The Concurrent Programming Abstraction

Contiene illustrazioni prese da:

M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Programmazione concorrente

- Un programma concorrente contiene due o più processi (o sottoprocessi – threads) che lavorano assieme per eseguire una determinata applicazione
- Ciascun (sotto)processo è un programma sequenziale
- I (sotto)processi comunicano tra loro utilizzando variabili condivise (shared memory) o scambiandosi messaggi (message passing)
- ASTRAZIONE: i (sotto)processi sono in esecuzione contemporanea
  - E' un'astrazione… in realtà, potrebbe non essere esattamente così…

Le origini

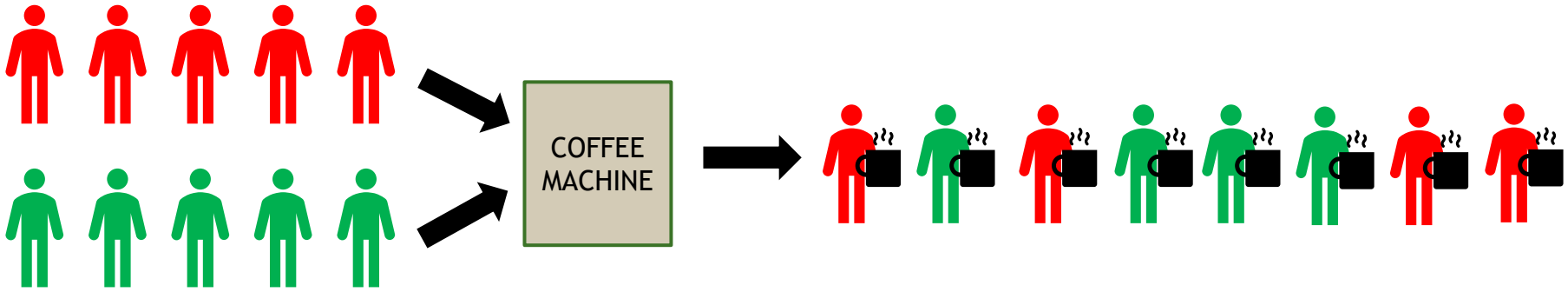
- La programmazione nasce negli anni '60 nell'ambito dei sistemi operativi
- Nasce dall'esigenza di far continuare l'esecuzione dei programmi durante lo svolgimento di (lunghe) operazioni di I/O
- Negli anni '80, si diffondono sistemi operativi con preemptive multitasking
  - Possibilità di mantenere attivi più programmi (o processi) contemporaneamente alternandone l'esecuzione nel processore (interleaving)
  - L'alternarsi dei programmi è sotto il controllo del sistema operativo
  - Richiede supporto hardware (interrupt programmabili)

Sistemi multiprocessore

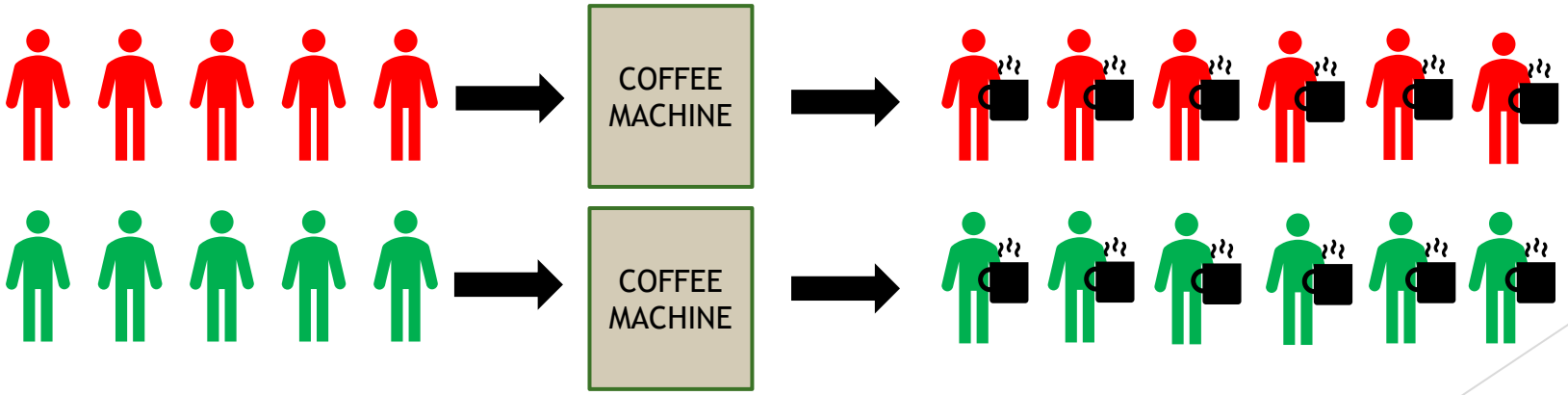
- Successivamente fuorono introdotti i sistemi multiprocessore, dotati di più CPU (e/o di CPU multi-core)
- I sistemi multiprocessore:
  - consentono di eseguire diversi processi in parallelo (ossia, contemporaneamente su CPU diverse)
  - consentono di eseguire le applicazioni (costituite da più processi) più velocemente rispetto all'esecuzione su singola CPU

Concorrenza VS Parallelismo

- Programmazione concorrente: 1 CPU e N task contemporaneamente



- Programmazione parallela: N CPU e N task contemporaneamente



Ma frequentemente capita di avere N CPU e M task con N<M

Esecuzione non sequenziale

- Concorrenza e parallelismo sono accomunati da un aspetto:
  - Esecuzione non sequenziale del programma
  - Idealmente, ogni (sotto)processo che costituisce il programma ha un proprio program counter che avanza autonomamente
- Esempio con due processi, ognuno con due comandi:
  - Processo P: p1,p2
  - Processo Q: q1,q2

- Possibili interleaving in caso di concorrenza (singola CPU):

  p1 -> p2 -> q1 -> q2

  p1 -> q1 -> p2 -> q2

  p1 -> q1 -> q2 -> p2

  q1 -> p1 -> p2 -> q2

  q1 -> p1 -> q2 -< p2

  q1 -> q2 -> p1 -> p2

Esempio

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

- n è una variabile globale (inizializzata a 0)
- k1 e k2 sono variabili locali
- p e q sono processi concorrenti

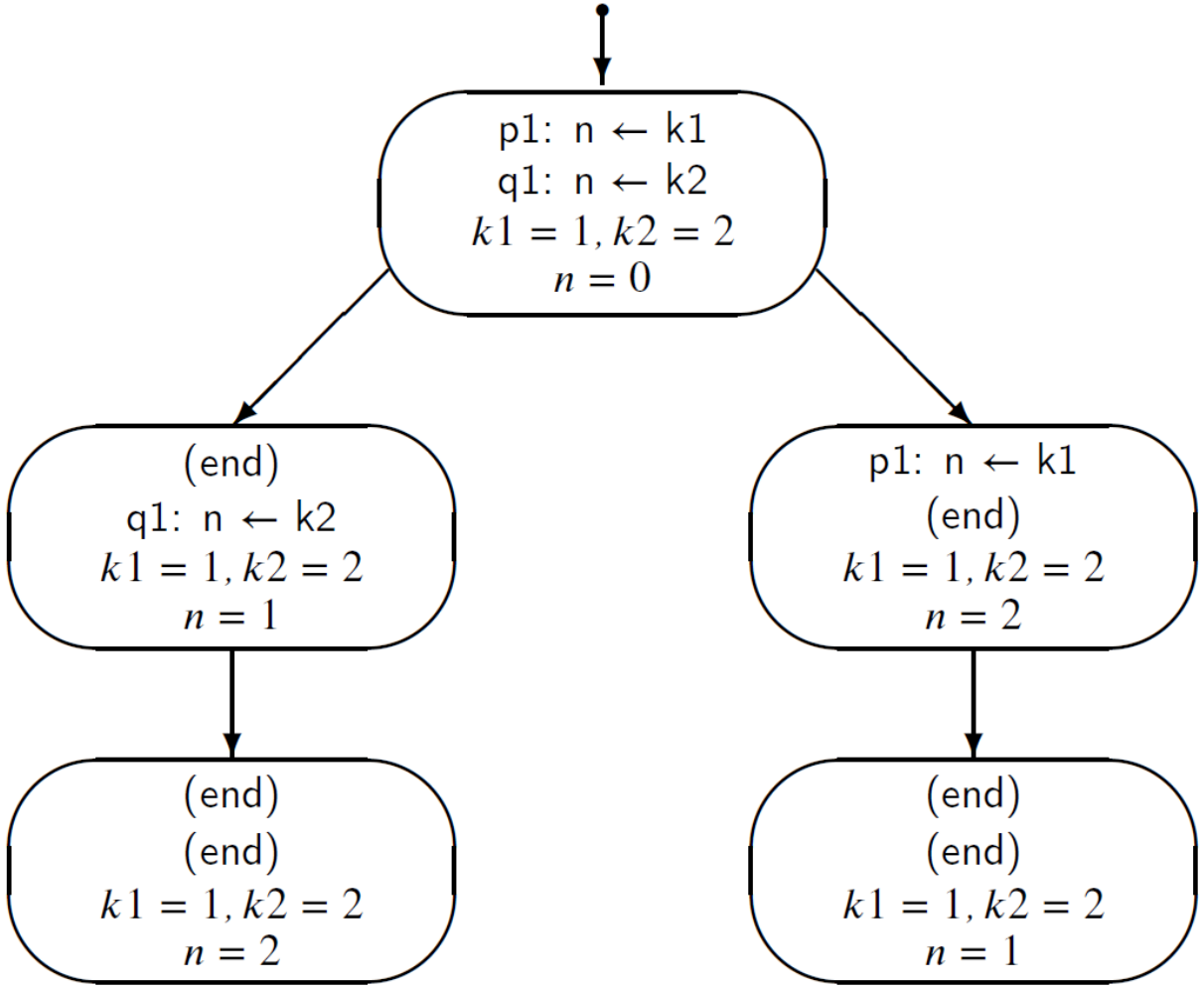
qual è il valore di n al termine?

Analizziamo i possibili comportamenti dell'esempio tramite un sistema di transizioni

- E' dato dalla semantica del linguaggio
- Descrive tutti i possibili passi di computazione che il programma può fare
- Tiene conto dei possibili interleaving dei processi

Esempio:

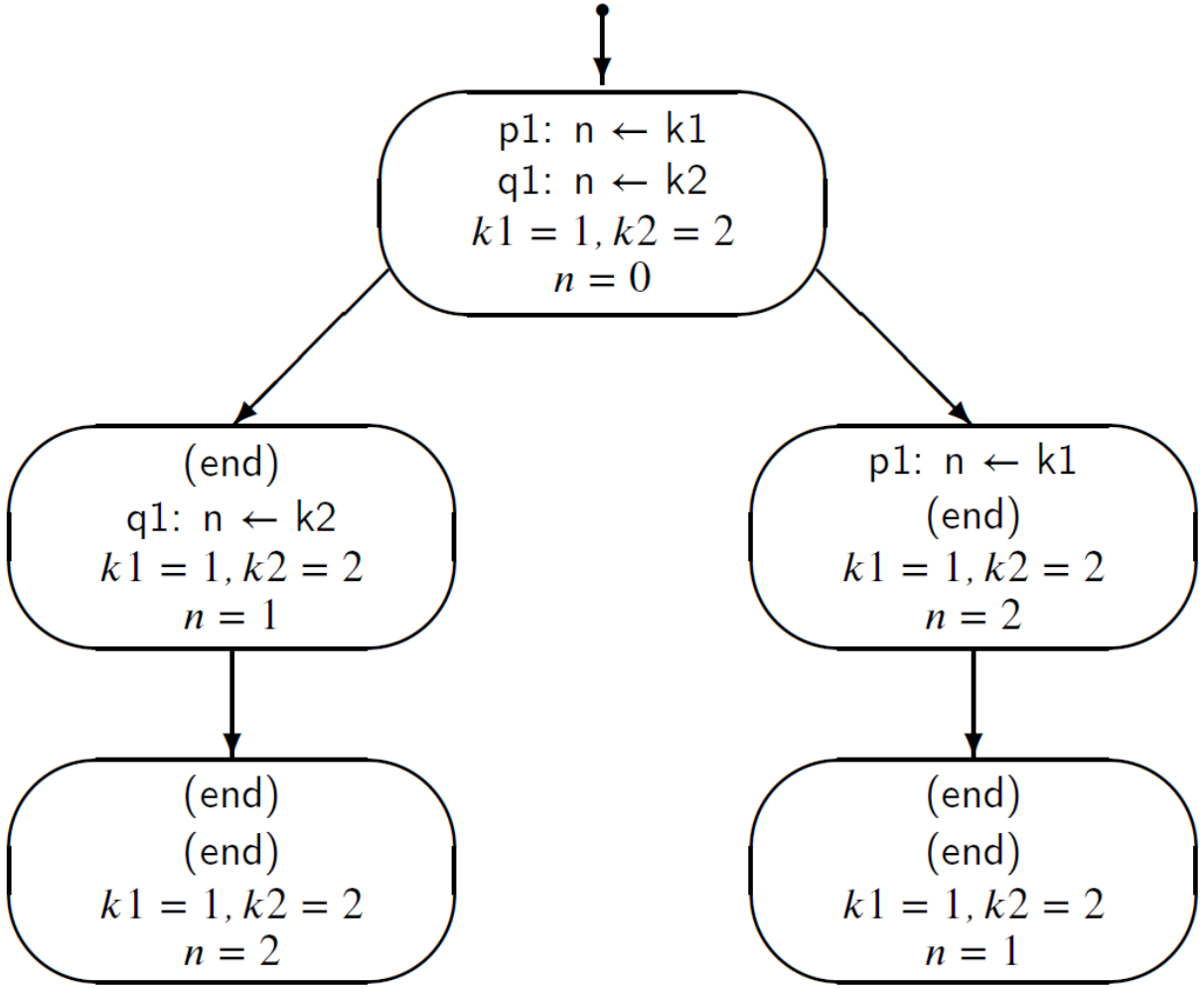
- n ha due possibili valori finali: 2 e 1…



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Il sistema di transizioni:

- E' non deterministico (lo stato iniziale ha due alternative possibile)
- Il non determinismo ASTRAE dal criterio usato dallo scheduler del sistema operativo per scegliere quale processo far avanzare (es. processi prioritari su altri)
- Conoscendo lo scheduler si potrebbe concludere che alcuni cammini nel sistema di transizioni non sono in realtà realizzabili
- Il sistema di transizioni non fa assunzioni sullo scheduler (descrive tutte le esecuzioni possibili)



M. Ben-Ari. Principles of Concurrent and Distributed Programming, Second edition © M. Ben-Ari 2006

Operazioni atomiche

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1 p1: n ← k1	integer k2 ← 2 q1: n ← k2

L'analisi che abbiamo fatto è corretta sotto l'assunzione che le operazioni di assegnamento p1 e q1 siano atomiche

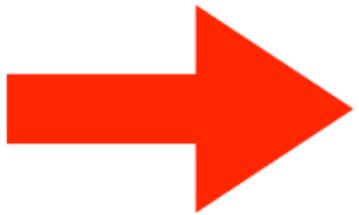
- elementari e non interrompibili

Ma non abbiamo tenuto conto del compilatore!

Il compilatore

Il compilatore traduce un singolo comando del linguaggio di alto livello in una sequenza di operazioni nel linguaggio macchina

```
n = k + 1;
```



```
load  R1,k
add   R1,#1
store R1,n
```

nota: R1 è un registro

Analisi al livello macchina

In pratica, l'interleaving ha luogo al livello del linguaggio macchina, e non del linguaggio di alto livello

- Sono le operazioni assembler ad alternarsi, non i comandi del linguaggio di alto livello

Analisi al livello macchina

Dovremmo rivedere l'analisi dell'esempio di programma concorrente considerandolo così:

Algorithm 2.1: Trivial concurrent program	
integer n ← 0	
p	**q**
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

p1: load R1,k1
p2: store R1,n

q1: load R1,k2
q2: store R1,n

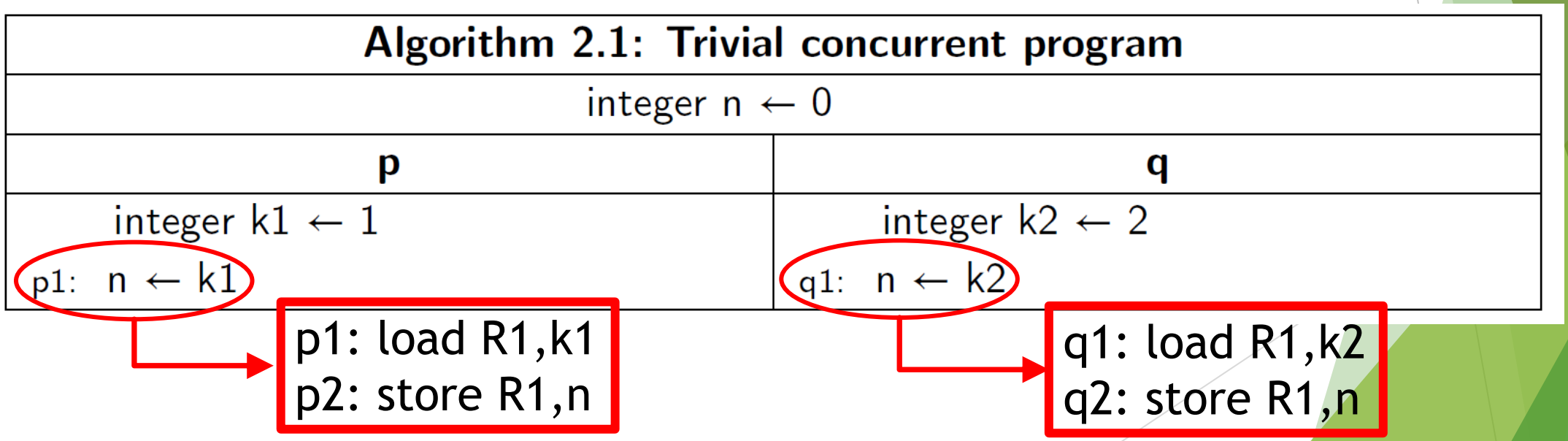
Immaginiamo che ogni processo abbia un proprio registro R1 (vedrete a S.O. come questo è reso possibile)

TUTTO OK! Come prima, deduciamo che n alla fine potrebbe avere valore 1 o 2 (a seconda che l'ultima operazione eseguita sia p2 o q2)

E in caso di parallelismo?

Altro problema: se p e q fossero eseguiti su due CPU diverse, le operazioni p2 e q2 potrebbero essere eseguite contemporaneamente

- Non in interleaving, ma in "true parallelism"



E in caso di parallelismo?

Altro problema: se p e q fossero eseguiti su due CPU diverse, le operazioni p2 e q2 potrebbero essere eseguite contemporaneamente

- Non in interle

Che succede se due processi cercano di scrivere contemporaneamente la stessa variabile globale?

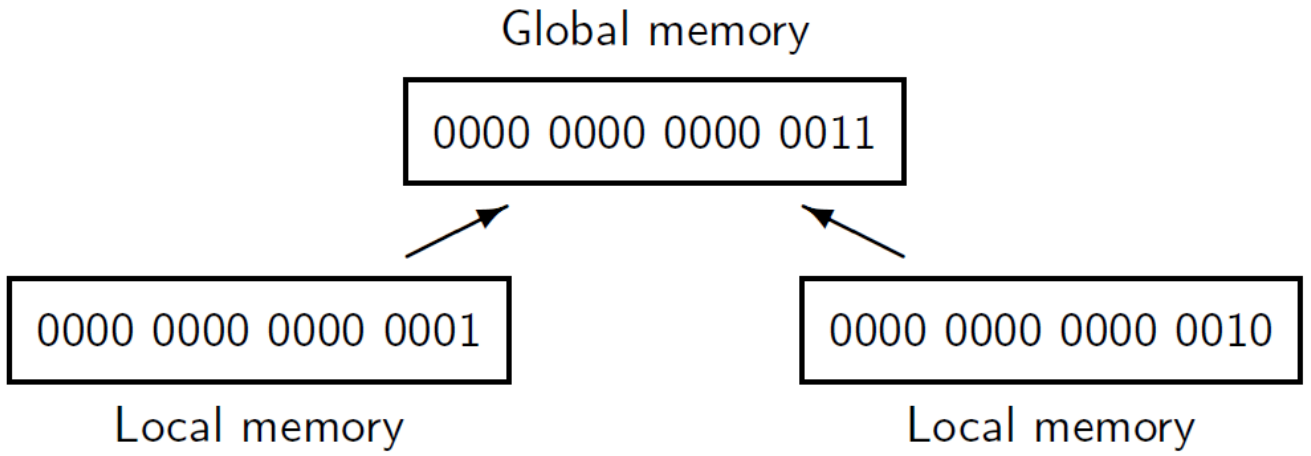
p	q
integer k1 ← 1	integer k2 ← 2
p1: n ← k1	q1: n ← k2

p1: load R1,k1
p2: store R1,n

q1: load R1,k2
q2: store R1,n

Scenario possibile?

Modificando contemporaneamente la stessa locazione di memoria i due processi potrebbero sovrapporre i propri risultati?



Scenario possibile?

Modificando contemporaneamente la stessa locazione di memoria i due processi potrebbero sovrapporre i propri risultati?

NO!

Anche in caso di parallelismo l'hardware garantisce che al più un processo per volta possa scrivere una certa locazione di memoria

Quindi…

Il fatto che anche in un contesto di parallelismo le modifiche contemporanee alla stessa locazione di memoria siano gestite dall'hardware eseguendole una per volta, ci riporta in uno scenario di concorrenza

- I processi possono operare in parallelo solo su locazioni di memoria distinte

- La concorrenza (interleaving) è un'astrazione che consente di fare molte analisi dei programmi che varranno anche in situazioni di parallelismo
- Come per lo scheduling del sistema operativo, anche le scelte operate dall'hardware saranno modellate tramite non determinismo nei sistemi di transizione

Analisi a livello di macchina VS Analisi ad alto livello

Torniamo sulla questione dell'analisi a livello macchina con un altro esempio:

Algorithm 2.3: Atomic assignment statements	
integer n ← 0	
p	**q**
p1: n ← n + 1	q1: n ← n + 1

Ragioniamo ad alto livello, quindi assumiamo che gli assegnamenti (del linguaggio di programmazione) siano operazioni atomiche (indivisibili)

Che valore ha n alla fine?

Analisi a livello di macchina VS Analisi ad alto livello

Queste sono le due (uniche) esecuzioni possibili del nuovo esempio di programma concorrente

- Corrispondono a due cammini alternativi nel sistema di transizioni che descrive il comportamento del programma
- In entrambi i casi alla fine n vale 2

Process p	Process q	n
p1: n←n+1	q1: n←n+1	0
(end)	**q1: n←n+1**	1
(end)	(end)	2

Process p	Process q	n
p1: n←n+1	**q1: n←n+1**	0
p1: n←n+1	(end)	1
(end)	(end)	2

Analisi a livello di macchina VS Analisi ad alto livello

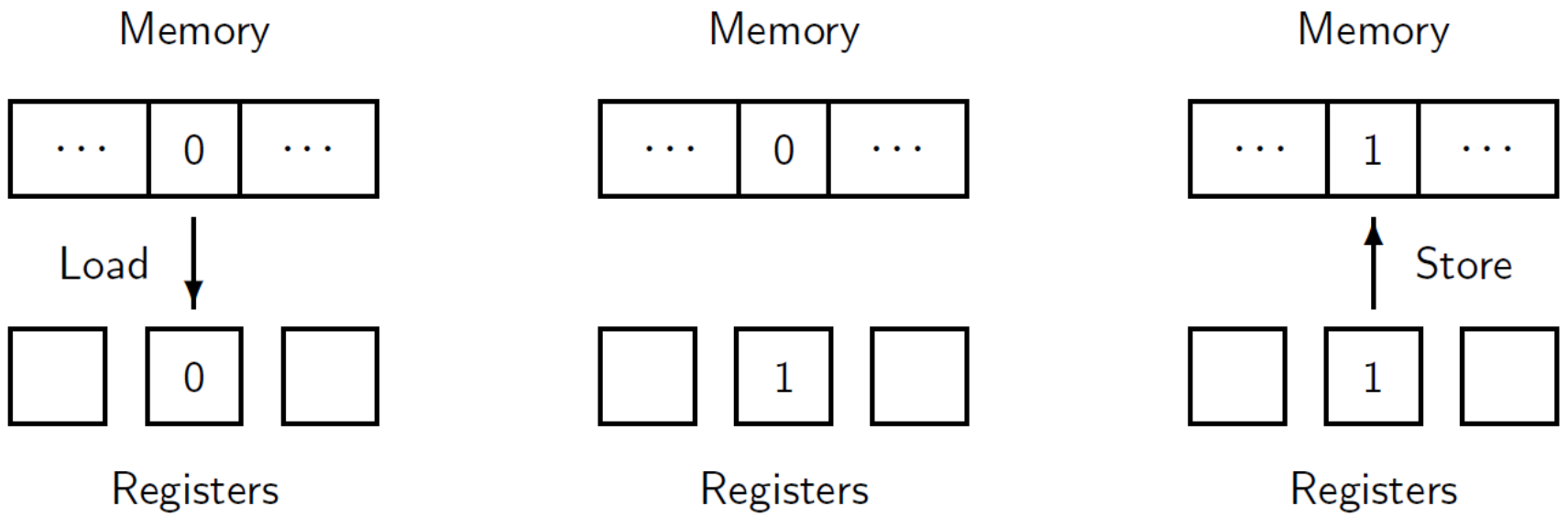
Vediamo ora lo stesso programma a livello macchina

- Queste le corrispondenti istruzioni in assembler

Algorithm 2.6: Assignment statement for a register machine	
integer n ← 0	
p	**q**
p1: load R1,n	q1: load R1,n
p2: add R1,#1	q2: add R1,#1
p3: store R1,n	q3: store R1,n

Analisi a livello di macchina VS Analisi ad alto livello

- Rappresentazione schematica dell'esecuzione di ognuno dei due processi



Analisi a livello di macchina VS Analisi ad alto livello

Questa è una possibile esecuzione descritta al livello di linguaggio macchina

- in questa particolare esecuzione alla fine n vale 1…
- Rispetto a prima, la load di un processo non è successiva alla store dell'altro
- chiaramente ci sono altre esecuzioni possibili in cui n alla fine vale 2 (ad es. p1,p2,p3,q1,q2,q3)
- Questa esecuzione non è colta dalla descrizione ad alto livello del comportamento (la semantica del linguaggio di programmazione)
- L'analisi ad alto livello non è completa (non cattura tutti i comportamenti possibili)

Process p	Process q	n	p.R1	q.R1
p1: load R1,n	q1: load R1,n	0	?	?
p2: add R1,#1	**q1: load R1,n**	0	0	?
p2: add R1,#1	q2: add R1,#1	0	0	0
p3: store R1,n	**q2: add R1,#1**	0	1	0
p3: store R1,n	q3: store R1,n	0	1	1
(end)	**q3: store R1,n**	1	1	1
(end)	(end)	1	1	1

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Possibile soluzione:

- simulare l'interleaving che si ha a livello macchina con assegnamenti a variabili temporanee (locali) nel linguaggio ad alto livello per disaccoppiare load e store alle variabili globali (che sono condivise tra processi)

Algorithm 2.4: Assignment statements with one global reference	
integer n ← 0	
p	**q**
integer temp	integer temp
p1: temp ← n	q1: temp ← n
p2: n ← temp + 1	q2: n ← temp + 1

Nota: p e q usano due variabili temp diverse

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Questo consente di catturare ad alto livello anche il comportamento di basso livello che nel caso precedente non veniva descritto

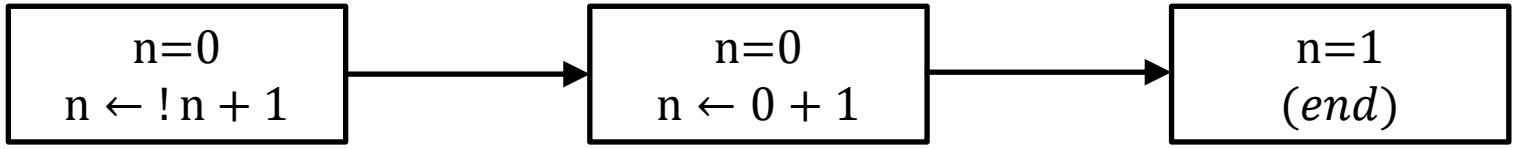
Process p	Process q	n	p.temp	q.temp
p1: temp←n	q1: temp←n	0	?	?
p2: n←temp+1	**q1: temp←n**	0	0	?
p2: n←temp+1	q2: n←temp+1	0	0	0
(end)	**q2: n←temp+1**	1	0	0
(end)	(end)	1	0	0

Nota: p e q usano due variabili temp diverse

Simulare il comportamento a livello macchina nel linguaggio ad alto livello

Altro modo di effettuare la stessa simulazione è del comportamento di basso livello consiste nell'assumere nel linguaggio un'operazione di dereferenziazione esplicita delle variabili

- Invece di scrivere $\mathbf{n \leftarrow n + 1}$ scriveremo $\mathbf{n \leftarrow !\, n + 1}$ dove **!** rende esplicita l'operazione di lettura della variabile (richiede un passo di computazione)
- Esecuzione:



è una soluzione meno realistica (il valore 0 letto dalla variabile viene temporaneamente scritto nel codice del programma…) ma che cattura correttamente quello che accade a livello più basso (lettura e scrittura in due passi)

Ricapitolando

Possiamo usare la concorrenza come astrazione che:

1. descrive il comportamento di processi tramite interleaving,
2. formalizzandoli come sistemi di transizioni non deterministici
3. ottenuti dalla semantica del linguaggio di alto livello

Questa astrazione:

- Cattura anche aspetti di parallelismo
  - in particolare sulle variabili globali condivise
- Cattura anche aspetti di interleaving a livello macchina
  - eventualmente aggiungendo variabili temporanee locali o esplicitando le operazioni di dereferenziazione

Shared Memory VS Message Passing

Shared memory VS Message Passing

Per coordinare il lavoro del programma, è frequente che due o più (sotto)processi debbano comunicare tra loro:

- Per sincronizzarsi in certi momenti
- Per scambiarsi dati

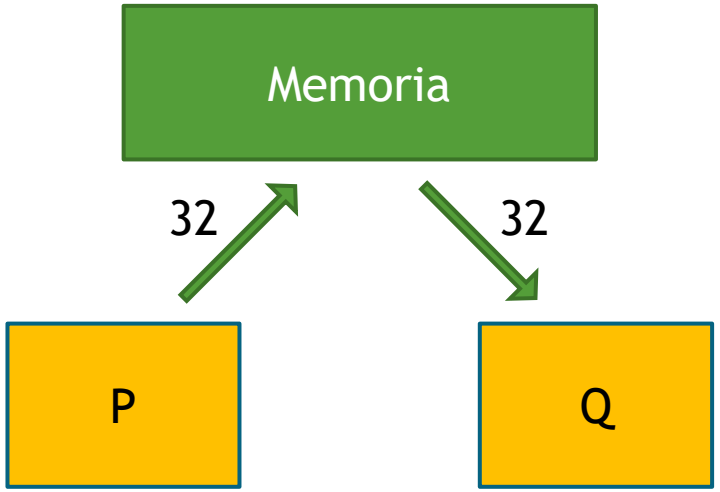
Due modi diversi per far comunicare processi:

- Memoria condivisa (shared memory)
- Scambio di messaggi (message passing)

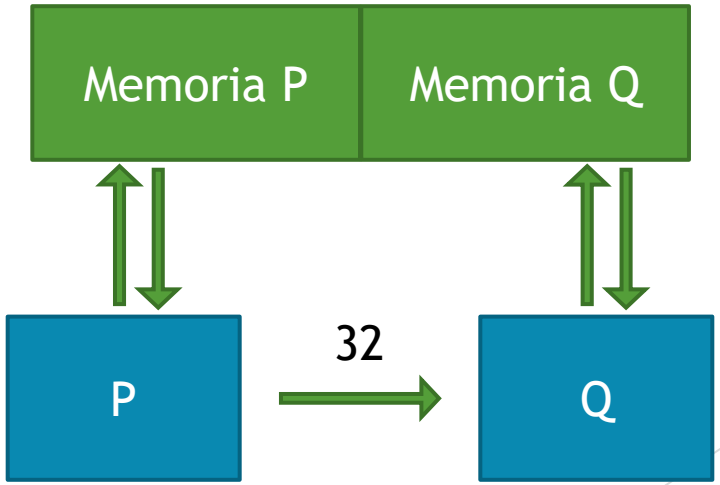
Shared Memory VS Message Passing

Shared Memory:

- i processi (o thread, in questo caso) possono accedere alle stesse aree di memoria
- si sincronizzano e comunicano tra loro scrivendo e leggendo variabili condivise



Shared Memory

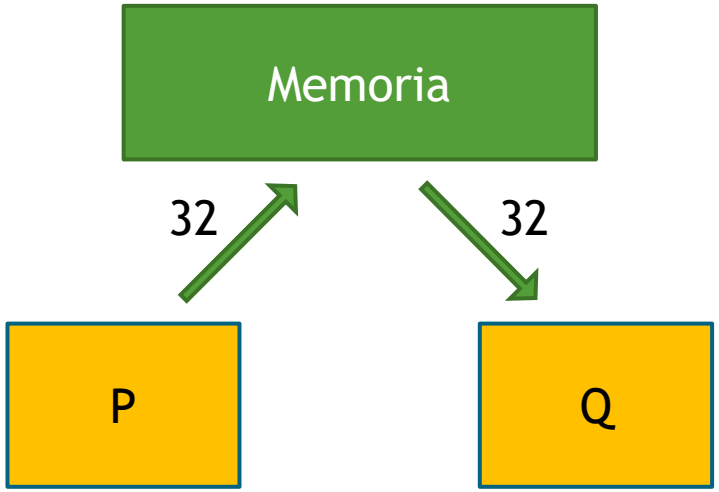


Message Passing

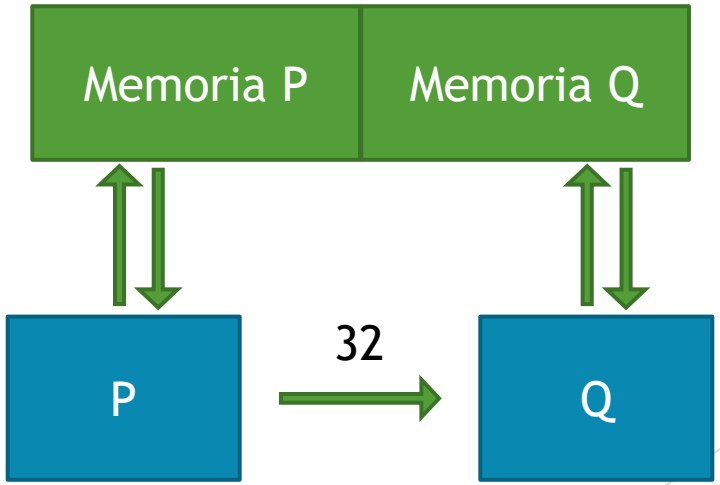
Shared Memory VS Message Passing

Message Passing:

- i processi accedono ad aree diverse della memoria, ma possono inviarsi messaggi e sincronizzarsi usando servizi di inter-process communication (IPC) messi a disposizione dal sistema operativo



Shared Memory



Message Passing

Esempio di comunicazione con shared memory (in pseudocodice)

```
// variabili globali (memoria condivisa)
int x = 0;

// THREAD 1                    // THREAD 2
producer() {                   consumer() {
  int k = 6;                     while (x==0) sleep(10);
  x = fattoriale(k);             print(x); // stampa 720
}                              }
```

Il thread consumer attende che il producer abbia scritto un valore in x usando una tecnica di busy waiting (testa il valore di x ogni 10 millisecondi)

- I due thread si sincronizzano! (il secondo attende il primo)
- la strategia busy waiting è inefficiente (il secondo thread consuma tempo di CPU solo per testare ripetutamente il valore di x)

Esempio di comunicazione con shared memory (in pseudocodice)

```
// variabili globali (memoria condivisa)
int x = 0;

// THREAD 1                    // THREAD 2
producer() {                   consumer() {
  int k = 6;                     if (x==0) sleep();
  x = fattoriale(k);             print(x); // stampa 720
  wakeup();                    }
}
```

Il runtime del linguaggio può mettere a disposizione servizi di segnalazione tra thread

- Chiamando sleep() il consumer si mette in attesa
- Chiamando wakeup() il producer sblocca il consumer segnalandogli che il dato è pronto per essere letto

Esempio di comunicazione con message passing (in pseudocodice)

```
// PROCESSO 1
producer() {
   int k = 6;
   int x = fattoriale(k);
   send(x);
}
```

```
// PROCESSO 2
    consumer() {
       int y = receive();
       print(y); // stampa 720
    }
```

Il sistema operativo mette a disposizione dei processi un vero e proprio servizio di messaggistica (Inter-Process Communication, IPC)

- chiamando receive() il processo consumer si mette in attesa di ricevere messaggi
- chiamando send(x) il processo producer sblocca il consumer e gli passa il valore di x
- il trasferimento del dato dalla memoria di un processo a quella dell'altro è a carico del sistema operativo

Necessità di meccanismi di sincronizzaizone

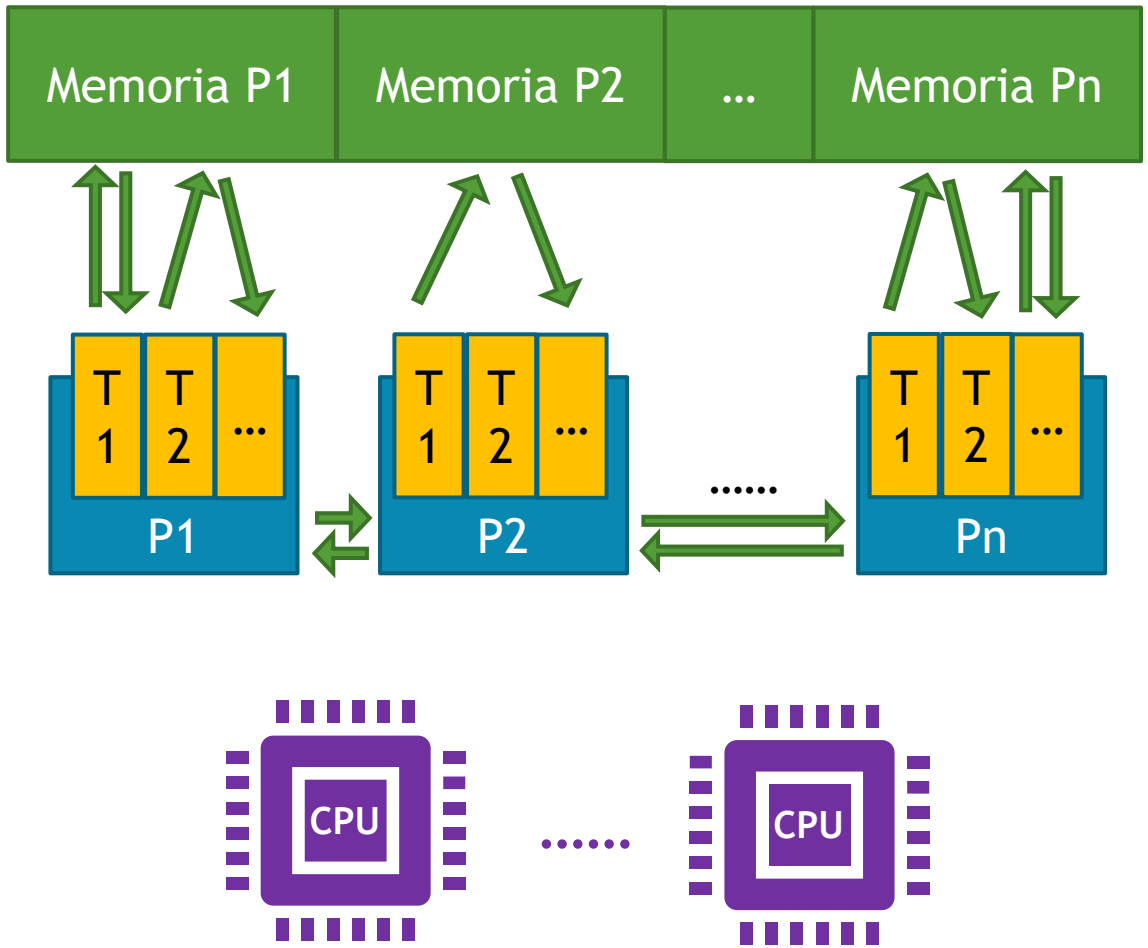
Gli esempi mostrano l'importanza di meccanismi di sincronizzazione

- busy waiting
- sleep() e wakeup()
- send() e receive()
- …

Nel caso dei thread questi meccanismi possono essere realizzati a livello di runtime del linguaggio di programmazione

Nel caso di processi, deve necessariamente essere il sistema operativo a fornire servizi di sincronizzazione e comunicazione (i processi sono isolati l'uno dall'altro)

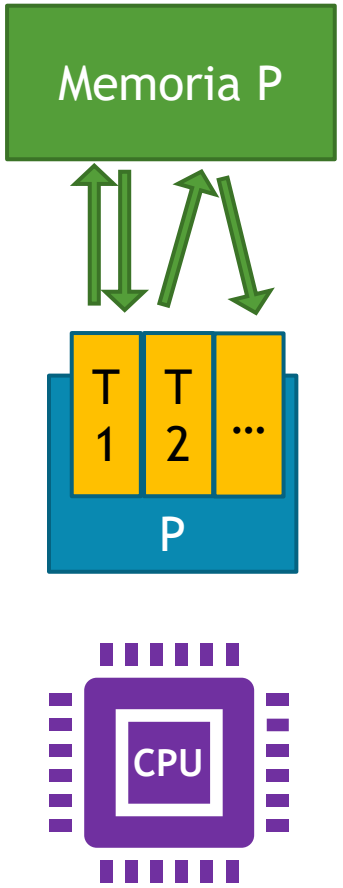
Visione astratta (e semplificata) dell'hardware e del sistema operativo



In generale:

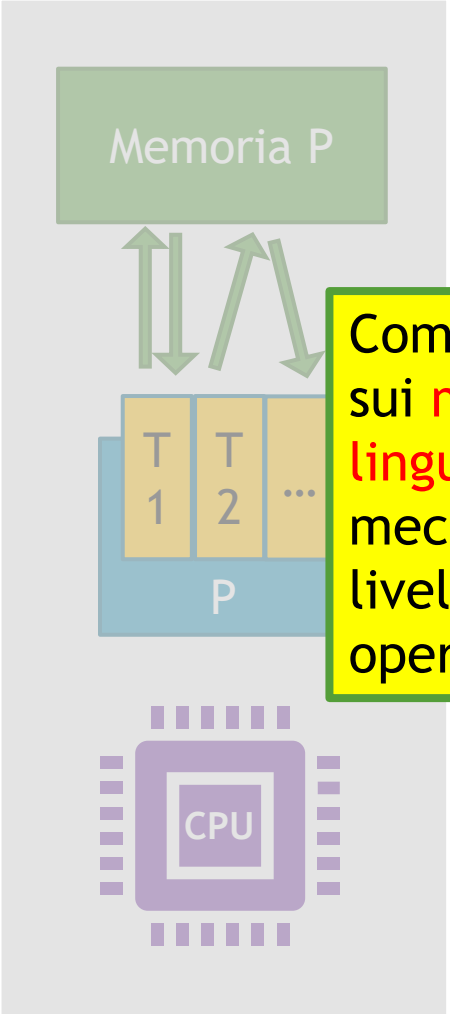
- il sistema operativo può mantenere in esecuzione vari processi fornendo servizi di IPC
- ogni processo può prevedere uno o più thread, che condividono tra loro la memoria del processo
- i processi (e i loro thread) possono essere eseguiti in parallelo su più CPU

Scenario che consideriamo per studiare la concorrenza (multithreading)



Per studiare le problematiche legate alla programmazione concorrente e i meccanismi di sincronizzazione ci concentreremo su uno scenario multithreading assumendo un singolo processo e una singola CPU

Scenario che consideriamo per studiare la concorrenza (multithreading)



Come al solito, ragioneremo sui modelli e sui costrutti linguistici, astraendo dai meccanismi che si attivano a livello hardware e di sistema operativo