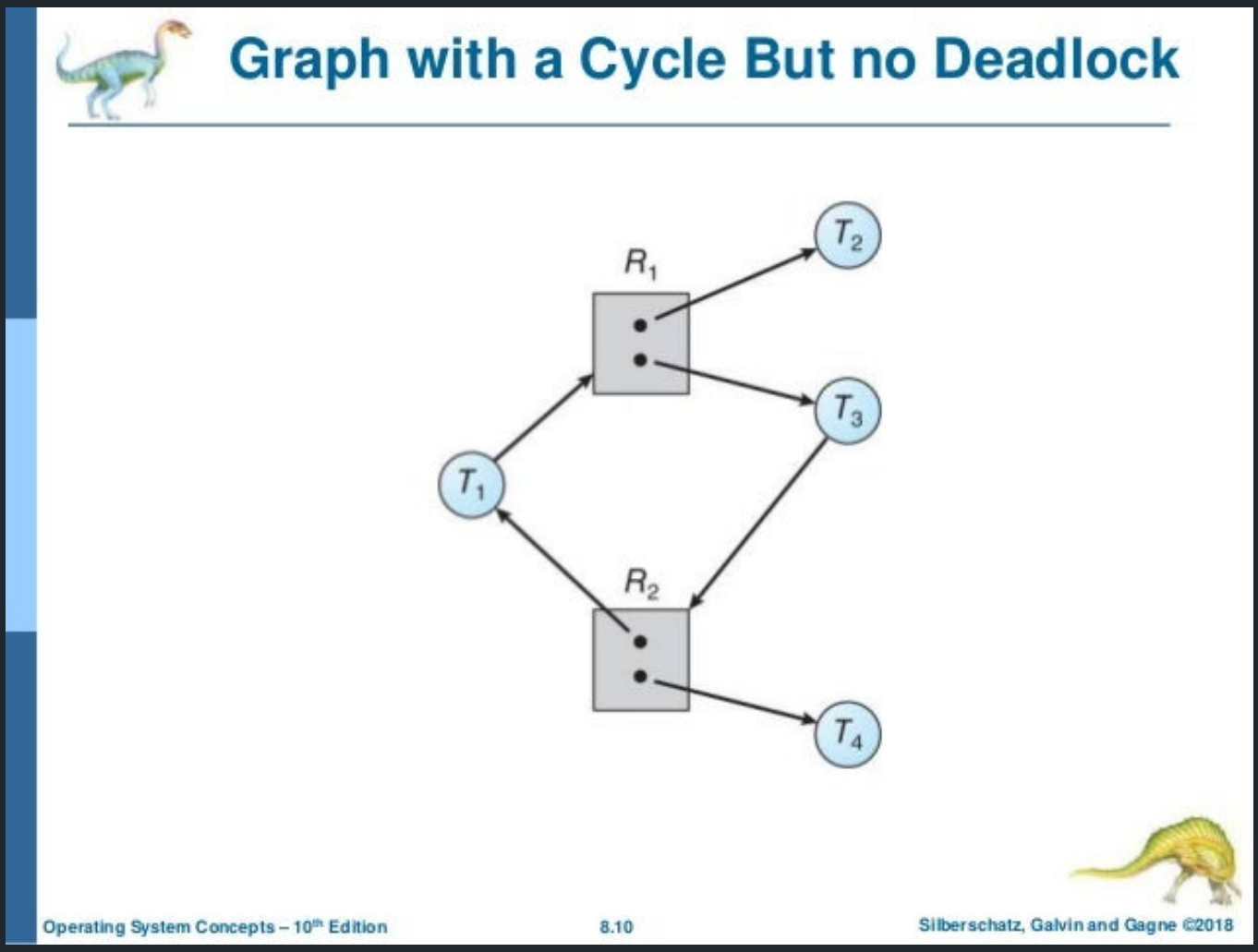
Come faccio a capire se in questo grafo è presente un deadlock?

Se riesco a trovare un percorso chiuso (ciclo), in qualche modo c'è una catena di richieste infinite, che vanno a bloccare il tutto.

In questo caso non è presente nessun ciclo, ed infatti non è un caso di Deadlock.

Potrebbe sembrare un caso di deadlock, ma questa situazione, prima o poi, verrà risolta con T3 che finisce di utilizzare R3, che viene quindi assegnata a T2.

Un ciclo è sempre un deadlock?



Se le risorse sono multiple no.

In questo caso, T2 e T4 non hanno richieste "inesaudite", per cui prima o poi rilasceranno R1 ed R2 che potranno così essere assegnate a T1 e T2.

Quindi la presenza di un ciclo è una condizione **necessaria, ma non sufficiente.**

Linee Generali per la ricerca di un deadlock

Se un grafo contiene un ciclo:

- Se è presente una sola istanza per tipo di risorsa, allora **abbiamo un deadlock.**
- Se abbiamo diverse istanze per tipo di risorsa, abbiamo la **possibilità di un deadlock** .

Deadlocks

Come possono essere gestiti i deadlock?

Possiamo gestire i deadlock in tre modi diversi:

1. Assicurarsi che il sistema non vada mai in uno stato di deadlock:

- **Prevenendoli**
- **Evitandoli**

2. Permettiamo al sistema di entrare in uno stato di deadlock e poi lo recuperiamo
3. Ignoriamo il Problema e facciamo finta che i deadlock non avverranno mai all'interno del sistema.

Più precisamente:

1. Evito fin dall'inizio che il sistema entri in uno stato di deadlock usando dei metodi **preventivi** .
2. Consento al sistema di entrare in uno stato di deadlock, non faccio quindi prevenzione, però ogni tanto controllo se ci sono stati di deadlock, se mi accorgo che c'è un deadlock, effettuo una **recover** , ovvero effettuare la **kill** di un processo e far si che la situazione venga sbloccata.
3. Possiamo ignorare completamente il problema, nella speranza che esso non si verifichi.

Quale soluzione scegliere?

Dipende dalla frequenza del fenomeno: se si tratta di due utenti che stanno tentando di avere la stessa risorsa I/O (ad esempio un masterizzatore DVD), è una situazione che avverrà di rado, e quindi possiamo scegliere la **terza soluzione**.

Se il problema è molto frequente, quindi la presenza di **deadlock** è più frequente, è meglio utilizzare la **seconda soluzione.**

Prima soluzione - Prevenzione

In qualche modo prendo delle opportune precauzioni in modo tale che il sistema non si porti mai in una condizione di deadlock.

Se torniamo sulla caratterizzazione dei deadlock vista nella lezione nove, vediamo che **tutte** le condizioni devono essere soddisfatte affinché si verifichi un deadlock. Se faccio in modo che almeno una di quelle condizioni non si verifichi **mai**, significa che non ci sarà mai un deadlock.

Invalidiamo quindi una delle quattro condizioni necessarie per la presenza di un deadlock:

- **Mutua esclusione:** non possiamo fare nulla, se le risorse fossero ad accesso *mutuo*, ovvero condivisibili, il problema non si porrebbe; solitamente le risorse coinvolte in un deadlock sono risorse su cui non possiamo operare contemporaneamente (non condivisibili).

- **Hold and wait:** in questo caso è possibile fare qualcosa. Visto che i processi Coinvolti in un deadlock hanno delle risorse, e tentano di acquisirne altre, come posso agire?
  Nel momento in cui un processo si avvia, il sistema gli fornisce **tutte** le risorse di cui ha bisogno, in modo che il processo non dovrà mai richiedere ulteriori risorse.
  Un altro modo di risolvere la cosa è quella di far partire il processo **senza alcuna risorsa**, e permetto al processo di richiedere delle risorse.

  Il problema di questa soluzione è che **tutte le risorse** devono essere **allocate in una sola volta**. Inoltre le risorse vengono utilizzate male, visto che vengono allocate con molto tempo di anticipo, quando potrebbero essere utilizzate da altri processi.

  Inoltre è possibile si verifichi il fenomeno della **starvation**, visto che la maggior parte delle risorse potrebbero essere utilizzate da *pochi processi*.

- **No Preemption**: Un deadlock si verifica nel momento in cui **non è possibile sottrarre le risorse**. Se riesco a realizzare un meccanismo in cui faccio rilasciare ad un processo **tutte le sue risorse**, per poi riacquistarle tutte in una volta in un secondo momento, potrebbe funzionare.

  Tuttavia non è un sistema particolarmente brillante.

- **Attesa circolare:** questo è il sistema più *brillante*, ovvero il sistema che tenta di evitare l'attesa circolare.

Se riuscissi a dare un **numero d'ordine** alle risorse, ovvero prevedendo una sequenza ragionevole dell'utilizzo delle risorse, posso dire al processo che può richiedere delle risorse, ma solo **per un numero d'ordine maggiore rispetto a quello attuale**.

Se il processo ha risorse di tipo 2, può richiedere risorse di tipo 3/4/... ma non quelle di ordine inferiore (ad esempio 1).

Con questo sistema non si può mai chiudere un anello: Pensiamo ad una catena circolare in cui ogni elemento chiede qualcosa al prossimo elemento; il primo chiede una risorsa maggiore rispetto a quella che ha, così come il secondo, il terzo ecc.

Capiamo quindi che le risorse che vengono richieste sono **in crescita**, a questo punto la catena (circolare) non può essere chiusa, siccome l'ultimo deve chiedere al primo una risorsa **minore** rispetto a quella cha ha già, il che è impossibile.

Il fatto che le richeste siano via via di ordine maggiore, **impedisce la chiusura dell'anello**.

Questa può essere una soluzione ragionevole.

Evitamento di Deadlock (avoidance)

Questi metodi sono diversi dai precedenti, perchè invece di evitare di ottenere una delle quattro condizioni che portano ad un deadlock, utilizzano ulteriori informazioni sui cosa i processi "vogliono fare".

Questo vuol dire che i processi fanno utilizzo di una conoscenza di quello che i processi chiederanno nel futuro. Questa soluzione **non è sempre utilizzabile**, visto che è utile solo nel momento in cui so esattamente cosa un processo richiederà (risorse, nel caso peggiore).

Il fatto che il processo **dichiari a priori** le risorse che utilizzerà è molto limitante nei confronti del modello;

🏁 05-20 0:32

Quindi, il modello è che il processo dichiara fin dall'inizio il numero massimo di risorse che utilizzerà. L'idea è esaminare lo stato del sistema che si otterrebbe se il processo **ottenesse le risorse chieste**; visto che sono a conoscenza delle risorse che i processi richiederanno, prima che il sistema le conceda effettivamente, si controlla prima se, dando le risorse al processo, si potrebbe entrare in una situazione di deadlock.

Se concedendo le risorse al processo si entra in una situazione di deadlock, il processo deve attendere un momento migliore per ricevere le risorse richieste, altrimenti le risorse vengono concesse.

Safe state - stato sicuro del sistema

Il sistema è in un **safe state** se esiste una sequenza <P1, P2, ... , Pn> di **tutti** i processi nel sistema, in modo che ogni Pi, le risorse che Pi può ancora richiedere possono essere soddisfatte dalle risorse disponibili correntemente + le risorse mantenute da tutti i Pj con j<I

In altre parole...

Ho un certo numero di processi, i quali effettueranno ulteriori richieste (per risorse); questi processi hanno dichiarato inizialmente il numero massimo di risorse di cui avranno bisogno; a questi processi sono già state concesse delle risorse (non tutte); durante il resto della loro vita effettueranno altre richieste.

Lo scopo del sistema è quello di portarli a terminazione tutti, e far sì che questi non si "blocchino" (deadlock) per via della mancanza di risorse disponibili.

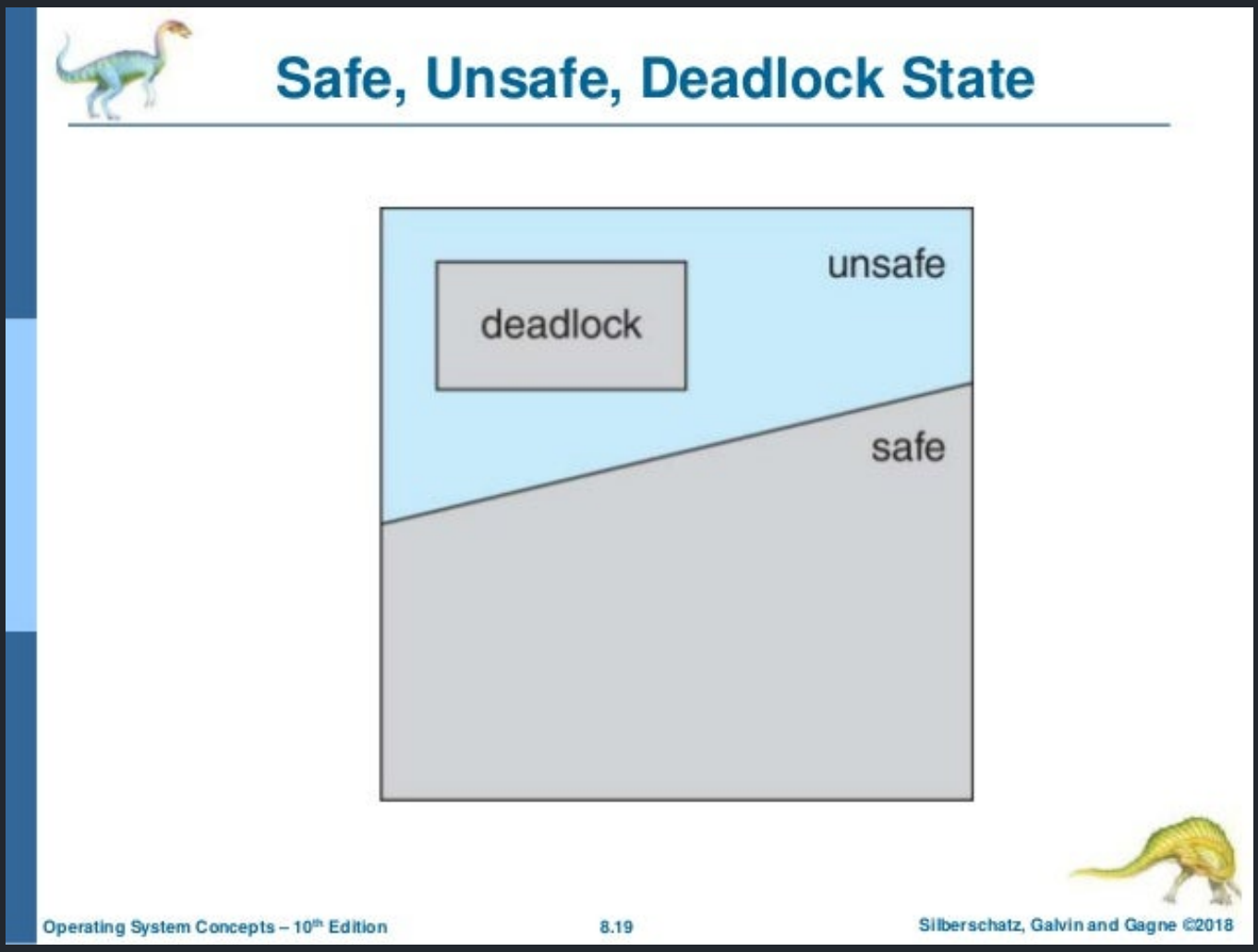
Uno stato sicuro è quindi uno stato in cui, con le risorse disponibili, i processi (nel peggiore dei casi) riescono a terminare, anche uno alla volta.
Questo vuol dire che una volta terminato un processo, questo restituisce tutte le risorse, e con queste risorse più quelle già disponibili, riesco a far terminare un altro processo, e così via.

Fatti basilari

- Se il sistema è in uno **stato sicuro** ➡️ nessun deadlock.
- Se il sistema è in uno stato **non sicuro** ➡️ possibilità di avere un deadlock. **(a)**
- Avoidance ➡️ assicurarsi che il sistema non entri mai in uno stato **non safe** .

(a) in questo caso **non abbiamo la certezza** di un deadlock perchè le richieste dichiarate dal processo **sono quelle del caso peggiore**, ed il processo potrebbe **non richiedere TUTTE** le risorse dichiarate durante la sua vita.
Inoltre, il "fattore fortuna" può avere una parte nell'evitare i deadlock, infatti le richieste si "incastrano" tra loro in modo tale da non creare un deadlock.

Stati sicuri, insicuri e di Deadlock



- In uno stato Safe il deadlock non può verificarsi.
- In uno stato Unsafe non è detto che ci sia un deadlock, ma potrebbe verificarsi.

Soluzine 1 : Algoritmi di Avoidance

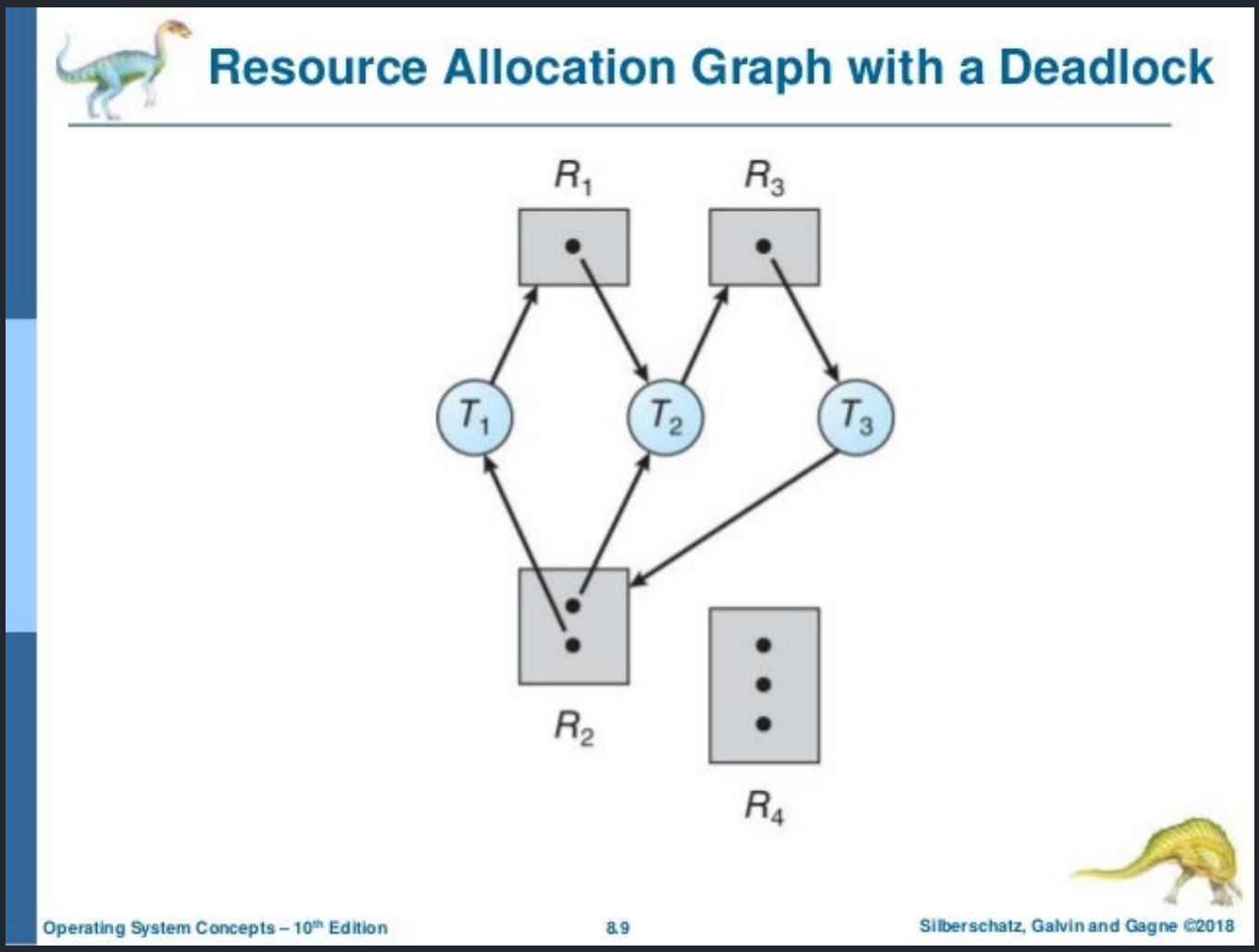
Come si gestisce la transizione tra stati che siano sempre **safe**?

I metodi sono diversi a seconda che le risorse siano a singola istanza, o siano disponibili in più istanze:

- Singola istanza per tipo di risorsa
  - Usare un grafo di allocazione di risorse
- Multiple istanze per tipo di risorsa
  - Usare l'algoritmo di **Banker**

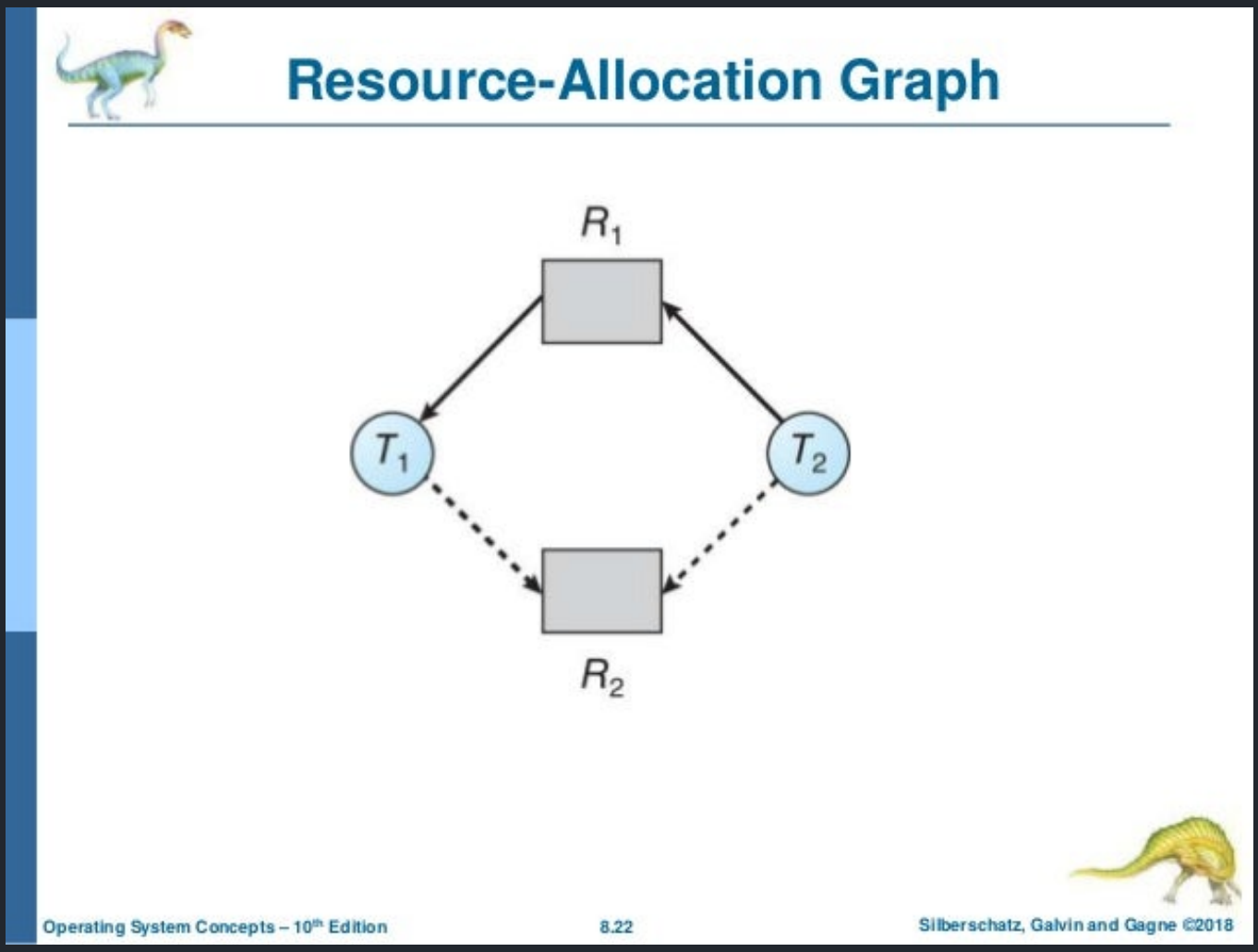
Singola istanza

Cosa cambia rispetto al grafo delle risorse visto precedentemente?



In questa figura, non sono visualizzate le **richieste future** di un processo; quindi dobbiamo rappresentare quelle che sono le possibili richieste di un processo.

Questi nuovi archi, chiamati **claim edge**, sono rappresentati da linee tratteggiate:



Anche in questo caso l'arco va dal processo alla risorsa, così come accadeva per le richieste: Pi ➡️ Rj.

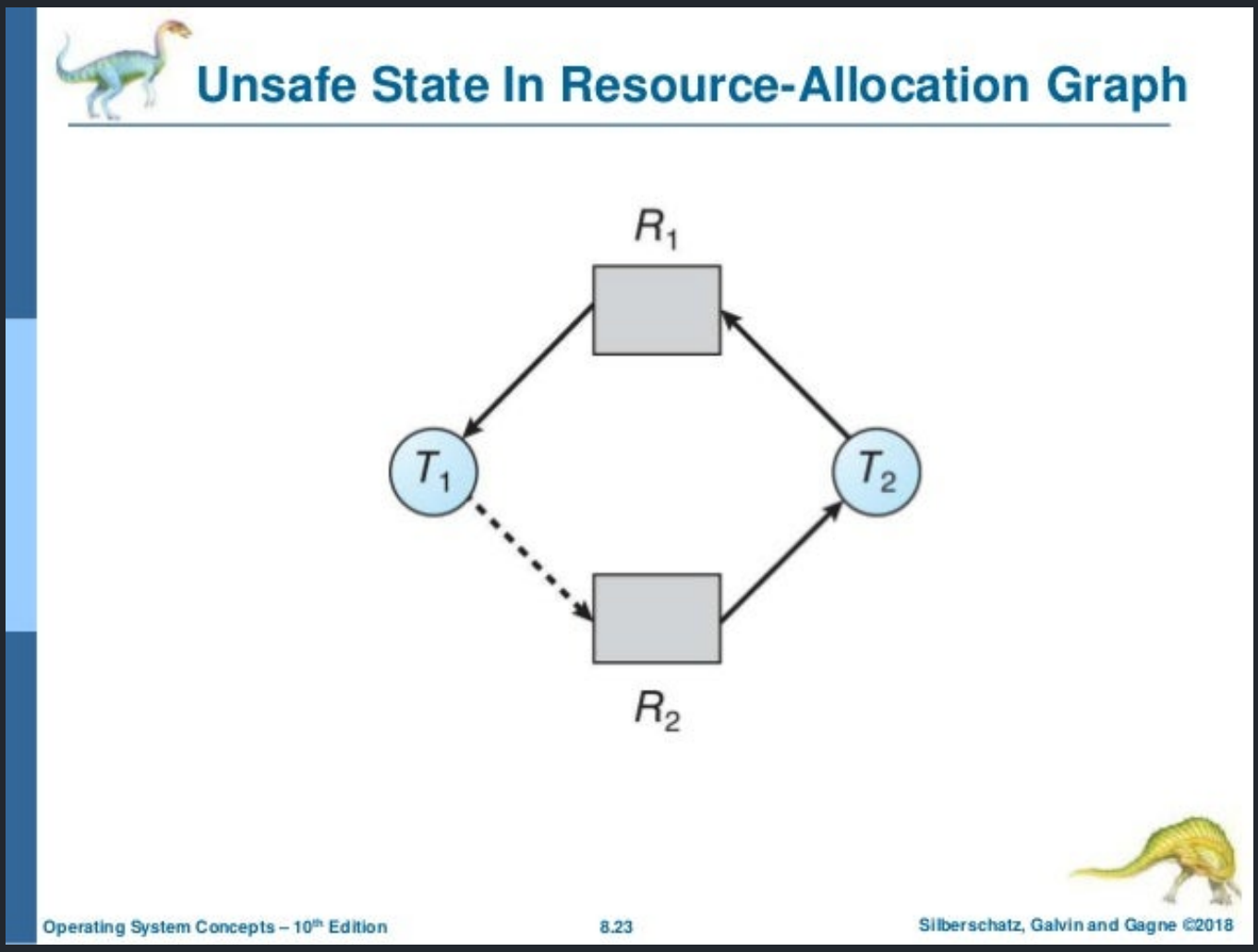
Nel momento in cui il processo effettua la richiesta, l'arco claim viene convertito in un arco **request**, e quindi in pratica da tratteggiato diventa solido.

Quando la risorsa viene effettivamente concessa al processo, il verso dell'arco viene invertito.

Ritorniamo all'immagine:
Poniamo il caso che T2 dichiari **inizialmente** di volere la risorsa R2, e poi successivamente la richiede effettivamente. Il sistema può concedergliela?

Si va a vedere quindi la situazione in cui il sistema si troverebbe se la risorsa venisse concessa. Ci troviamo quindi in un **ipotetico stato**; lo stato è sicuro?

Per vedere se lo stato ipotetico è sicuro, basta vedere se nel grafo esiste un **ciclo che coinvolge anche gli archi tratteggiati.**



Come si nota nell'immagine, esiste un ciclo che comprende anche gli archi tratteggiati; questo vuol dire che se ad un certo punto T1 richiede la risorsa R2, ci troveremmo in uno stato di deadlock!

Morale della favola: la risorsa R2 non può essere concessa a T2.

🏁 05-20 0:49

Algoritmo di Banker - risorse disponibili in istanze multiple

L'algoritmo del banchiere, serve a capire, nel caso in cui ci siano risorse disponibili in istanze multiple, se il sistema si porta in uno stato safe o meno.

Il concetto è sempre lo stesso: nel momento in cui un processo chiede delle risorse, si va a vedere lo stato in cui il sistema si porterebbe nel caso concedesse la risorsa; se lo stato è sicuro può concederle, altrimenti il processo attende.

Perchè si chiama algoritmo del banchiere?

C'è una analogia con un banchiere avente una cassa colma di soldi e deve distribuirli a delle persone che devono realizzare un qualcosa;

Nel momento in cui queste persone completano il loro progetto, restituiranno tutto il denaro prestato.

Il banchiere deve evitare di portarsi in una di queste situazioni:

- Non ha più soldi in cassa
- Nessuno può terminare il proprio progetto perchè ha bisogno di denaro addizionale (ed io non posso darglielo)

Questo vuol dire che, seguendo l'analogia, il banchiere concede dei soldi solo se si porta in uno stato in riesce, almeno **uno alla volta**, a far terminare il progetto dei richiedenti, in modo da avere indietro il denaro (da poter prestare ad altri).

Cosa serve per far funzinoare l'algoritmo?

- Come tutti gli algoritmi di **avoidance** ogni processo deve dichiarare a priori le risorse (massime) che utilizzerà.
- Nel momento in cui viene richiesta una (o più) risorsa, vado a vedere lo stato in cui il sistema si porterebbe
- Se il sistema ritiene che lo stato è safe, concede le risorse.

Ogni processo nel momento in cui ha avuto tutte le risorse richieste, prima o poi dovrà restituirle, in un tempo **finito**.

Algoritmo Safety - parole povere (non matematiche)

Esempio: abbiamo 5 processi chiamati P0, ... ,P4; abbiamo inoltre 3 tipi di risorse con diverse istanze:

- A (10 istanze)
- B (5 istanze)
- C (7 istanze)

Possiamo costruirci una tabella:

	Allocazione	**Max**	**Disponibile**
	A B C	A B C	A B C
P0	0 1 0	7 5 3	3 3 2
P1	2 0 0	3 2 2	
P2	3 0 2	3 2 2	
P3	2 1 1	2 2 2	
P4	0 0 2	4 3 3	

Ad esempio, P0 ha avuto 0 risorse di tipo A, 1 di tipo B e 0 di tipo C; P0 **potenzialmente**, potrebbe richiedere 7 risorse di tipo A, 5 di tipo B e 3 di tipo C.

L'ultima colonna, quella denominata **Available**, denota le risorse che il banchiere ha in cassa, ovvero quelle disponibili.

Se effettuiamo la differenza membro a membro tra le risorse **allocate** e quelle **massime** che il processo può richiedere, otteniamo la colonna **Need**, ovvero le risorse che il processo può ancora chiedere:

Processo	Need
	A B C
P0	7 4 3
P1	1 2 2
P2	6 0 0
P3	0 1 1
P4	4 3 1

> Possibili richieste future

In questo istante, questo è uno **stato safe**?

Questo stato è **safe** se il sistema riesce a far terminare tutti i processi in un modo o nell'altro, anche uno alla volta.

In questo momento il sistema ha disponibile `3 3 2`, il che è sufficiente per soddisfare le richieste di P1 e farlo terminare; il sistema "punta tutto" su P1. Quando P1 termina, restituisce non solo `3 3 2`, ma anche `2 0 0`, ottengo quindi `5 3 2`, che posso utilizzare per "accontentare" un altro processo.

Il sistema procede quindi con l'esecuzione di P3, che richiede `0 1 1`, che restituisce anche `2 1 1`, portando le risorse disponibili a `7 4 3`.

Con `7 4 3` il sistema esegue P4, che richiede `4 3 1`, che restituisce anche `0 0 2`, portando le risorse disponibili a `7 4 5`.

Il sistema procede con P2, ed infine con P0. **Capiamo quindi che questo è sicuramente uno stato safe.**

Quindi

L'algoritmo funziona in questo modo: prima di concedere delle risorse ad un processo, il sistema controlla che lo stato sia safe, ovvero vede se può terminare **tutti i processi**, anche uno per volta, con le risorse disponibili.

Richiesta non safe

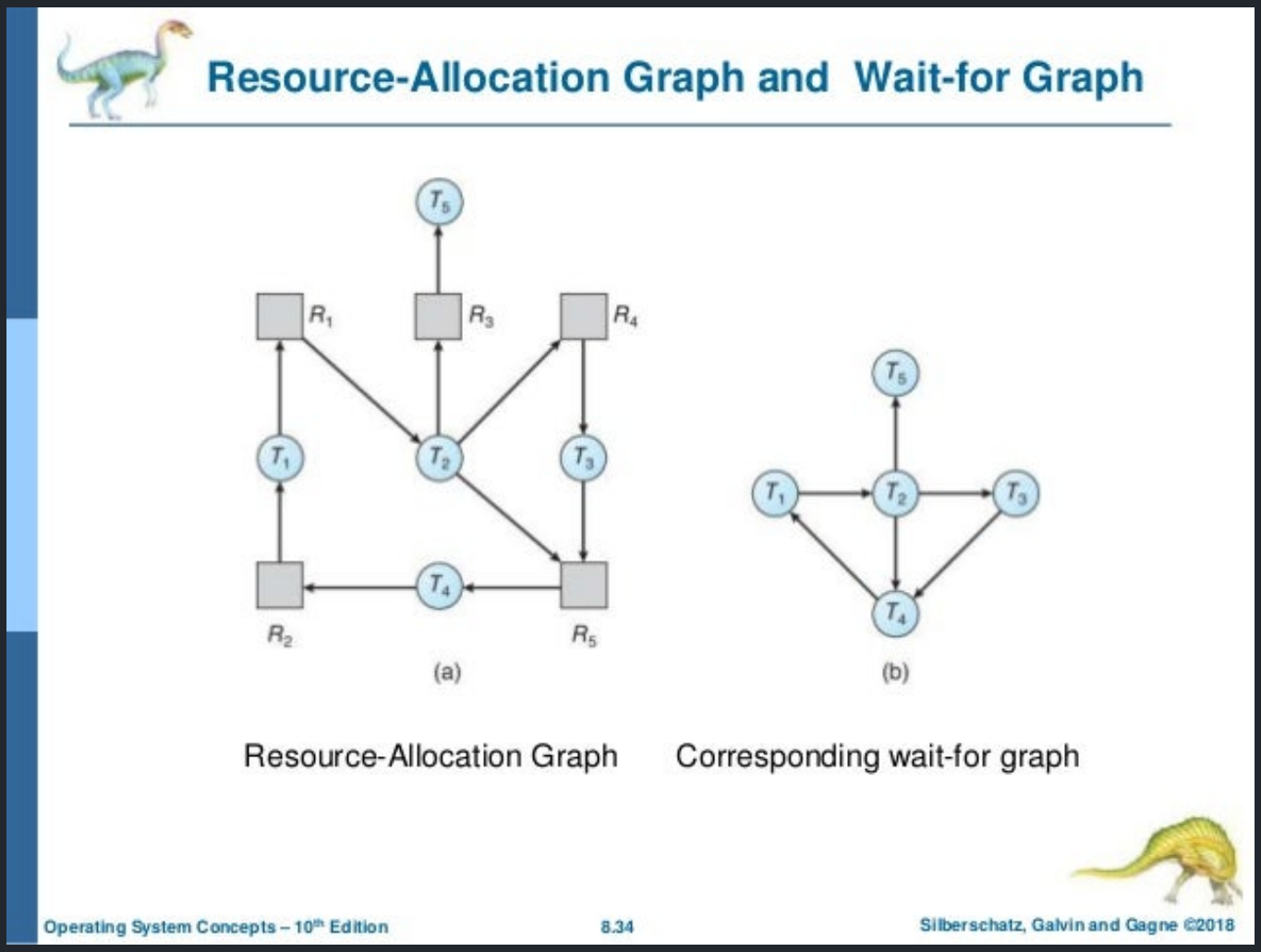
Se arriva una richiesta di `3 3 0` da parte di P4, che in **Need** ha `4 3 1`, prosciuga completamente le risorse disponibili su A e B, ed il sistema non riesce più a far terminare **tutti** i processi.

Soluzione 2 : Detection

Il sistema gestisce le risorse in libertà senza alcuna restrizione, dove i processi potrebbero portarsi in uno stato di deadlock; ogni tanto però il sistema controlla, se è presente un deadlock trova una strategia di ripristino che porta il sistema ad uno stato stabile.

Caso più semplice: unica istanza per risorsa

In questo caso basta usare un grafo (semplificato rispetto al caso 1); infatti basta rappresentare le richieste nel grafo **wait for**:



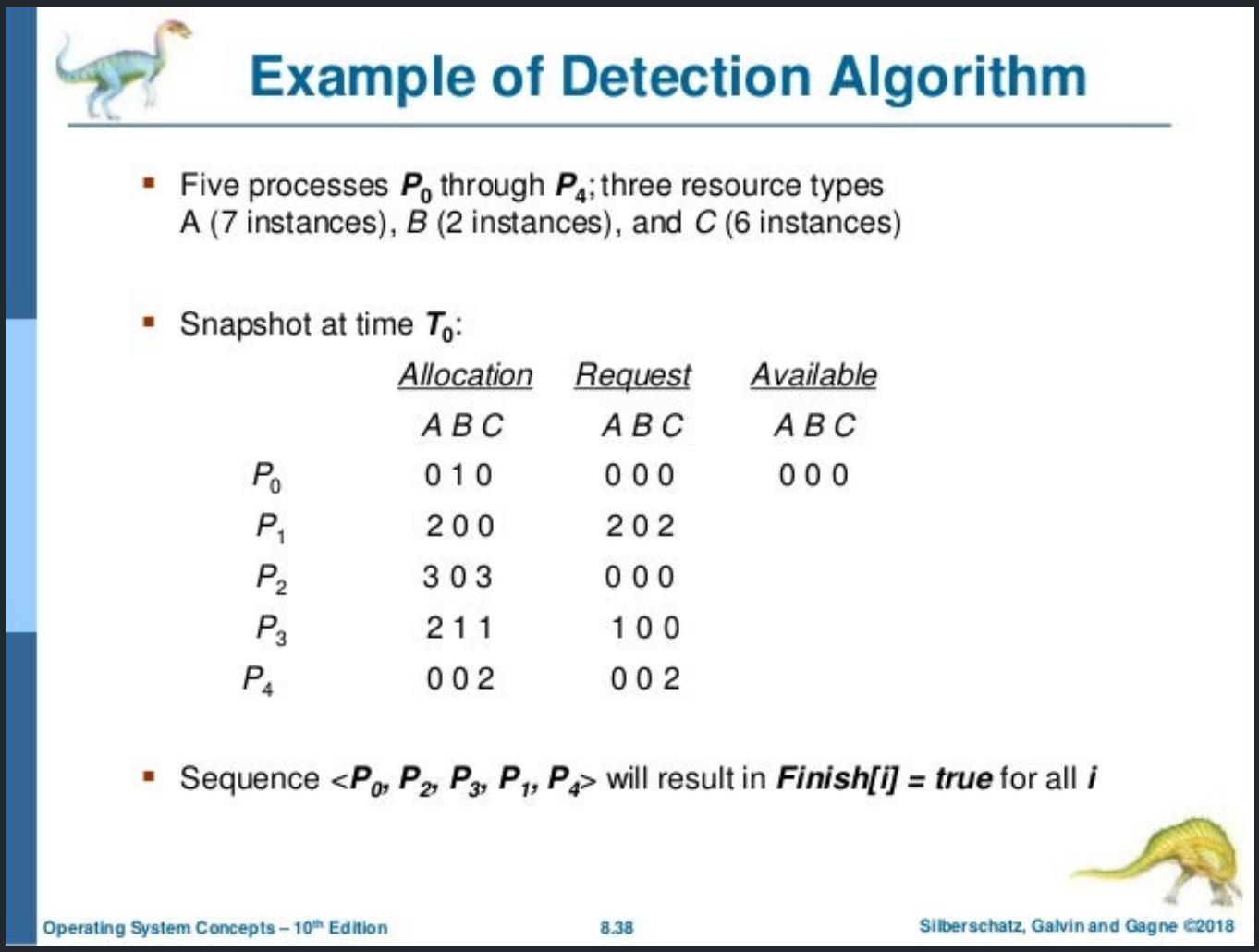
Sono presenti i processi, e tra i processi sono posti degli archi orientati. Se abbiamo Pi -> Pj, Pi sta attendendo una risorsa che è in utilizzo da Pj.

Ad intervalli di tempo il sistema effettua uno snapshot della situazione, costruisce il grafo **wait-for**, se viene trovato un ciclo, siamo in presenza di un **deadlock**.

I problemi

Questo non è un task "semplice" da compiere, infatti la ricerca di un ciclo in un grafo, nel caso peggiore, è pari a n^2. Di conseguenza, ogni volta che il sistema effettua il controllo, spreca del tempo ed introduce una **notevole fonte di overhead**.

Inoltre, il sistema potrebbe effettuare uno snapshot, controllare e notare che non sono presenti deadlock. Il problema risiede nel fatto che il deadlock potrebbe verificarsi anche qualche istante dopo il controllo. Si tratta quindi di "ipotesi ottimistiche", e non scienza esatta.



Questo algoritmo è efficiente?

Questo algoritmo è ancora peggio del precedente (banchiere), infatti la sua complessità temporale è pari ad **$(m \times n^2)$**, dove m sono le risorse ed n sono le istanze.

Conviene usare questo algoritmo?

Dipende dalla possibile frequenza dei deadlock. Se i deadlock avvengono con una bassa frequenza, può convenire utilizzare questo tipo di algoritmo.

Come ripristinare uno stato di deadlock? - Rollback

Nasce il problema del **rollback**.

Potrei avere più cicli, e quindi "più deadlock"; devo quindi effettuare un kill di processo per ogni ciclo (nel grafo), o almeno riportarlo in uno stato in cui non quel dato processo non possiede più quelle date risorse.

Tecnica del checkpointing

Supponiamo di avere un programma di elaborazione scientifica che gira per ore / giorni; se dopo due giorni si verifica un problema, ad esempio l'alimentazione viene interrotta, il programma dovrebbe iniziare la sua elaborazione da zero.

Questo ovviamente non è ammissibile; servono quindi opportune strategie di checkpointing:

Nel mio programma sono arrivato ad un certo punto ottenendo certi valori, salvo questi valori in un file e pongo un flag che dice al programma che, nel caso dovesse ripartire, salta tutta l'elaborazione fino a quel punto, e riprende da lì.

Esistono **librerie di checkpointing**, che salvano tutto lo stato del programma, per cui ogni tanto viene invocata una funzione che salva lo stato del programma, in modo tale che quando il programma verrà rilanciato, esso ripartirà da quel dato checkpoint.

Possibili soluzioni per riprendersi da un deadlock

- **terminazione di TUTTI i processi coinvolti nel deadlock**
- Terminazione dei processi coinvolti nel deadlock **uno alla volta** finché il ciclo del deadlock è eliminato.
- In che ordine dovremmo scegliere il processo da terminare?
  - Priorità del processo: se ho un processo **real-time** non è di certo il processo da killare
  - Da quanto tempo il processo è in esecuzione: di certo non posso killare un processo che gira da 2 giorni, perchè probabilmente ha bisogno di molto altro tempo di CPU.
  - Quante risorse ha usato
  - Il processo è interattivo o batch?

Quindi

- **Selezionare una vittima** - minimizzare i costi.
- **Rollback** - ritornare ad uno stato sicuro, riavviare il processo per quello stato. Questo vuol dire portare il processo ad uno stato intermedio, se possibile è preferibile.
- **Starvation** - Questo processo scelto (da killare o per rollback) potrebbe essere soggetto a starvation, essendo sempre scelto.