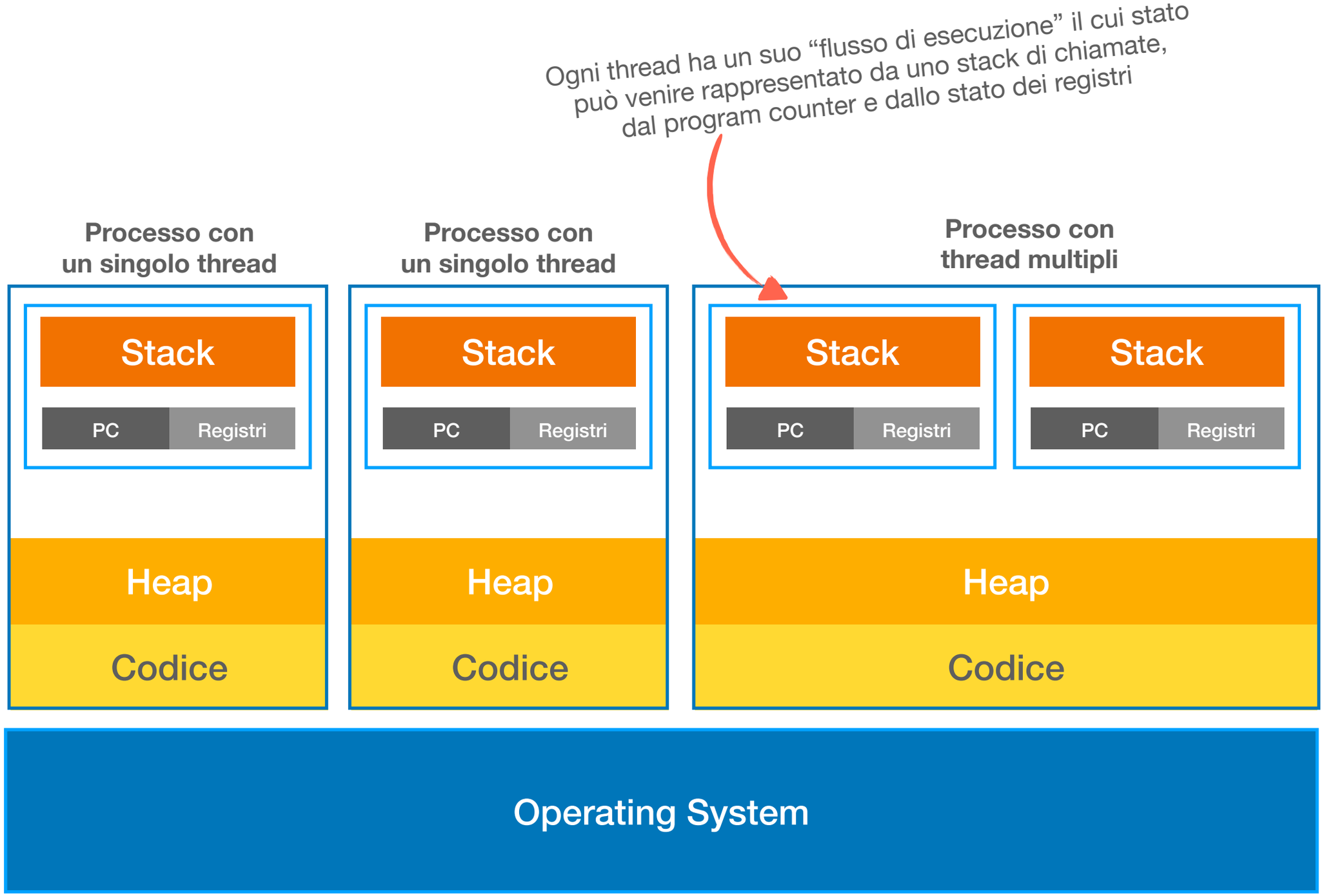
Programmazione e Architetture (Modulo B)

Lezione 15
Thread e concorrenza

Luca Manzoni

Thread e Processi



Thread

Cosa sono?

- I processi, come li abbiamo visti fino ad ora (con un singolo thread) incapsulano due concetti diversi:
  - Flusso di esecuzione: cosa stiamo eseguendo
  - Spazio degli indirizzi: le risorse a nostra disposizione
- Avere thread multipli all'interno dello stesso processo permette di condividere lo spazio degli indirizzi ma mantenere flussi di esecuzione separati
- Vediamo un esempio in cui chiamiamo la stessa funzione (con alcuni argomenti in comune) tra due thread diversi

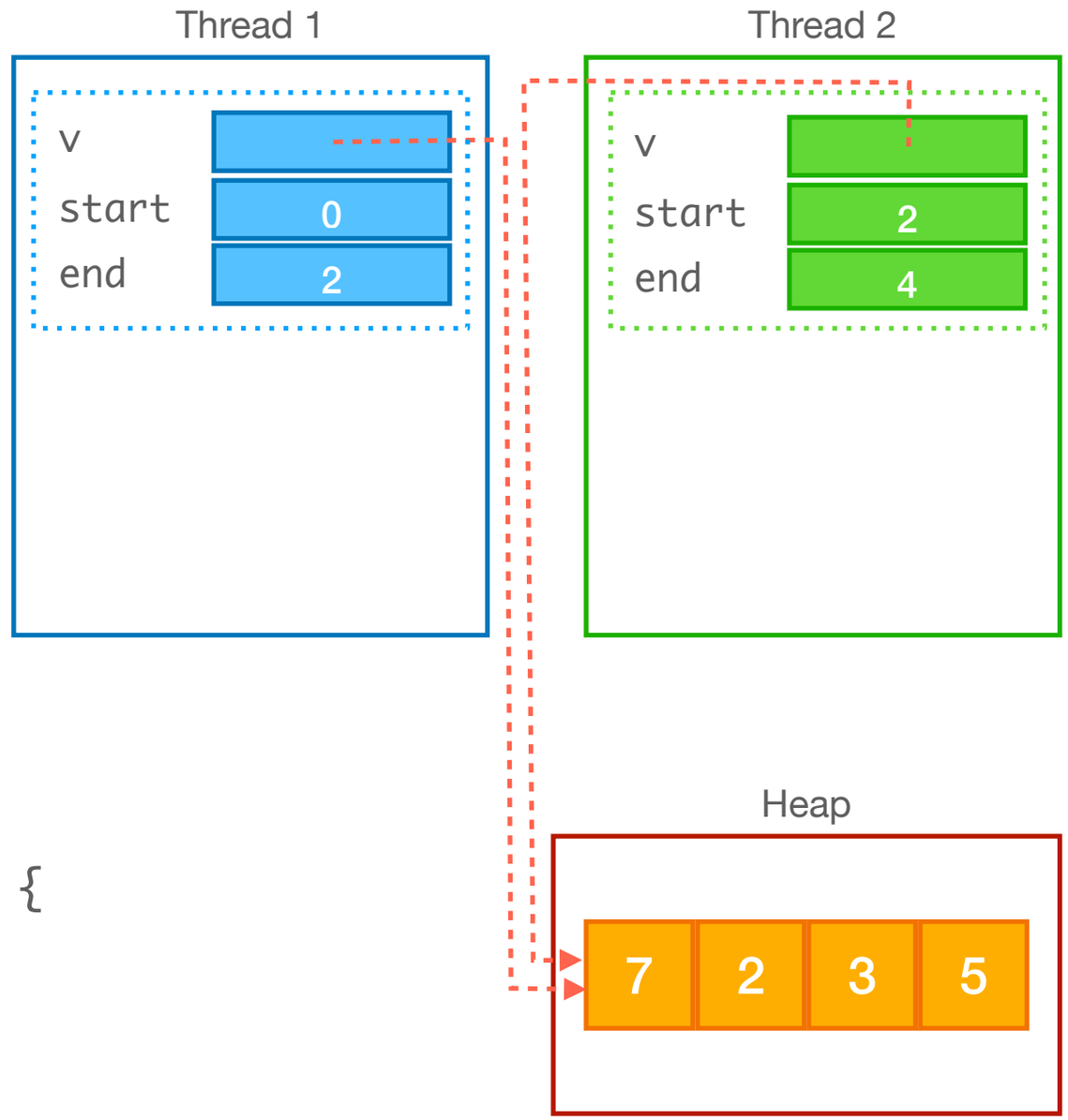
Thread: esempio di esecuzione

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}
```

PC (thread 2)

PC (thread 1)

```
void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```



Thread: esempio di esecuzione

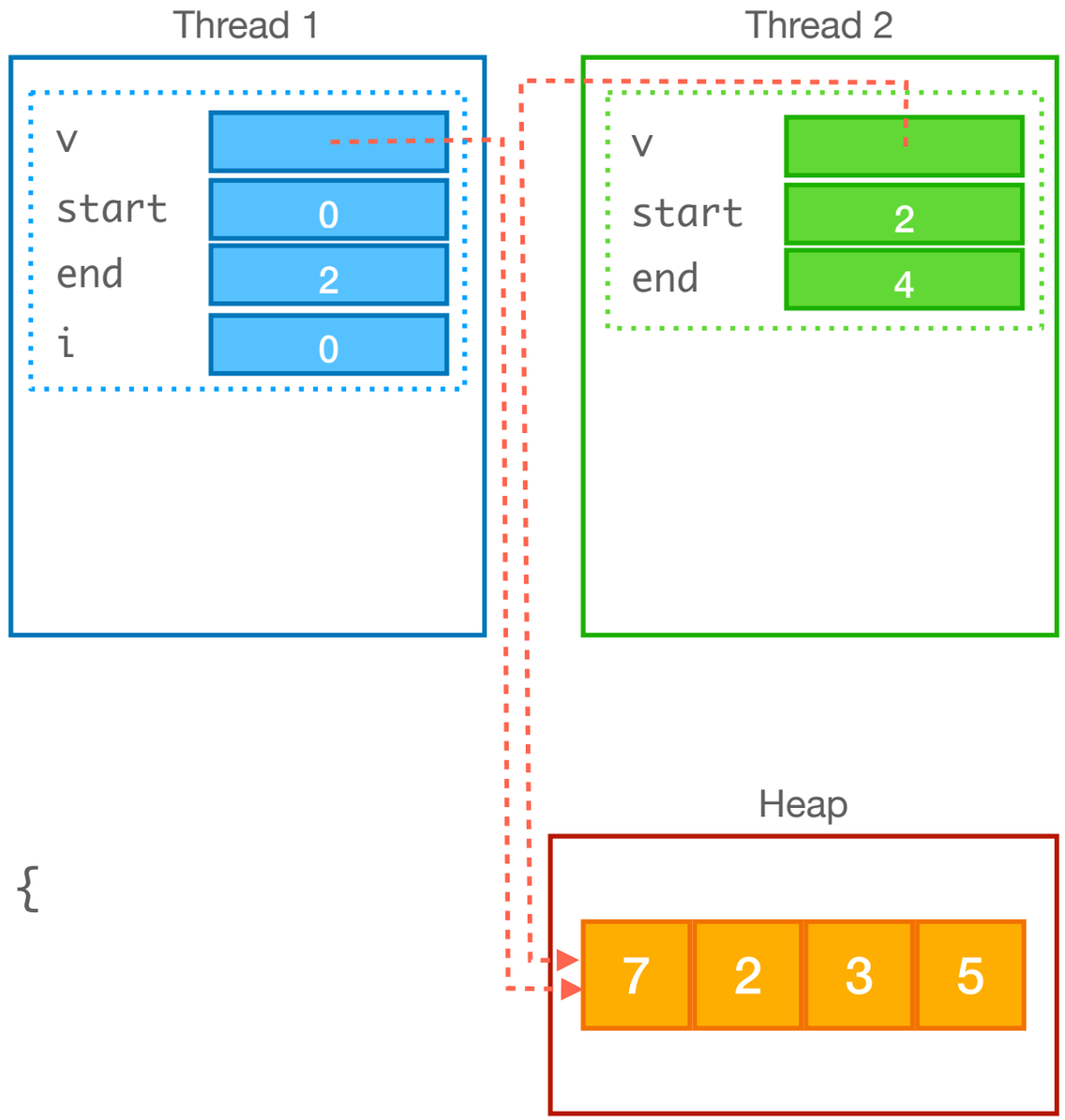
Supponiamo che esegua prima il thread 1

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}
```

PC (thread 2)

```
void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
    v[i] = g(v[i]);
  }
}
```

PC (thread 1)



Thread: esempio di esecuzione

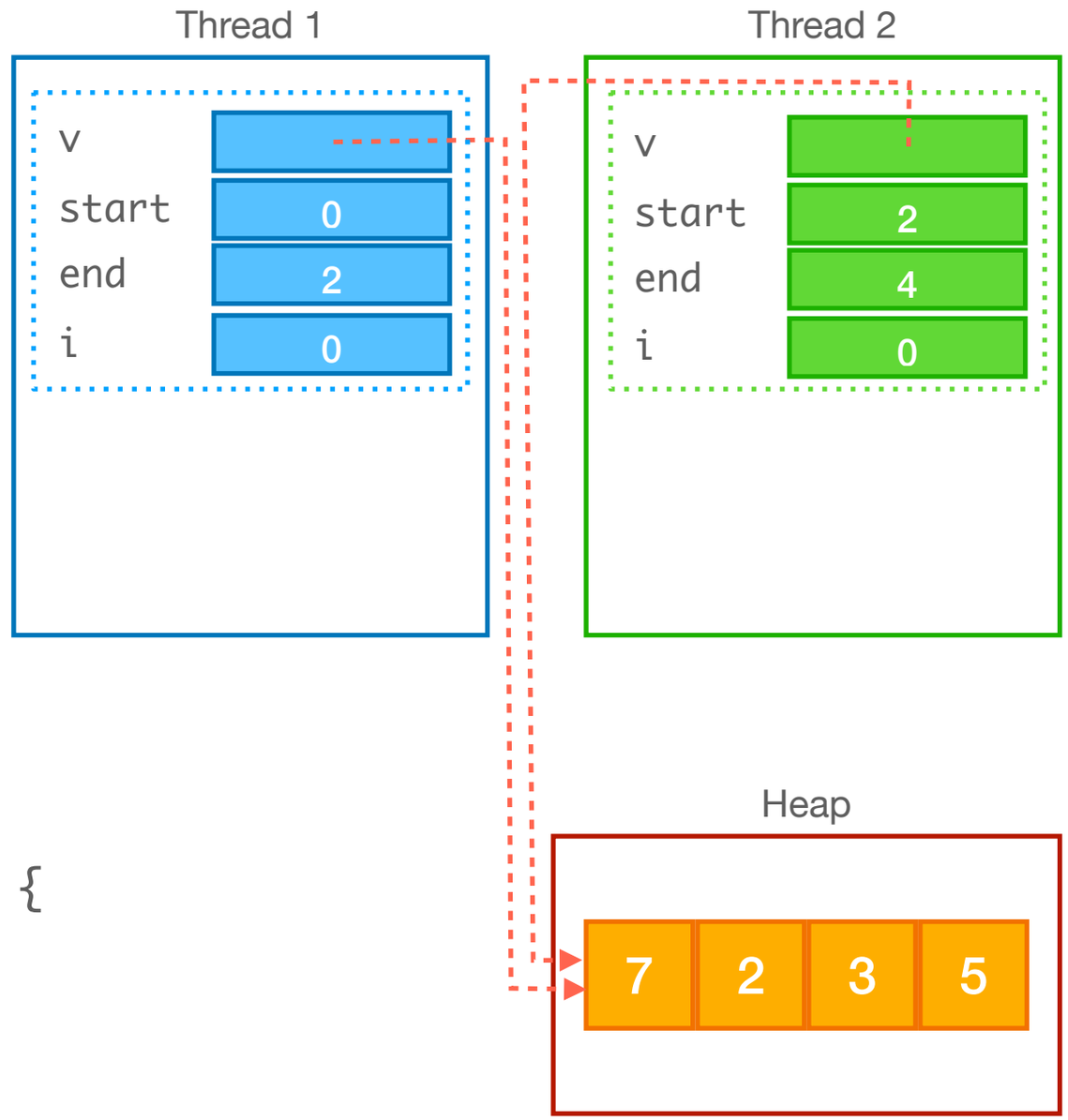
E a seguire il thread 2

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```

PC (thread 2)

PC (thread 1)



Thread: esempio di esecuzione

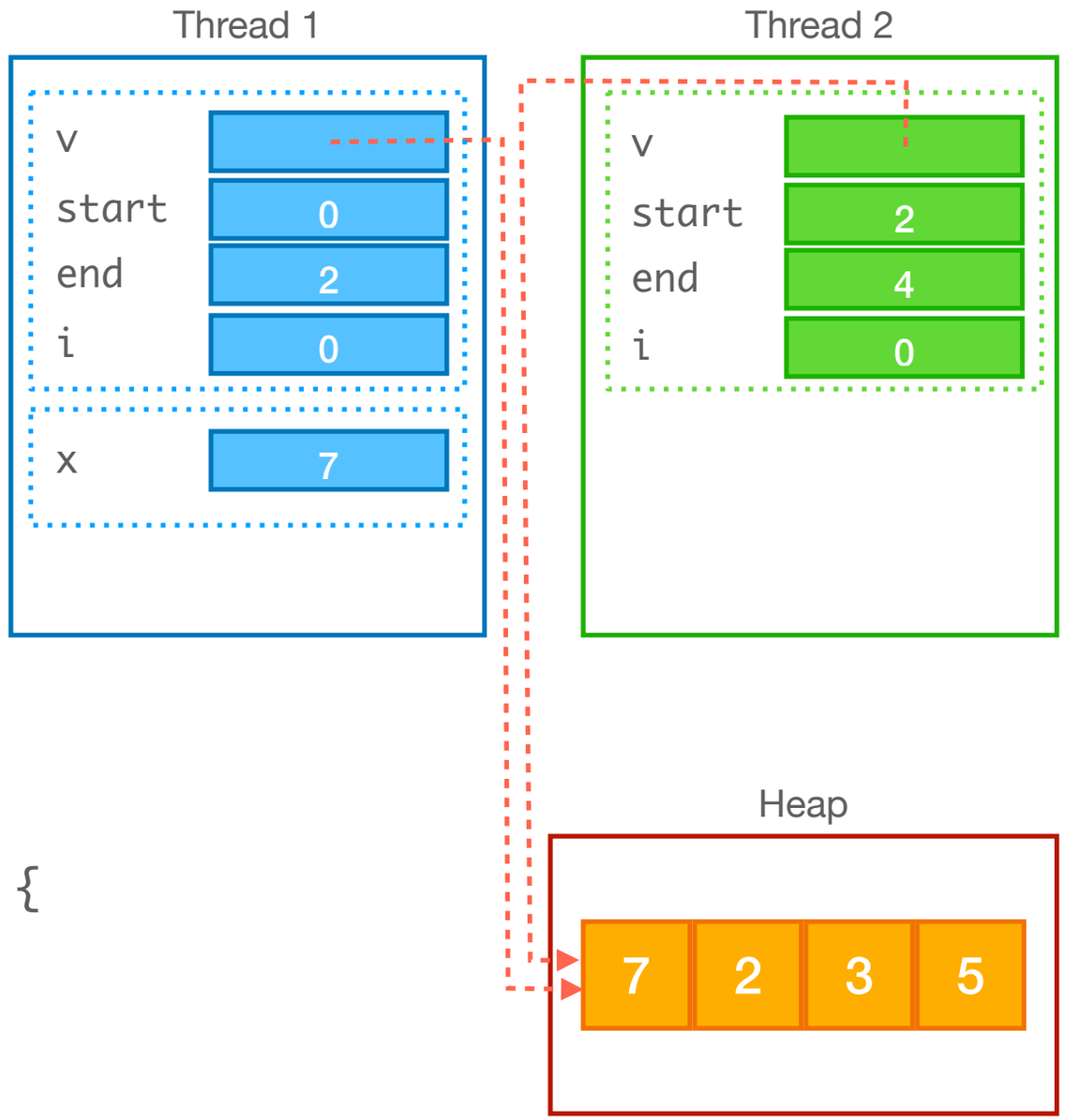
Di nuovo il thread 1

PC (thread 1)

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
    v[i] = g(v[i]);
  }
}
```

PC (thread 2)



Thread: esempio di esecuzione

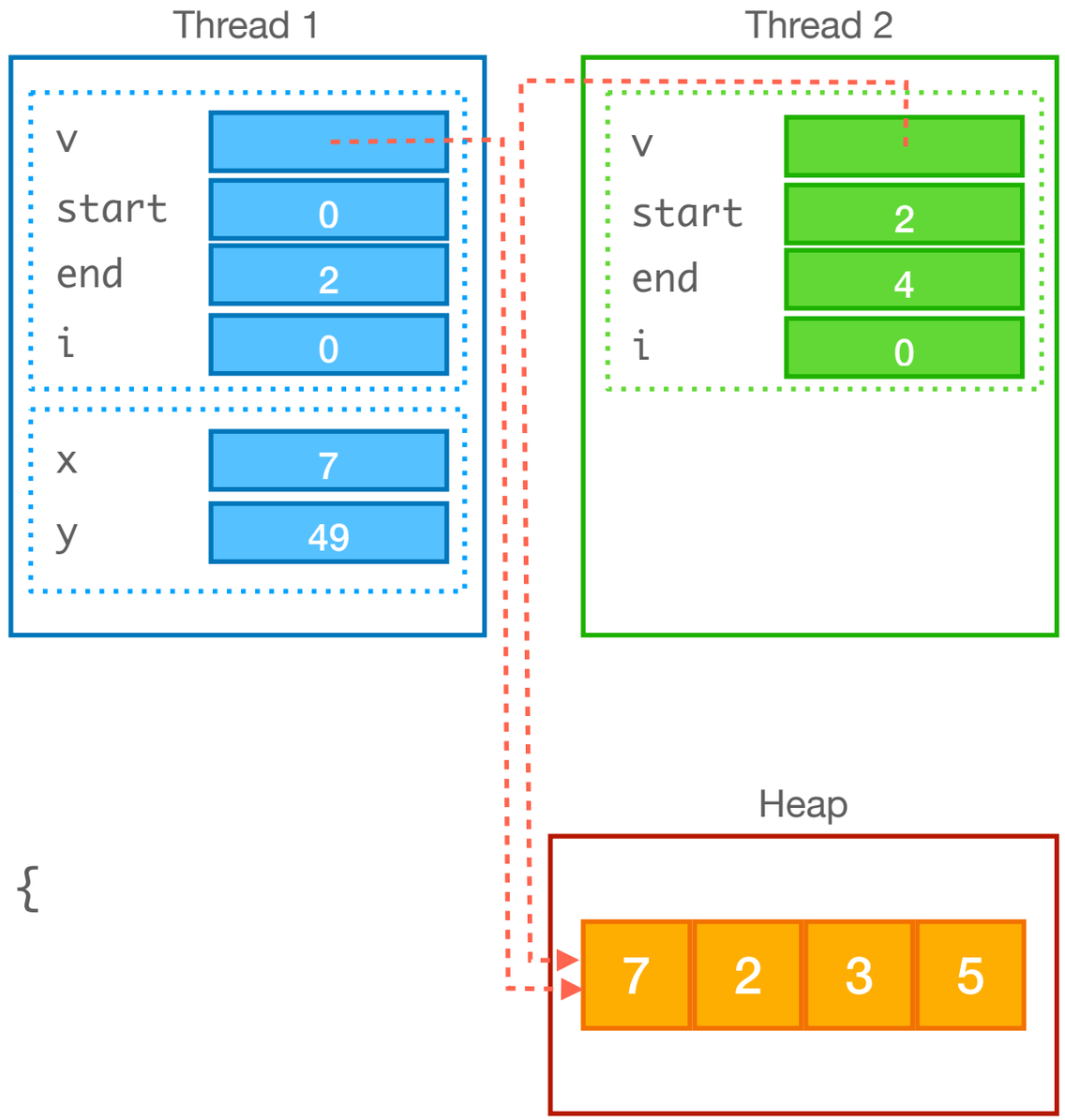
Il thread 1 continua a eseguire

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
    v[i] = g(v[i]);
  }
}
```

PC (thread 1)

PC (thread 2)



Thread: esempio di esecuzione

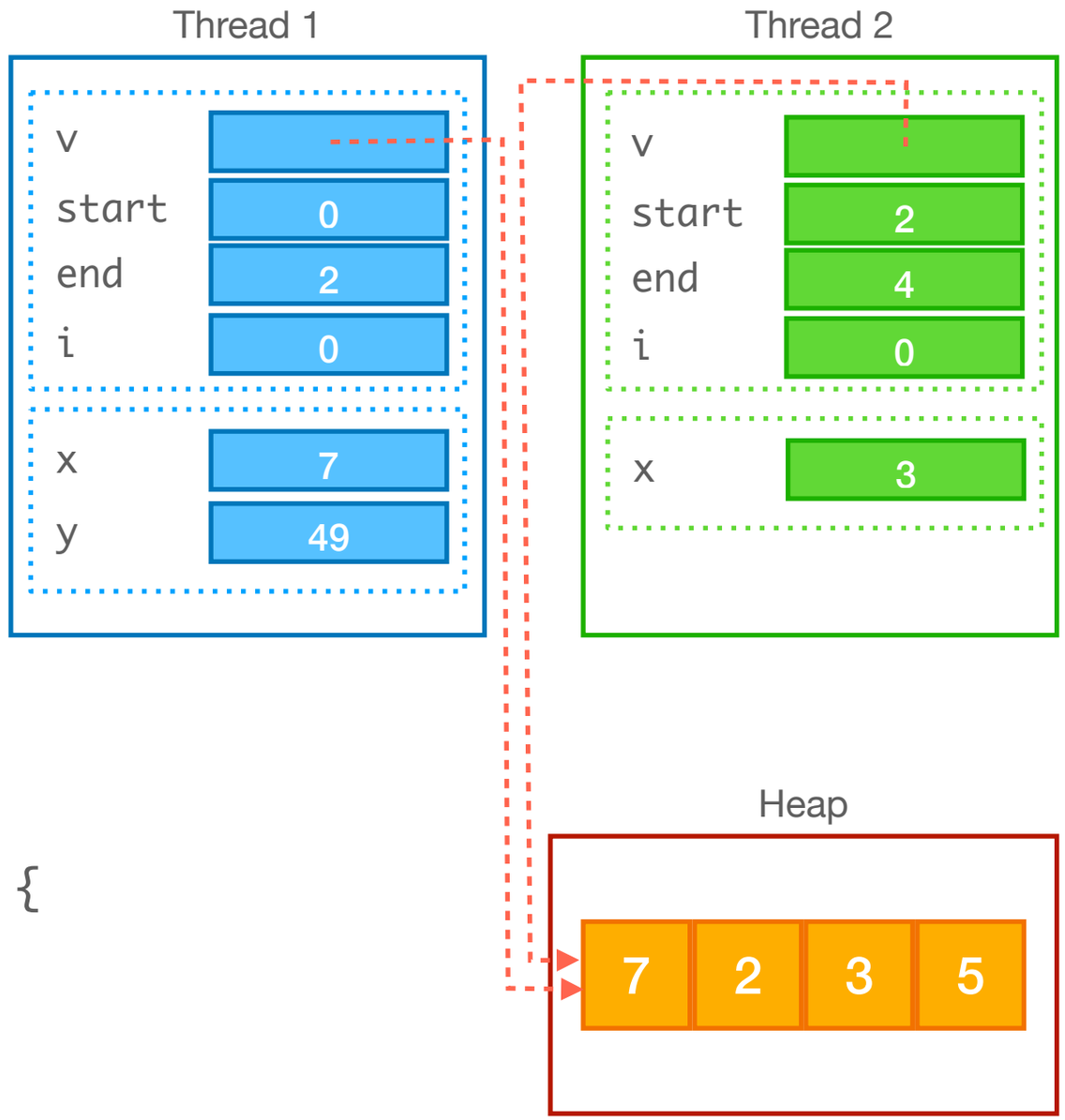
L'esecuzioni continua con il thread 2

PC (thread 2)

PC (thread 1)

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```



Thread: esempio di esecuzione

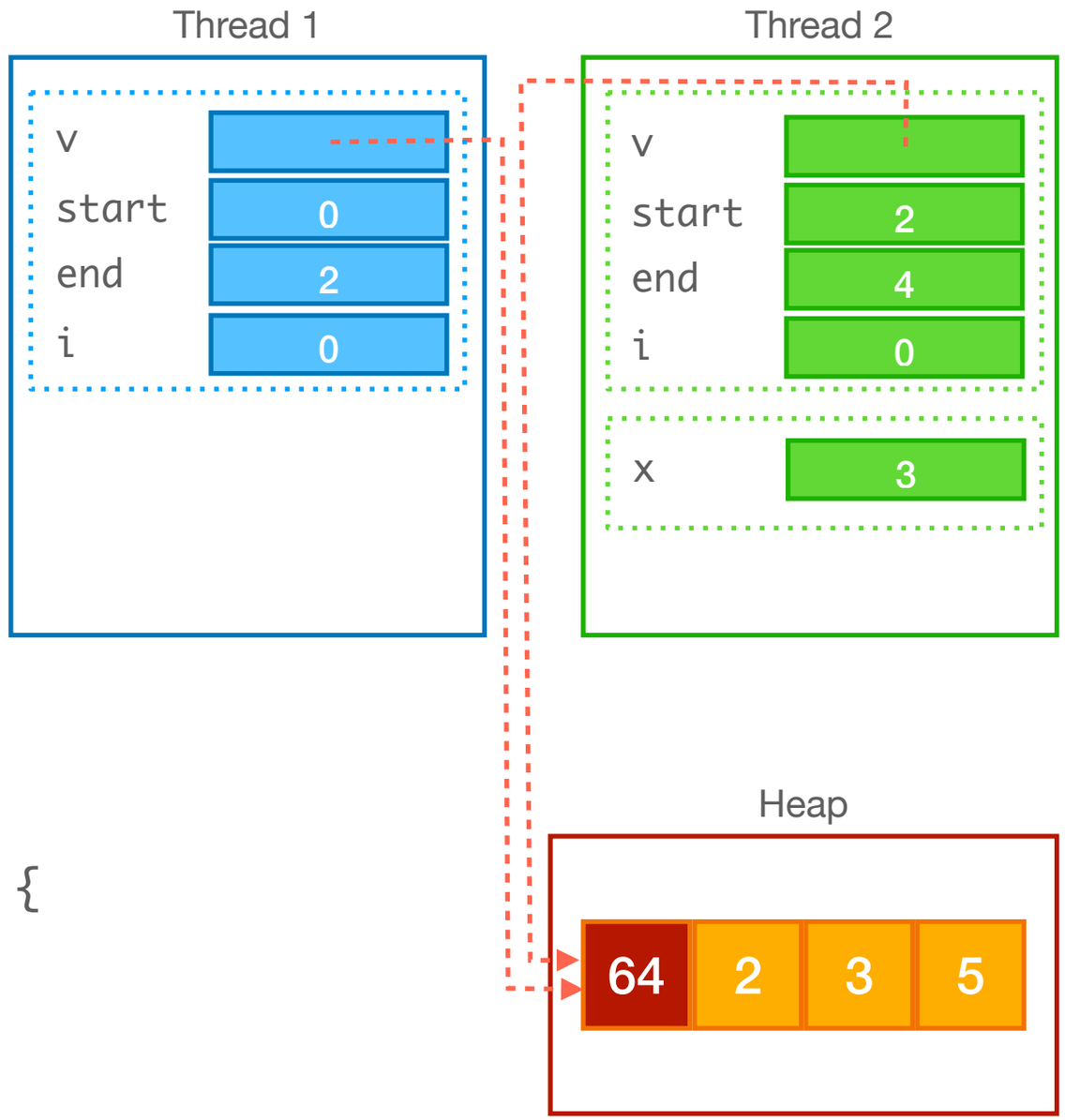
E ora si passa al thread 1

PC (thread 2)

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```

PC (thread 1)



Thread: esempio di esecuzione

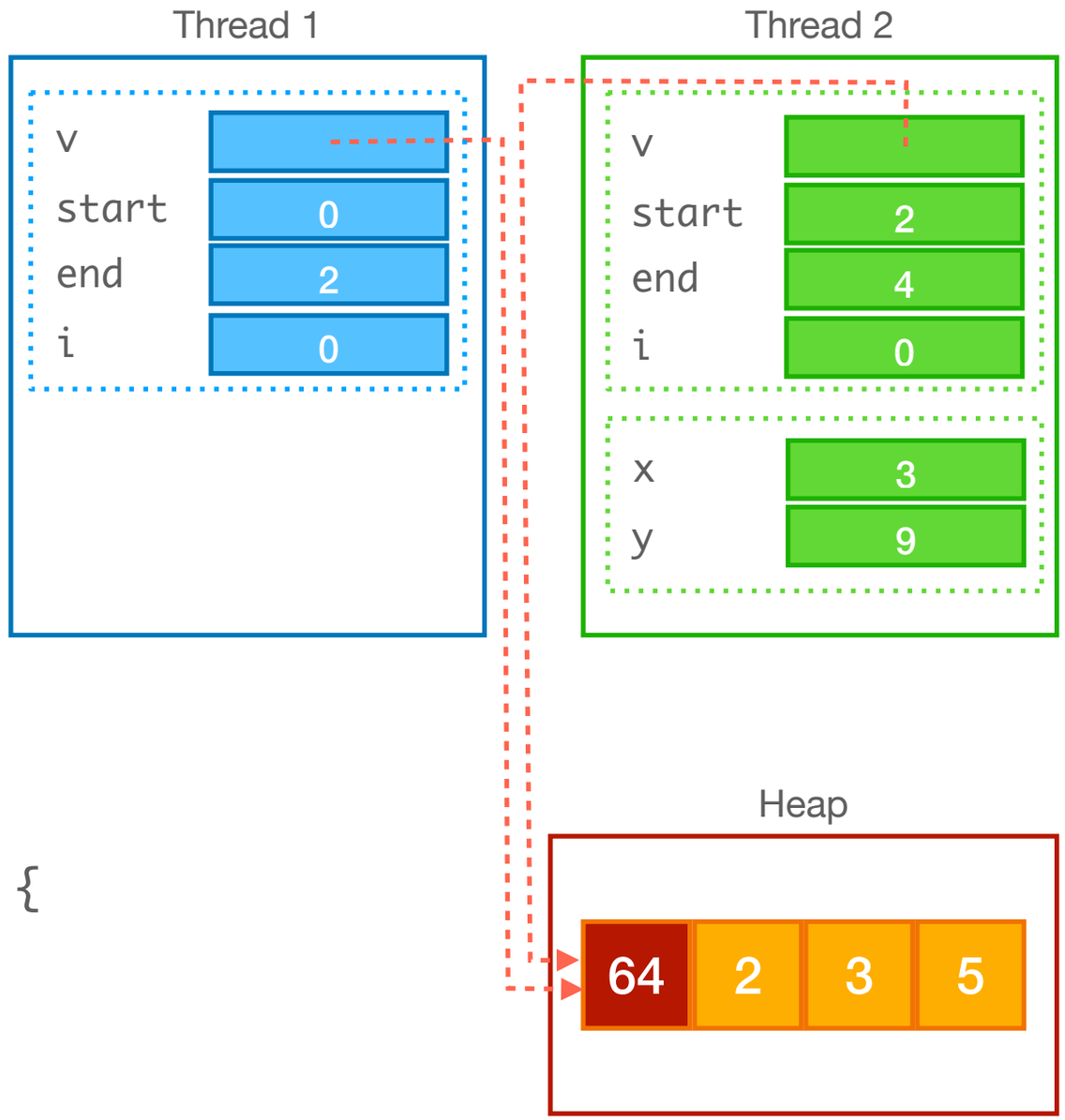
Continua con il thread 2

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```

PC (thread 2)

PC (thread 1)



Thread: esempio di esecuzione

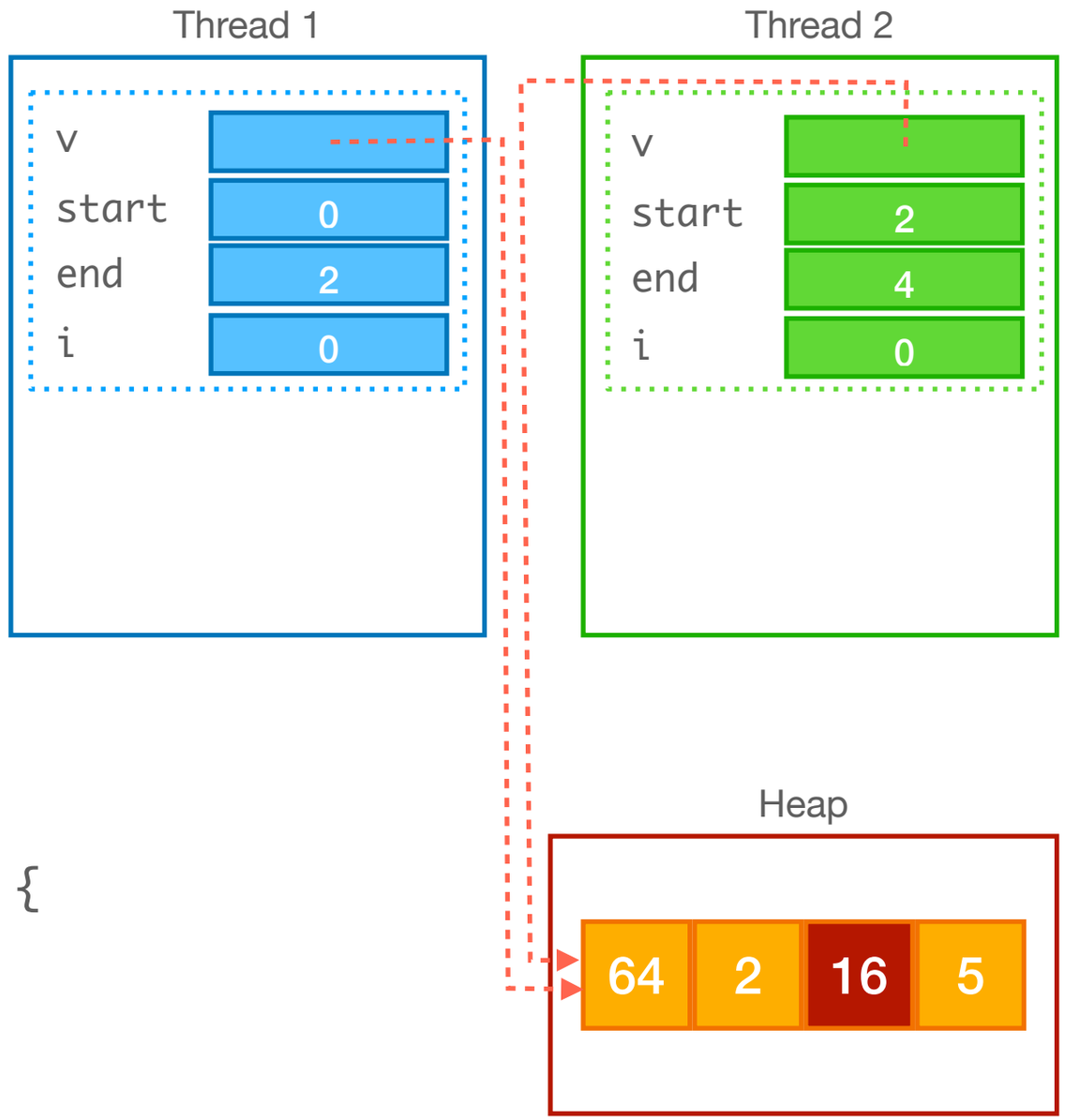
Continua con il thread 2

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```

PC (thread 2)

PC (thread 1)



Thread: esempio di esecuzione

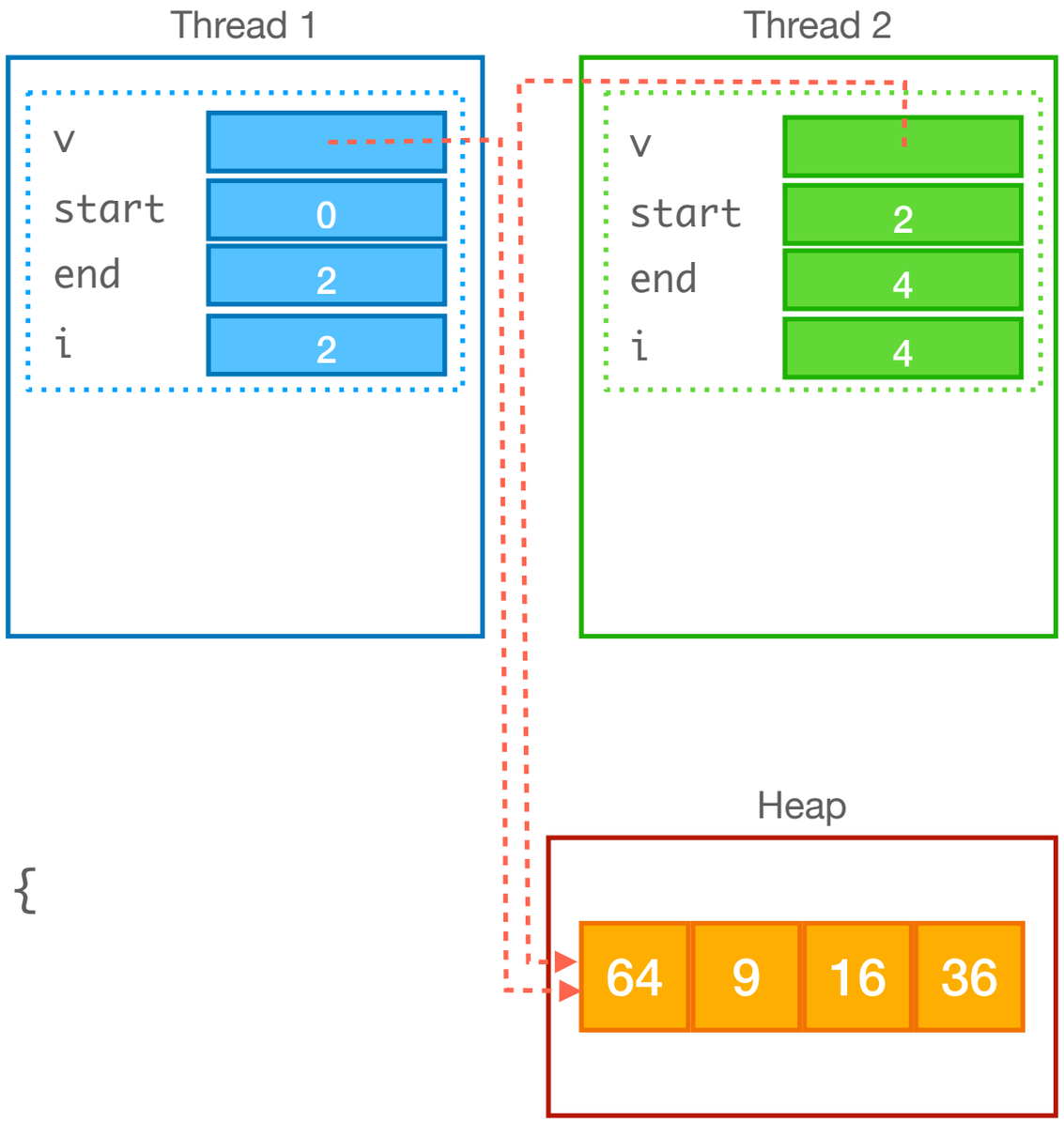
Dopo un certo numero di passi…

```
int g(int x)
{
  int y = x * x;
  return y + 2*x + 1;
}

void f(int * v, int start, int end)
{
  for (int i = start; i < end; i++) {
     v[i] = g(v[i]);
  }
}
```

PC (thread 2)

PC (thread 1)



Thread: commentiamo l'esempio

Esecuzione concorrente

- Notiamo che i due thread possono alternarsi nell'esecuzione, ma l'ordine preciso in cui si alternano dipende dallo scheduler
- Non esiste più un unico modo in cui il programma può eseguire, ma molteplici dato che i thread eseguono in modo indipendente
- Questo vale anche su una macchina con un solo processore, non è necessario avere una esecuzione in contemporanea dei due thread!
- Su una macchina con più processori possono anche agire in parallelo (i.e., i due thread eseguono istruzioni nello stesso momento), potenzialmente modificando i valori nell'array in meno tempo che con un unico thread

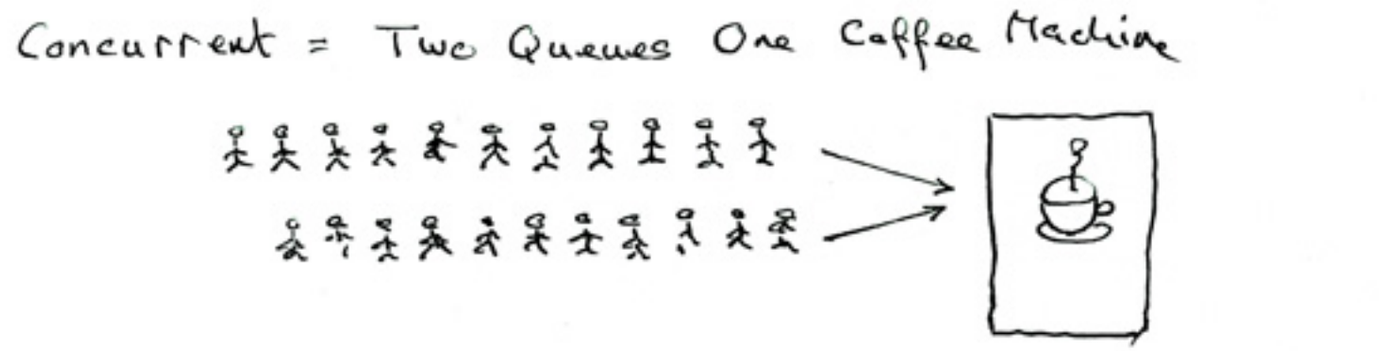
Esecuzione parallela vs concorrente

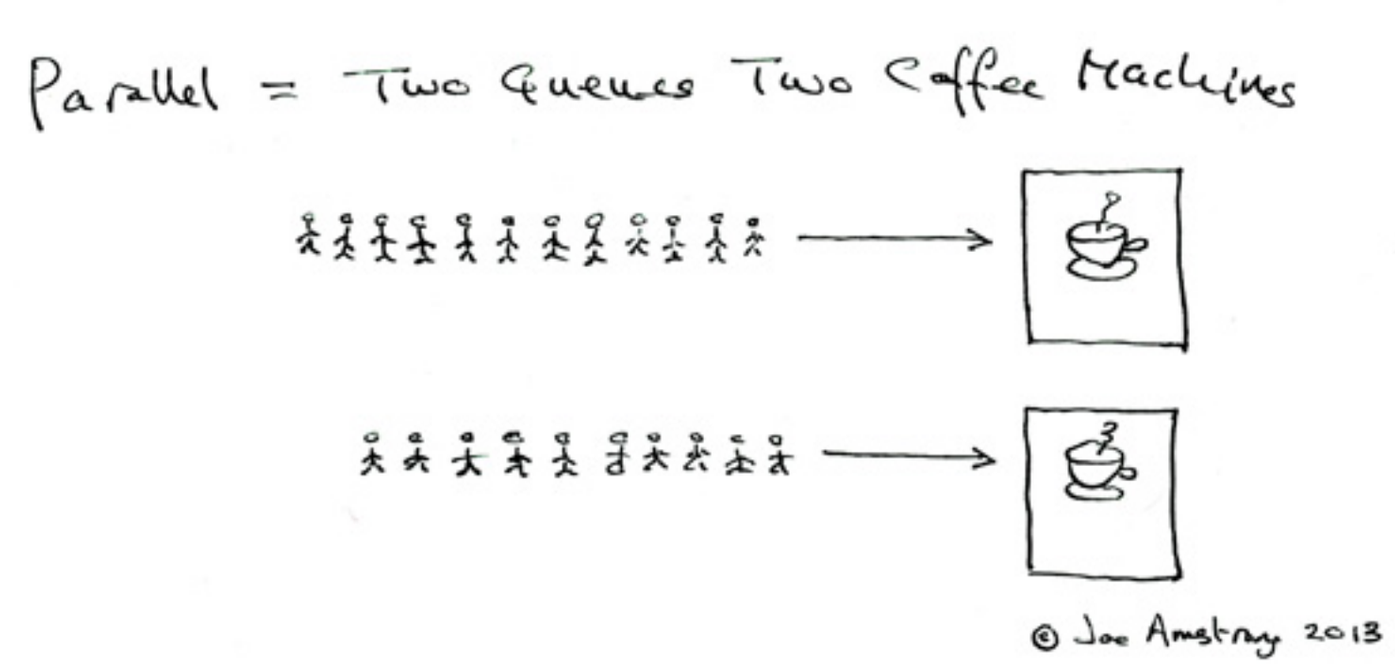
Hint: non sono la stessa cosa!

- Si confonde spesso l'esecuzione parallela rispetto all'esecuzione concorrente
- **Esecuzione concorrente**: due o più computazioni avanzano in modo indipendente l'una dall'altra. Questo può essere compiuto anche in un sistema con un solo processore alternando tra le due computazioni
- **Esecuzione parallela**: due o più computazioni eseguono nello stesso momento. Questo richiede più unità di esecuzione
- Sono due cose distinte: può esserci esecuzione concorrente senza avere nessuna esecuzione parallela

Concorrente vs Parallelo

Rappresentazione grafica





https://joearms.github.io/published/2013-04-05-concurrent-and-parallel-programming.html

pthreads

POSIX Threads

- Esistono diverse librerie per i thread con interfacce diverse
- Nel mondo unix uno standard sono i POSIX thread
- Per lavorare coi pthread includiamo il file header **pthread.h**
- Questa libreria ci permetto di creare e sincronizzare thread
- Ma anche permette di utilizzare delle primitive di sincronizzazione (lock/mutex, condition variable), che ci permettono di evitare problemi (quali problemi? Vedremo a breve…)

Creazione di un thread

pthread_create

Puntatore a struttura di tipo pthread_t.
Verrà popolata da pthread_create e la useremo per interagire con il thread

Attributi aggiuntivi del thread (e.g., dimensione dello stack). Solitamente NULL per indicare di usare i valori di default

```
int pthread_create(pthread_t       *thread,
                   const pthread_attr_t *attr,
                   void            *(*start_routine)(void*),
                   void            *arg);
```

Puntatore agli argomenti da passare alla funzione da chiamare quando il thread viene fatto partire (i.e., il precedente argomento)

Funzione con cui far partire il thread. Questa notazione indica un *puntatore a funzione*, ma ci basta sapere che un valore valido è il nome della funzione da chiamare.
Questa funzione deve ritornare *void* * e ricevere un *void* * come argomento

Perché **void** *?
questo tipo significa che passiamo un puntatore a “qualcosa” o ritorniamo un puntatore a “qualcosa”. Sarà compito della funzione chiamata fare i casting necessari per trasformare il “qualcosa” (void *) in un tipo specifico (e.g., int *, un puntatore a una struct)

Attendere la terminazione di un thread

pthread_join

Così come possiamo attendere che un processo termini, possiamo anche attendere che un thread termini.

L’equivalente nei POSIX thread è:

```
int pthread_join(pthread_t thread,
                 const void **   value_prt);
```

Thread di cui aspettare la terminazione

Puntatore a un puntatore void *. Vogliamo indicare un puntatore a un’area di memoria in cui salvare un puntatore a void * che è il valore di ritorno della funzione chiamata all’avvio del thread. NULL nel caso non si sia interessati a questo valore

Creazione di un thread

Esempio

```
#include <pthread.h>
#include <stdio.h>

void * f(void * arg)
{
    int * x = (int *) arg;
    printf("%d\n", *x);
    return NULL;
}

int main(int argc, char * argv[])
{
    pthread_t thread;
    int a = 10;
    pthread_create(&thread, NULL, f, &a);
    pthread_join(thread, NULL);
    return 0;
}
```

Funzione chiamata all'avvio del thread

Creazione del thread
viene chiamata f con argomento &a
come prima funzione dopo la creazione

Attendiamo che il thread appena creato
termini prima di proseguire

Mostrare l’esecuzione concorrente

Due thread che stampano

- Nell’esempio precedente non è chiaro che il thread principale e quello avviato agiscano davvero in modo concorrente
- Possiamo rendere evidente l’esecuzione concorrente usando due thread che stampano a schermo caratteri diversi e vedere come i caratteri in output si alternano
- Per esempio:

  ____.-……-.-.--------..--..-.--------.--------..-.---
  ____..-.--------..-.--------.-.........-.....................
  ....-.-------
- Perché questo ci mostra che l’esecuzione è concorrente?

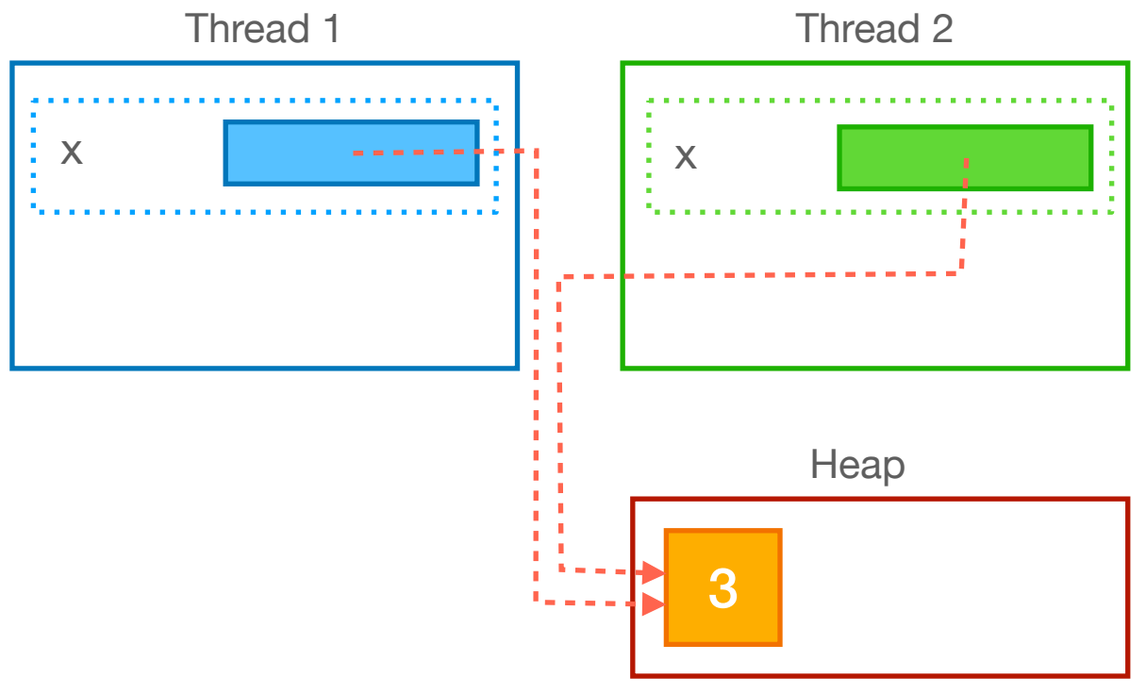
Thread: esempio di race condition

Quando il risultato dipende dall'ordine di esecuzione

PC (thread 2)

PC (thread 1)

```
void f(int * x)
{
    int y = *x;
    *x = y + 1;
    printf("%d\n", *x);
}
```



Cosa stampa questo codice?

a) 4
   4

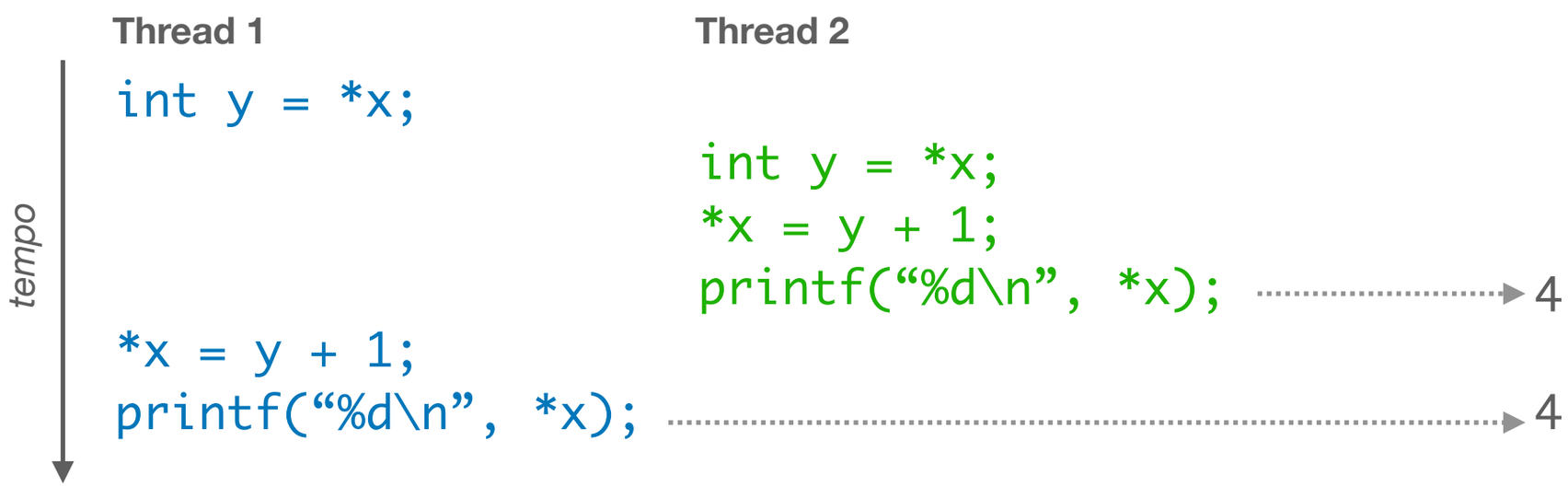
b) 4
   5

c) 5
   5

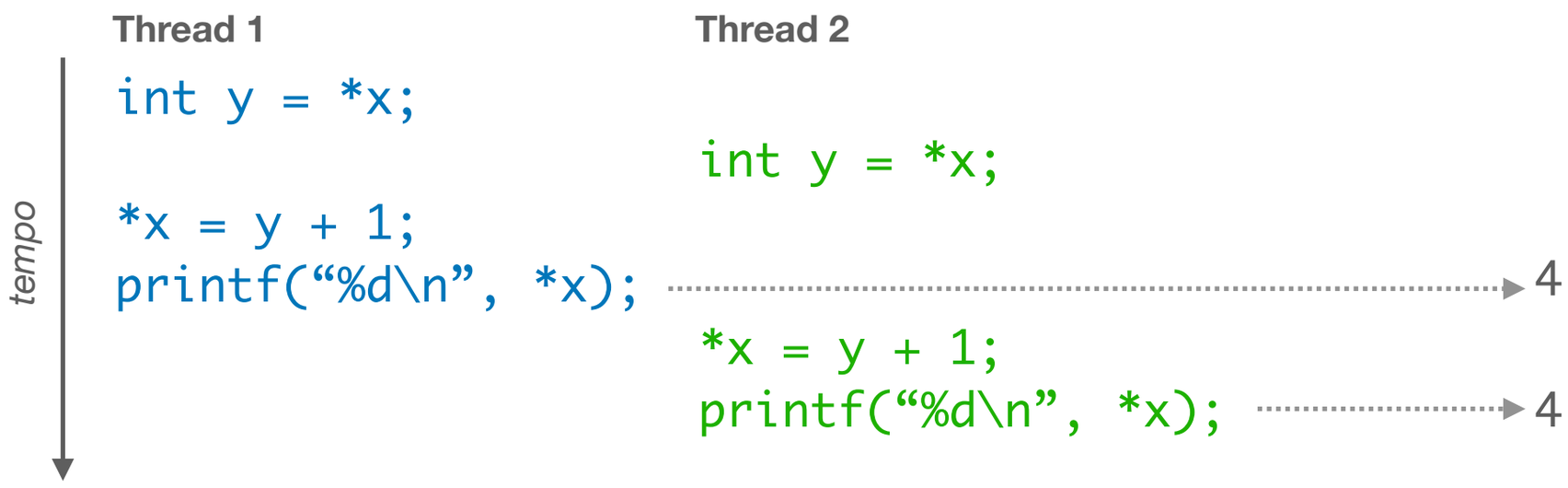
Tutte queste sono risposte valide a seconda dell'ordine in cui eseguono le istruzioni all'interno dei thread!

Thread: esempio di race condition

Opzione a) “4 4”

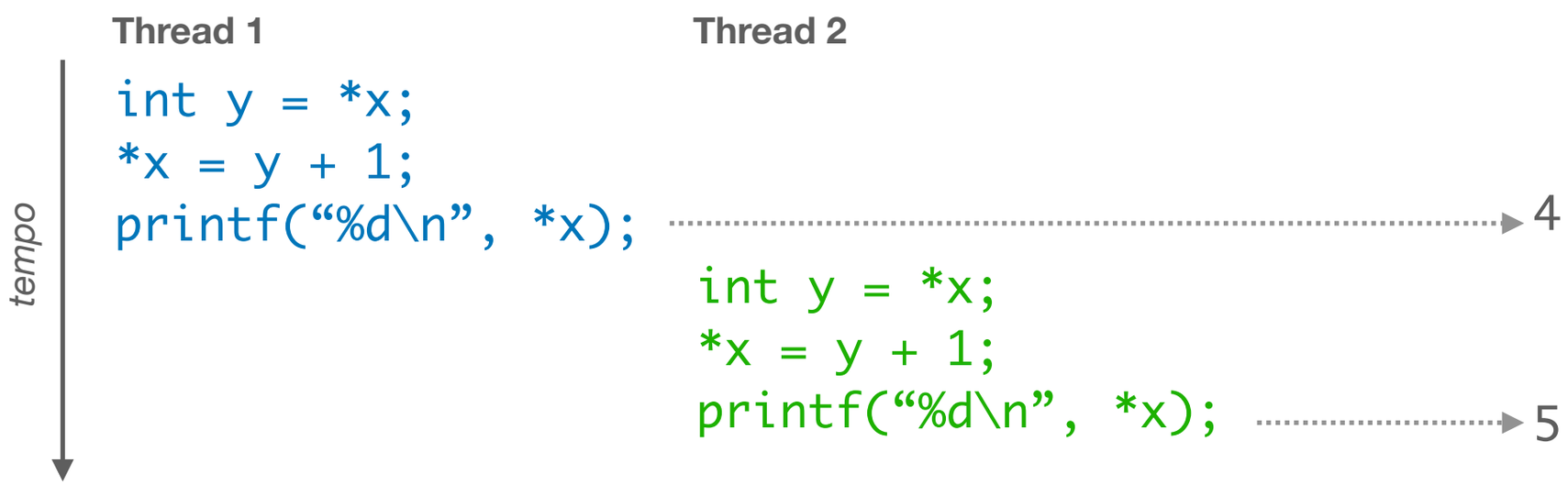


Ma non è l’unico caso in cui stampiamo due volte lo stesso valore

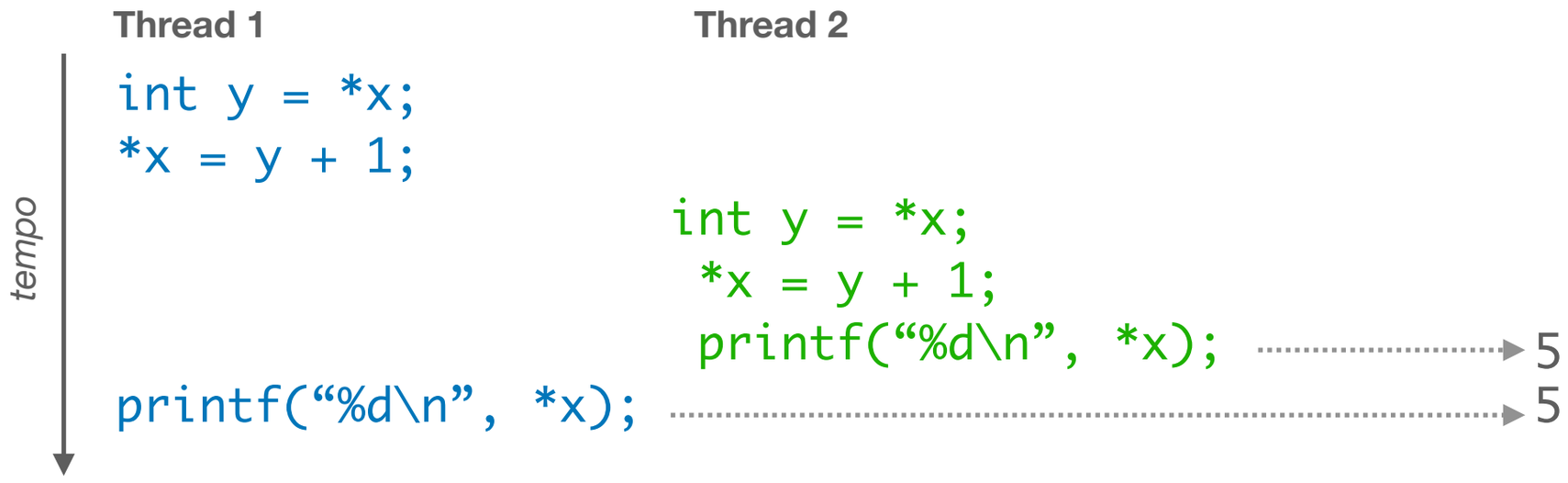


Thread: esempio di race condition

Opzione b) “4 5”



Opzione c) “5 5”



Race conditions

Cosa sono?

- Il nostro programma invece di avere un comportamento deterministico ha un comportamento non deterministico

- Ci sono alcune sequenze di operazioni che se interrotte possono provocare un comportamento non voluto

- Questo perché due o più thread stanno accedendo alla stessa risorsa condivisa (in questo caso una variabile) nello stesso momento

- Il risultato finale dipende da una “corsa” tra i thread, questo è solitamente chiamata una **race condition**

- In particolare abbiamo una race condition quando il risultato dipende dall’ordine di esecuzione delle operazioni

Sezioni critiche

A cosa servono

- Quello che vogliamo per evitare race condition è forzare un serie di operazioni ad essere eseguite da un solo thread alla volta

- Per esempio ogni sequenza di operazioni che riguarda la variabile X deve essere eseguita da un solo thread alla volta

- Vogliamo quindi inibire l'esecuzione concorrente di alcune parti del codice

```
void f(int * x)
{
  int y = *x;
  *x = y + 1;
  printf("%d\n", *x);
}
```

Al massimo un thread può essere "qui dentro" in ogni dato momento

Sezione critica
una parte del programma in cui vi è accesso a risorse condivise e in cui l'esecuzione concorrente deve essere inibita

Lock 

Cosa è un lock

Controllare l’accesso alle sezioni critiche

- I lock (o mutex) sono uno strumento per evitare l’accesso concorrente alle sezioni critiche
- Un lock viene acquisito prima di entrare in una sezione critica e rilasciato al termine della sezione critica
- Solo un thread può “possedere” un lock in un dato momento, quindi tutte le parti di codice protette dallo stesso lock sono tali per cui al massimo un thread le sta eseguendo
- Lock diversi possono proteggere risorse diverse, è anzi una buona idea avere una certa granularità nei lock

Lock: evitare le race condition

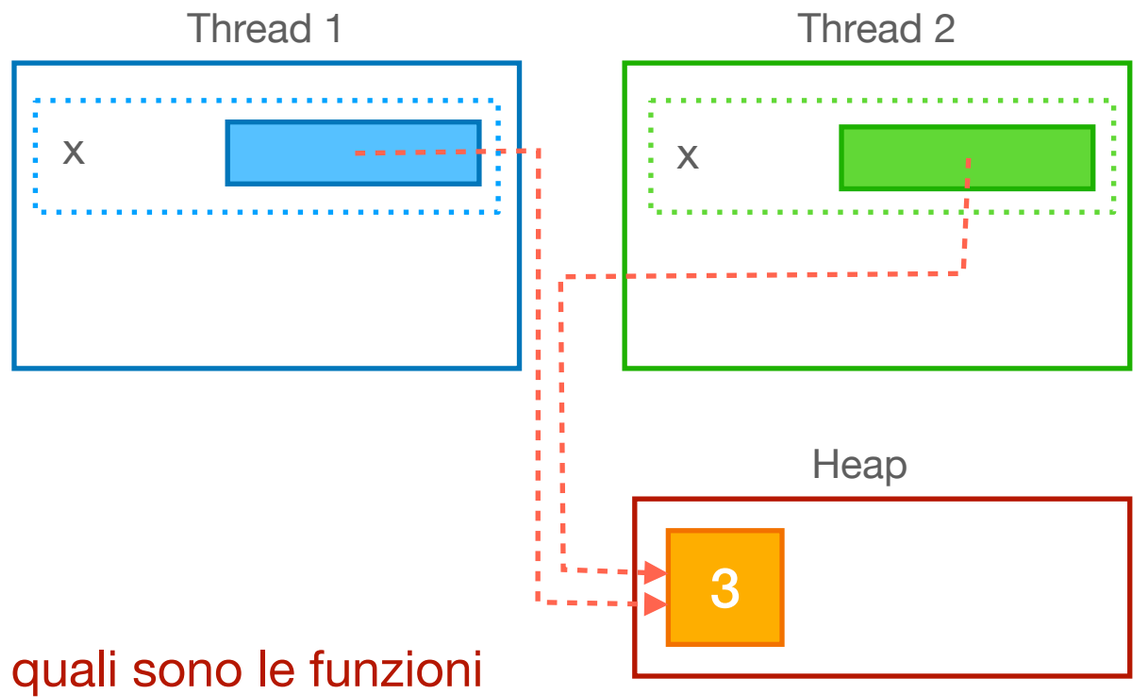
Aggiungiamo un lock al codice precedente

PC (thread 2)

PC (thread 1)

```
void f(int * x)
{
  acquisisci_lock();
  int y = *x;
  *x = y + 1;
  printf(“%d\n”, *x);
  rilascia_lock();
}
```

Vediamo dopo quali sono le funzioni corrette per acquisire e rilasciare i lock



Cosa stampa questo codice?

Solo una possibilità: 4
5

Lock: evitare le race condition

Possibili sequenze di istruzioni eseguite

tempo

Thread 1

```
acquisici_lock();
int y = *x;
*x = y + 1;
printf(“%d\n”, *x);
rilascia_lock();
```

Thread 2

```
acquisisci_lock();
int y = *x;
*x = y + 1;
printf(“%d\n”, *x);
rilascia_lock();
```

4

5

tempo

Thread 1

```
acqusisci_lock();
int y = *x;
*x = y + 1;
```

Thread 2

```
acquisisci_lock();
int y = *x;
*x = y + 1;
```

Impossibile, il thread 2 non può acquisire il lock fino a quando il thread 1 non lo ha rilasciato

Proprietà dei lock

Affinché un lock possa chiamarsi tale

- Vogliamo che un lock rispetti una serie di proprietà:
  - **Mutua esclusione**. La loro funzionalità di base. Serve che la parte di codice protetta dal lock possa venir eseguita da un solo thread per volta
  - **Fairness**. Si vuole che tutti i thread abbiano possibilità di acquisire il lock e che un thread non rimanga per sempre in attesa di acquisirlo
  - **Performance**. Acquisire e rilasciare i lock deve essere una operazione veloce. Non essenziale ma importante per ogni utilizzo pratico

Lock nei pthread

Come creare e lavorare coi lock in C e unix

- I lock sono strutture di tipo `pthread_mutex_t`
- Un lock **deve** venire inizializzato prima di essere usato tramite `PTHREAD_MUTEX_INITIALIZER` o usando la funzione `pthread_mutex_init`
- Un lock viene acquisito con `pthread_mutex_lock(pthread_mutex_t * mutex)`
- E viene rilasciato con `pthread_mutex_unlock(pthread_mutex_t * mutex)`

I lock risolvono tutto?

No, ci sono altri problemi

- Sebbene i lock siano molto utili per la programmazione concorrente, ci sono diversi problemi:

- I lock inibiscono l'esecuzione concorrente di codice, se tutto è protetto da un unico "grande" lock perdiamo alcuni dei vantaggi dell'esecuzione concorrente (e parallela, se ci sono le risorse)

- Quando più di un lock deve essere acquisito (e.g., lock per variabile A e poi lock per variabile B) possono crearsi delle situazioni in cui dei thread non possono proseguire l'esecuzione:

  - Per esempio,
    il Thread 1 ha il lock per A e vuole acquisire quello per B
    il Thread 2 ha il lock per B e vuole acquisire quello per A

- Vedremo queste situazioni (dette "deadlock") nelle prossime lezioni