Corso di Programmazione Concorrente

Algoritmi Non-Blocking

Valter Crescenzi

http://crescenzi.inf.uniroma3.it

Sommario

- Attese Attive vs Attese Passive
  - Costo della Sincronizzazione
  - Attese ibride
- Algoritmi *non-blocking*
  - **`java.util.concurrent.atomic`**
  - Strutture dati aggiornabili atomicamente
    - Contatore
    - Stack
  - Strutture dati con stato composito
    - Algorithmo di *Michael-Scott* su code
    - Il *Problema ABA*

Sincronizzazione ed Attese Attive

- Per f.d.e. sequenziali debolmente connessi (a “grana grossa”)
  - risultano convenienti le attese passive e lo *spinning* viene generalmente evitato

- Tuttavia meglio non perdere di vista che la primitiva *TestAndSet* fornisce un valido strumento di sincronizzazione che in altri contesti può risultare addirittura conveniente

Sincronizzazione *Non-Blocking*

- Un nuovo schema di implementazione delle due primitive basato direttamente sugli spin-lock

- Implementazione di un semaforo binario *non-bloccante*
  - P(S) con LOCK(SX)
  - V(S) con UNLOCK(SX)

  può comportare una attesa attiva <u>per tutta la durata della sezione critica di qualche altro f.d.e.</u> ma risparmia il costo del context-switching

- Ha senso solo per i multi-processor
- E la *fairness!*?

Costo della Sincronizzazione (1)

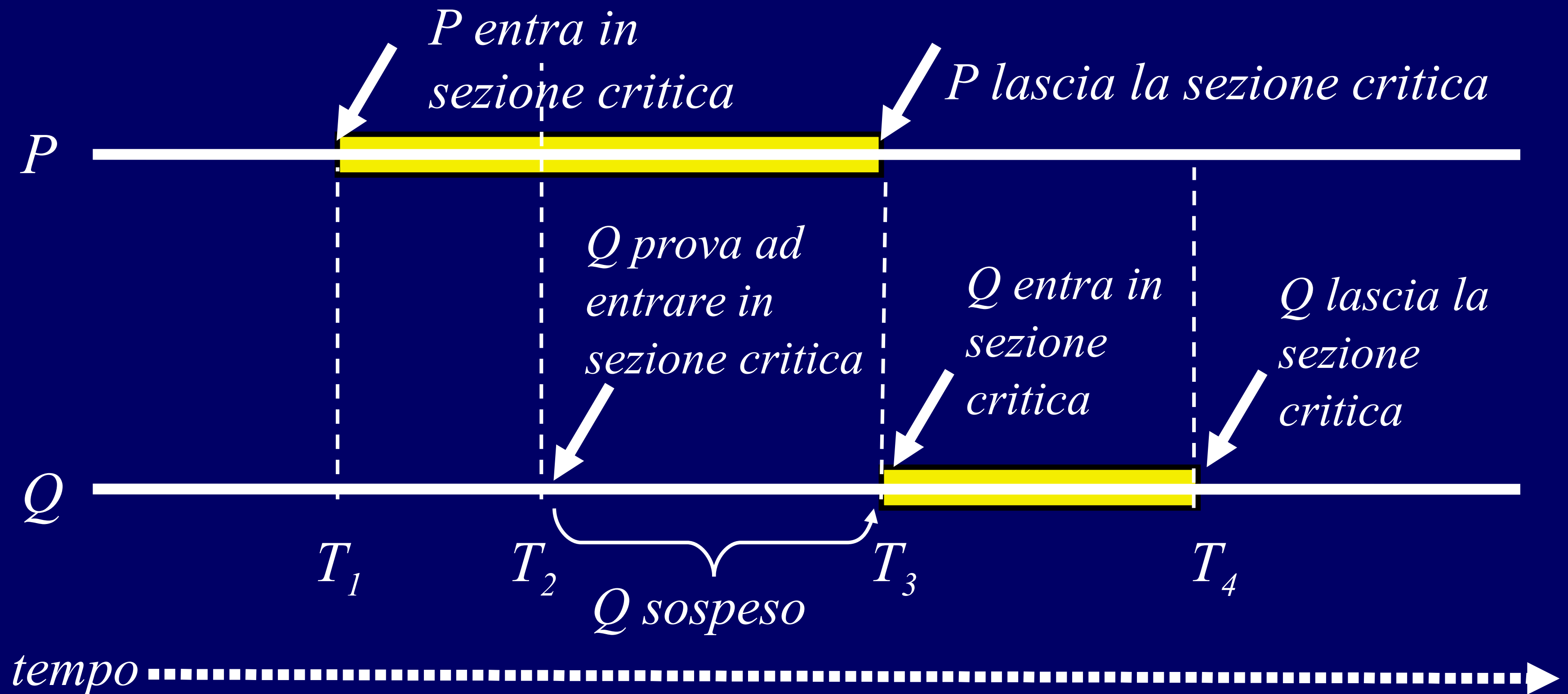
- *Overhead per singola esecuzione* di una sezione critica realizzata tramite primitive **P** e **V**

- Confrontiamo le due alternative
- *Blocking*: attesa passiva e context-switch
- *Non-blocking*: attesa attiva

  (nessun context-switch viene forzato)

- I costi più rilevanti risultano essere:
  - *Blocking*:
    - costo del context-switching
    - costi delle attese attive sullo spin-lock di basso livello trascurabili
  - *Non-blocking*:
    - costo delle attese attive (sullo spin-lock di alto livello)

Costo della Sincronizzazione (2)



- *Non-Blocking*: spinning per un int. di tempo T_3-T_2
- *Blocking*: 2 context switch
  (+spinning sul lock di basso livello trascurabile)

Costo della Sincronizzazione (3)

Il costo della sincronizzazione dipende da:

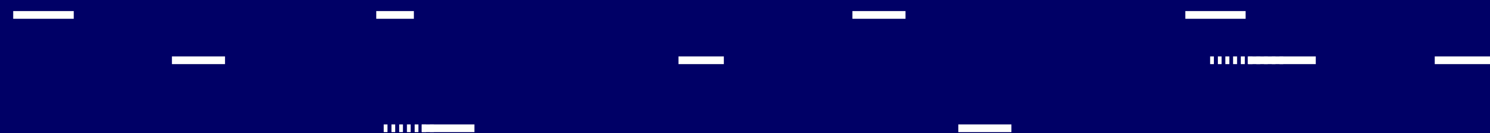
- livello di competizione
- durata delle sezioni critiche
- implementazione adottata (context-switch vs spinning)

➢ Livello di competizione elevato: sezioni critiche lunghe e frequenti

- aumenta il costo delle attese attive
- stabile il costo del context-switching

➢ Livello di competizione basso: sezioni critiche brevi ed infrequenti

- diminuisce il costo delle attese attive
- stabile il costo del context-switching



Costo della Sincronizzazione (4)

- Consideriamo l'andamento dei costi di sincronizzazione all'aumentare del livello di competizione e della durata delle sezioni critiche
  - *aumenta* per la sincronizzazione *non-blocking*
  - si può ipotizzare *costante* per la sincronizzazione *blocking*

- Per sole macchine multi-processor

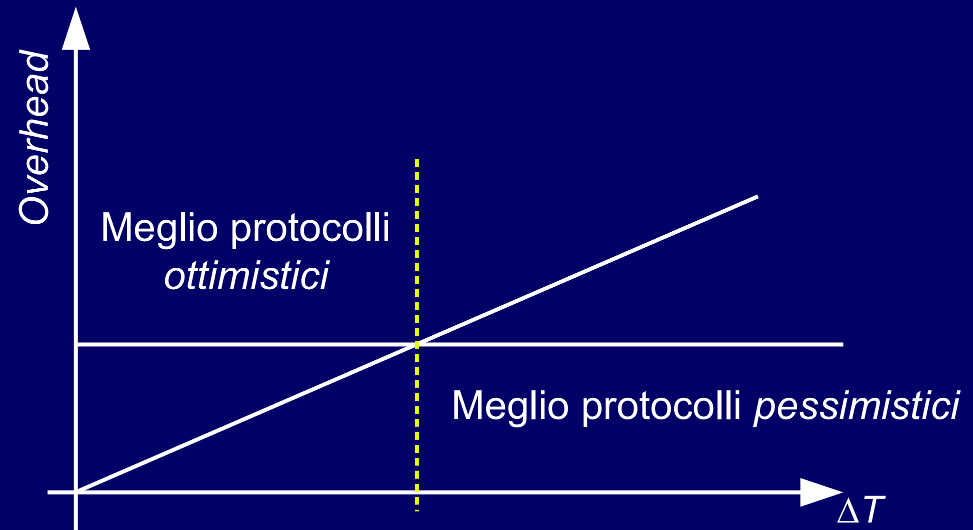
✔ *deve* esistere un certo *"livello di competizione"* in corrispondenza del quale l'overhead delle due tipologie di protocolli di sincronizzazione si equivalgono. Sotto questo livello

  - ✔ *risulta più efficiente una breve attesa attiva che due context-switch*

Attese Attive vs Attese Passive

- L'implementazione basata su attese passive può inquadrarsi come una gestione pessimistica della sincronizzazione
  - ✔ Più efficiente in presenza di forte competizione

- L'implementazione basata su attese attive può inquadrarsi come una gestione ottimistica della interferenza
  - ✔ Più efficiente in presenza di scarsa competizione



Approccio Ibrido

- Alcune piattaforme multi-processor possono decidere la migliore realizzazione della semantica di un semaforo sulla base di una analisi statistica della frequenza e della durata delle sezioni critiche a tempo di esecuzione
- Spesso, più semplicemente:
  - prima, spinning di breve durata
  - quindi, context-switch ed attesa passiva
- Altre possibilità:
  - *backoff* prima di ritentare lo spinning

Fine-grained adaptive biased locking

http://dl.acm.org/citation.cfm?id=2093157.2093184

Non-Blocking Synchronization e Algoritmi *Non-Blocking*

- Queste stesse osservazioni hanno stimolato lo studio di una nuova generazione di algoritmi concorrenti
  - Consentono di raggiungere prestazioni altrimenti impensabili
-

Algoritmi *non-blocking*:

- rinunciano all'uso di strumenti (come i semafori) che possano introdurre attese (passive)
- sincronizzazione *solo* tramite attese attive
- ammettono l'esistenza di stati *inconsistenti/transienti* (ovvero solo “parzialmente” costruiti)
- utilizzano istruzioni atomiche elementari (tipo *TestAndSet*) per tutti gli aggiornamenti di singole “componenti” dello stato
- ciascun f.d.e. *deve* saper lavorare in presenza di interferenza!

Algoritmi *Non-Blocking*: Vantaggi & Svantaggi

- Principali vantaggi:
  - efficienza e scalabilità
  - soffrono più difficilmente di stallo e starvation
- Principali svantaggi:
  - molto! difficili da progettare e testare
  - *starvation* (in senso stretto) comunque possibile
- In effetti il loro utilizzo è sufficientemente motivato in contesti piuttosto specifici:
  - porzioni del nucleo di un sistema operativo
  - librerie ad alte prestazioni (scalabili) di strutture dati thread-safe oggetto di numerosi accessi concorrenti

Java 5+: Supporto agli Algoritmi *Non-Blocking*

- **`java.util.concurrent.atomic`**
  - classi *wrapper* che offrono operazioni atomiche del tipo *TestAndSet* – qui *CompareAndSet* (CAS) –
  - ✔ realizzate grazie ad uno specifico supporto della *JVM*

  - **`java.util.concurrent.atomic.Atomic___`**
    - **`Boolean`**
    - **`Integer`**
    - **`Long`**
    - **`Reference`**
    - **`IntegerArray`**
    - **`LongArray`**
    - **`ReferenceArray`**
    - …

java.util.concurrent.atomic.AtomicInteger

- Un esempio per tutte le classi **Atomic__**

Constructor Summary

Constructors

Constructor and Description
`AtomicInteger()` Creates a new AtomicInteger with initial value `0`.
`AtomicInteger(int initialValue)` Creates a new AtomicInteger with the given initial value.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`int`	`accumulateAndGet(int x, IntBinaryOperator accumulatorFunction)` Atomically updates the current value with the results of applying the given function to the current and given values, returning the updated value.
`int`	`addAndGet(int delta)` Atomically adds the given value to the current value.
`boolean`	`compareAndSet(int expect, int update)` Atomically sets the value to the given updated value if the current value == the expected value.

Semantica Istruzioni *CAS*

- La semantica dei metodi di queste classi si può *documentare* con precisione direttamente nello stesso linguaggio java

- I metodi di **`SimulatedCAS>>`**
  - offrono la stessa semantica degli omonimi metodi nella classe **`AtomicInteger`**
  - l'implementazione e le prestazioni sono chiaramente molto diverse nei due casi

CompareAndSwap & CompareAndSet

```
public class SimulatedCAS {
    private int value;

    public synchronized int get() { return value; }

    public synchronized int compareAndSwap(int expectedValue,
                                           int newValue) {
        int oldValue = value;
        if (oldValue == expectedValue)
            value = newValue;
        return oldValue;
    }

    public synchronized boolean compareAndSet(int expectedValue,
                                              int newValue) {
        return (expectedValue
                == compareAndSwap(expectedValue, newValue));
    }
}
```

Proprietà degli Algoritmi Non-Blocking

- Con riferimento ad un sistema di f.d.e. che eseguono un algoritmo non-blocking:
  - *wait-free*: tutti i f.d.e. del sistema fanno progressi anche in presenza di competizione
  - *lock-free*: almeno un f.d.e. del sistema fa progressi anche in presenza di competizione
  - *obstruction-free*: almeno un f.d.e. del sistema fa progressi in *assenza* di competizione

✔ Proprietà via via più ambite e difficili da realizzare

Un Semplice Algoritmo *Lock-Free*

- Lo stato degli oggetti da aggiornare è formato da un unico campo da aggiornare atomicamente

```
public class CasCounter {
    private SimulatedCAS value;

    public int getValue() {
        return value.get();
    }

    public int increment() {
        int v;
        do {
            v = value.get();
        }
        while (v != value.compareAndSwap(v, v + 1));
        return v + 1;
    }
}
```

Stato Aggiornabile Atomicamente

- Gli algoritmi non-bloccanti sono semplici da progettare ogni qualvolta lo stato dei oggetti è aggiornabile mediante una singola istruzione atomica
  - ✔ stato formato da un <u>solo</u> campo
- Esempio: **`ConcurrentStack`**
  - ✔ stato dello stack = riferimento alla cima dello stack
- Lo stato di uno stack può essere rappresentato ed aggiornato atomicamente con le classi del package **`java.util.concurrent.atomic`**

Uno Stack *Lock-Free* (1)

```
public class ConcurrentStack <E> {
    AtomicReference<Node<E>> top =
          new AtomicReference<Node<E>>();

    public E pop() { ... }
    public void push(E item) { ... }
    private static class Node <E> {
        public final E item;
        public Node<E> next;
        public Node(E item) {
            this.item = item;
        }
    }
}
```

Uno Stack *Lock-Free* (2)

```
public E pop() {
    Node<E> oldHead, newHead;
    do {
        oldHead = top.get();
        if (oldHead == null)
            return null;
        newHead = oldHead.next;
    } while (!top.compareAndSet(oldHead, newHead));
    return oldHead.item;
}
```

```
public void push(E item) {
    Node<E> newHead = new Node<E>(item);
    Node<E> oldHead;
    do {
        oldHead = top.get();
        newHead.next = oldHead;
    } while (!top.compareAndSet(oldHead, newHead));
}
```

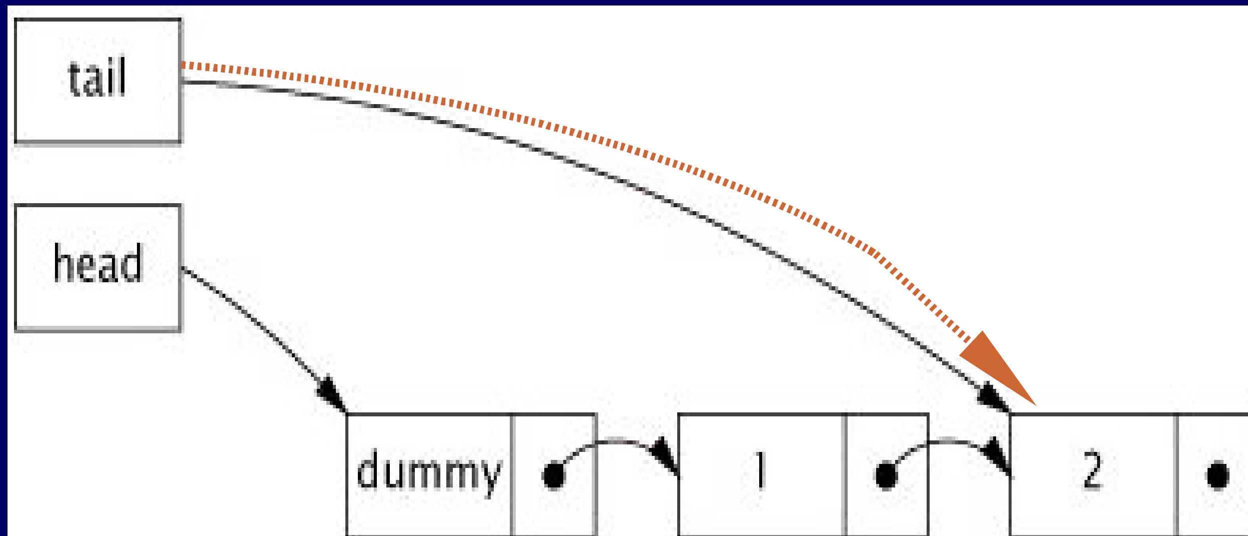
Stato Composito

- Le vere difficoltà nascono quando lo stato degli oggetti è composito
  - la singola componente dello stato (singolo campo) è aggiornabile atomicamente...
  - ...ma lo stato composito *NON* è aggiornabile atomicamente, nella sua interezza: servono diverse operazioni
- Es.: coda semplicemente collegata: **`LinkedQueue`**
- Lo stato è formato da due campi:
  - riferimento di testa
  - riferimento di coda
- Non è disponibile una singola istruzione che consenta di aggiornarli *atomicamente* entrambi

Linked Queue (1)

- Implementazione *wait-free*
  - a lista semplicemente collegata
  - con nodo *sentinella* di testa

- Esempio di stato *consistente*

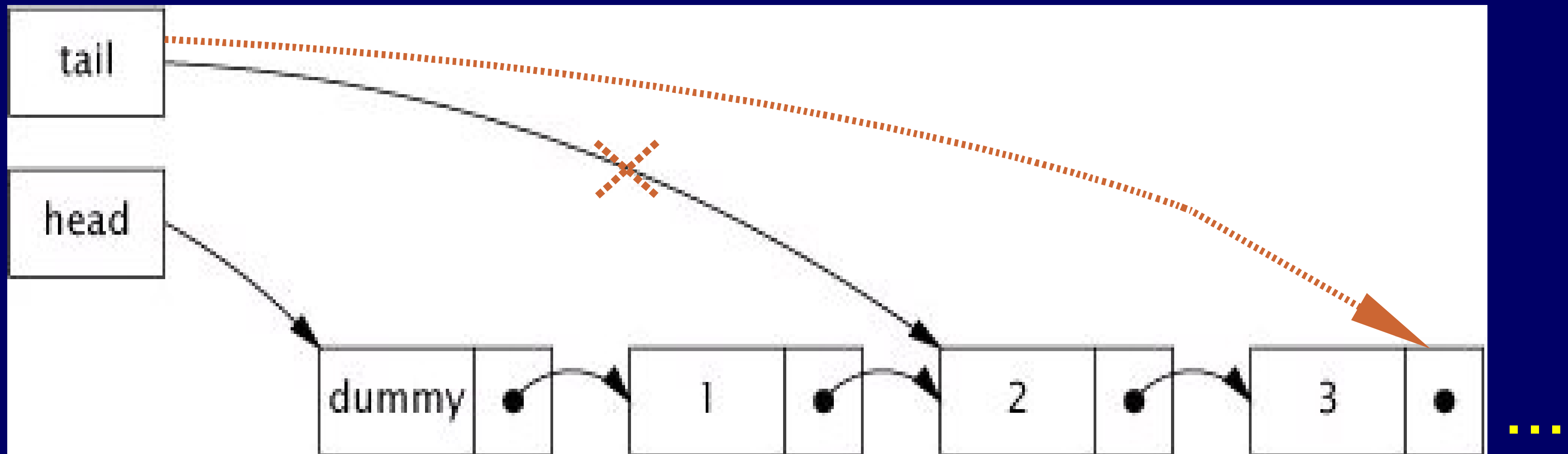


Linked Queue (2)

```
public class LinkedQueue <E> {

    private static class Node <E> {
        final E item;
        final AtomicReference<Node<E>> next;

        public Node(E item, Node<E> next) {
            this.item = item;
            this.next = new AtomicReference<Node<E>>(next);
        }
    }

    private final Node<E> dummy = new Node<E>(null, null);
    private final AtomicReference<Node<E>> head
            = new AtomicReference<Node<E>>(dummy);
    private final AtomicReference<Node<E>> tail
            = new AtomicReference<Node<E>>(dummy);

    public boolean put(E item) {
        ...
    }
}
```

Linked Queue (3)

- Esempio di stato *inconsistente* (non *quiescente*):



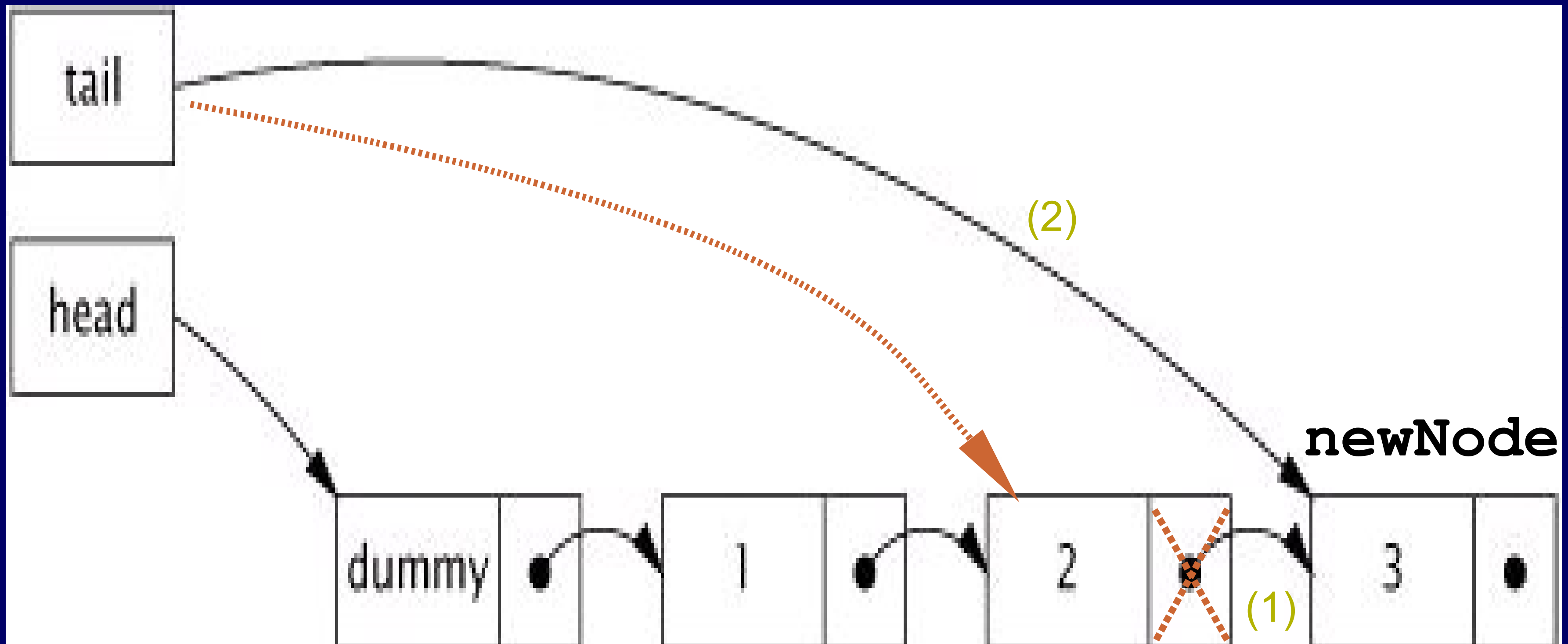
- Come riconoscerlo?
  - Lo stato è consistente se e solo se **`tail.next==null`**
  - n.b. predicato valutabile *atomicamente*
- Come "ripararlo"?
  - **`tail=tail.next`** *(ripetuto più volte se necessario)*

Linked Queue (4)

- Modifiche necessarie per un inserimento

  (1) `tail.next = newNode`

  (2) `tail = newNode`



Linked Queue: Algoritmo *Michael-Scott*

```
public class LinkedQueue <E> {
    ...
    public boolean put(E item) {
        Node<E> newNode = new Node<E>(item, null);
        while (true) {
            Node<E> curTail = tail.get();
            Node<E> tailNext = curTail.next.get();
            if (curTail == tail.get()) {
                if (tailNext != null) {                         (A)
                    // Queue in intermediate state, advance tail
                    tail.compareAndSet(curTail, tailNext);       (B)
                } else {
                    // In quiescent state, try inserting new node
            (1)     if (curTail.next.compareAndSet(null, newNode)) {  (C)
                        // Insertion succeeded, try advancing tail
            (2)         tail.compareAndSet(curTail, newNode);         (D)
                        return true;
                    }
                }
            }
        }
    }
}
```

Osservazioni sull'Algoritmo di *Michael-Scott*

A) Si verifica se la coda è in uno stato inconsistente; nel caso significa che qualche thread è tra gli step (C) o (D)

B) Piuttosto che aspettare invano, si preferisce *aiutare* aggiornando **`tail`**! Il ciclo di inserimento ricomincerà

C) La coda risulta in uno stato consistente con **`tail==null`**: si tenta ottimisticamente la prima (1) delle due modifiche necessarie per l'inserimento (l'aggiornamento di **`tail.next`**); in caso di fallimento (rilevato dalla *CompareAndSet*) il ciclo ricomincia

D) La prima (1) modifica di **`tail.next`** è andata a buon fine, si tenta la seconda (2), ovvero lo spostamento di **`tail`** verso la nuova coda; in caso di fallimento non si ritenta: qualcuno ha già aiutato...

Il *Problema ABA* (1)

- Gli algoritmi non-blocking basano la propria correttezza sulla disponibilità di operazioni che permettano di rilevare aggiornamenti concomitanti

- Gli aggiornamenti atomici possono esporre il cosidetto *Problema ABA*
  - situazioni di interferenza non rilevabili mediante una CAS
  - un doppio aggiornamento potrebbe essere erroneamente scambiato per l'assenza di aggiornamento

- Tipica operazione di aggiornamento di una variabile X tramite CAS
  - Lettura: S ← X.GET(); // → A; sia A il valore letto
  - Quindi: X.CAS(S,B); // → B; sia B il valore a cui aggiornare X

Il *Problema ABA* (2)

Problema ABA:

f.d.e. 1
- Lettura: → A
- Quindi: CAS(A,B)

f.d.e. 2
- Lettura: →A
- Quindi: CAS(A,B)
- CAS(B,A)

- Aggiornamenti tramite istruzioni CAS non adatti se la logica del problema richiede di aggiornare atomicamente lo stato solo quando:
  - si è certi che lo stato che si vuole aggiornare sia uguale a quello atteso, *ed inoltre*:
  - che sia *esattamente* lo stesso e non uno equivalente prodotto da altri f.d.e. tra la prima lettura ed il successivo tentativo di aggiornamento
- Problema ben noto: si solleva durante la gestione di aree di memoria da parte dei garbage collector

Prevenire Il *Problema ABA*

- **`java.util.concurrent.atomic.AtomicStampedReference`**
  - consente la creazione di oggetti di stato *timestamped:*
    - Un riferimento a un oggetto
    - Un intero, che in sostanza è un numero di *versione* (*timestamp*)

Constructors

Constructor and Description
`AtomicStampedReference(V initialRef, int initialStamp)` Creates a new `AtomicStampedReference` with the given initial values.

Method Summary

All Methods | **Instance Methods** | **Concrete Methods**

Modifier and Type	Method and Description
`boolean`	`attemptStamp(V expectedReference, int newStamp)` Atomically sets the value of the stamp to the given update value if the current reference is == to the expected reference.
`boolean`	`compareAndSet(V expectedReference, V newReference, int expectedStamp, int newStamp)` Atomically sets the value of both the reference and stamp to the given update values if the current reference is == to the expected reference and the current stamp is equal to the expected stamp.

Riferimenti

- *Java Concurrency in Practice.* Brian Goetz. – Addison Wesley. Capitolo 15.

- *The Art of Multiprocessor Programming*. Maurice Herlihy, Nir Shavit. [Capitolo 9.]