departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

NOVALINCS
LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS

Monitoring Concurrency Errors

In the Quest for Deadlocks and Atomicity Violations

ARVI Winter School — 2018-03-21

Joao Lourenço <joao.lourenco@fct.unl.pt>

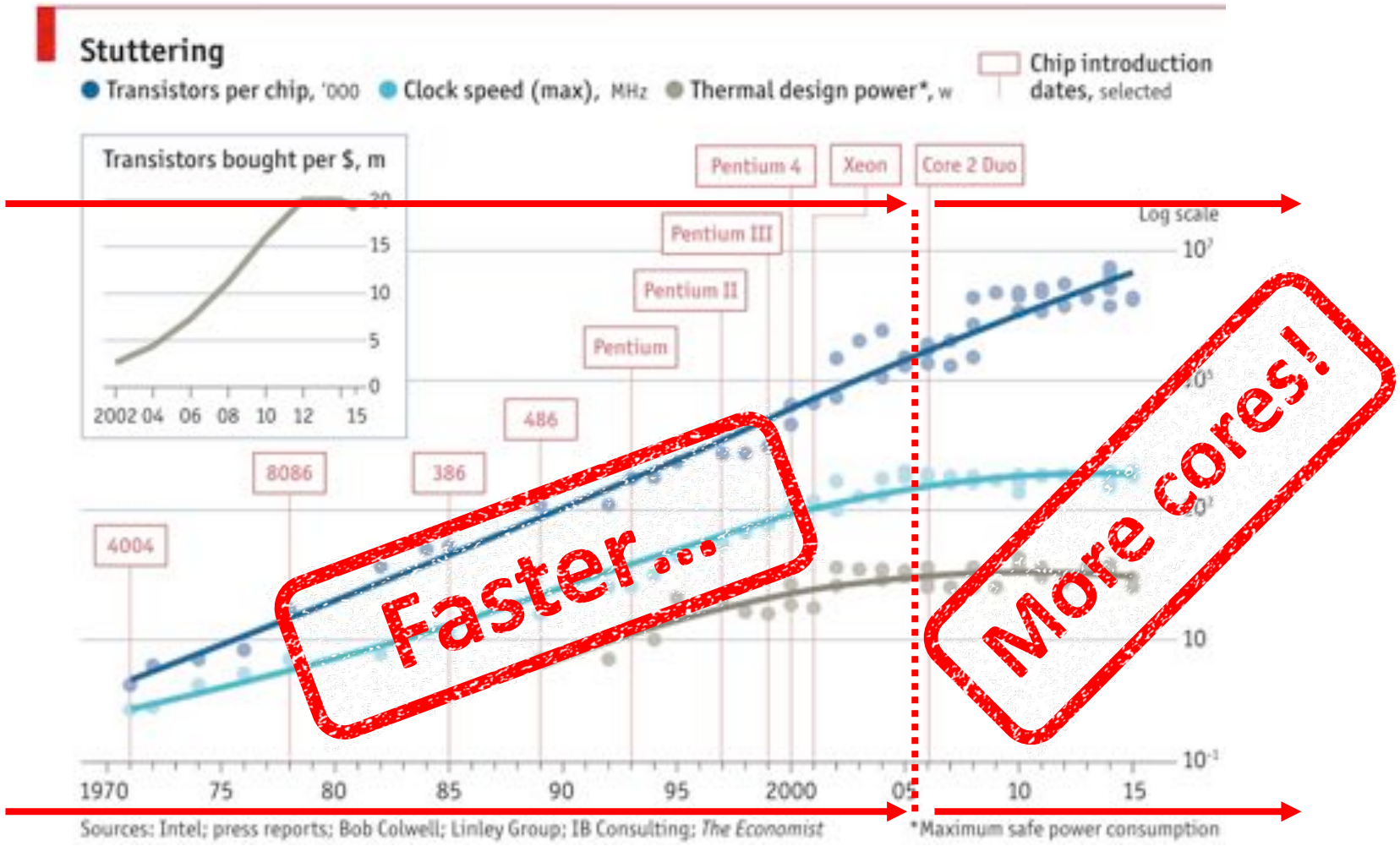
Agenda

- Why are we here?
- Concurrency Anomalies
- Assigning Semantics to Concurrent Programs
- Concurrency Errors
  - Detection of data races
  - Detection of high-level data races
  - Detection of deadlocks

Why are we here?

It is Moore's fault! ;)

Moore's Law


Stuttering
Transistors per chip, '000
Clock speed (max), MHz
Thermal design power*, w
Chip introduction dates, selected
Transistors bought per $, m
2002 04 06 08 10 12 15
Pentium 4
Xeon
Core 2 Duo
Pentium III
Pentium II
Pentium
486
386
8086
4004
Log scale
1970 75 80 85 90 95 2000 05 10 15
Faster...
More cores!
Sources: Intel; press reports; Bob Colwell; Linley Group; IB Consulting; The Economist
*Maximum safe power consumption

Is there a problem…

With having more cores?

We need multiple cooperating processes!

But… Is there a problem…

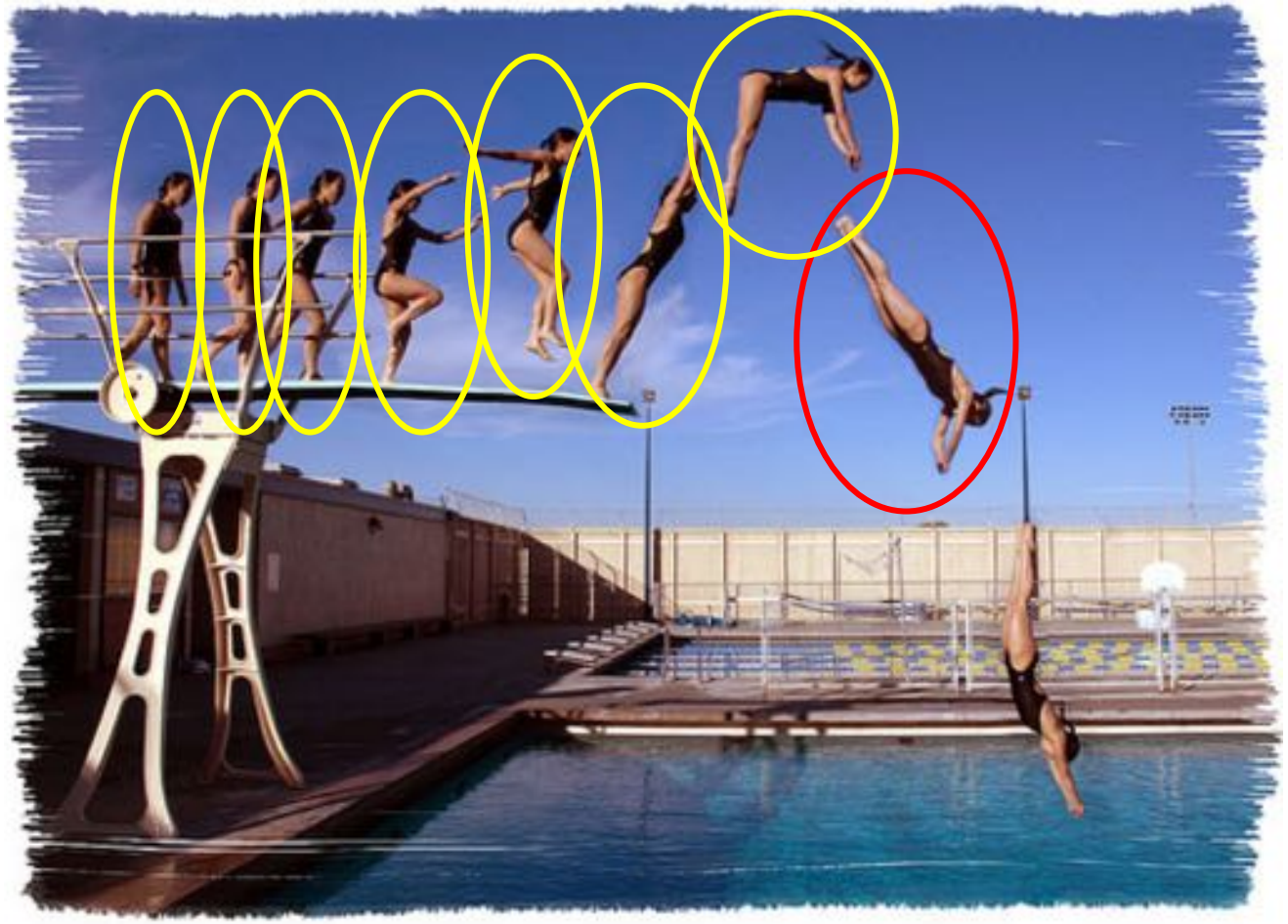
With using multiple cooperating processes?

But… Is there a problem?

Concurrency Anomalies

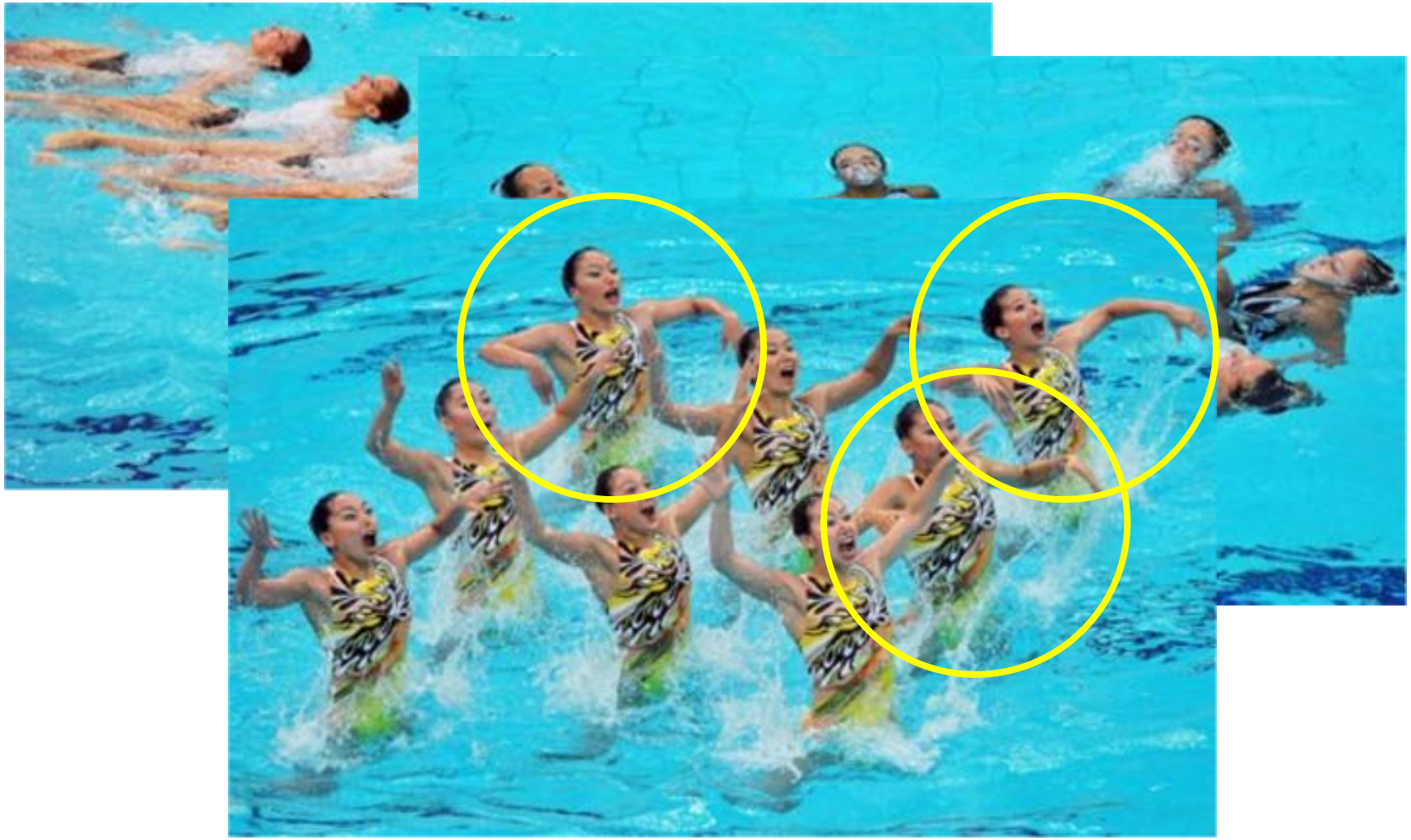
What and why?

Single Process

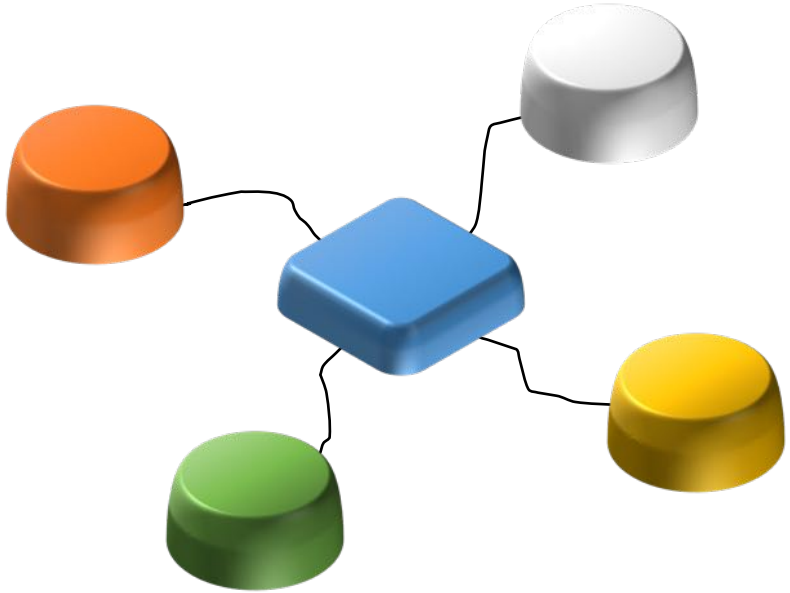


Single Process

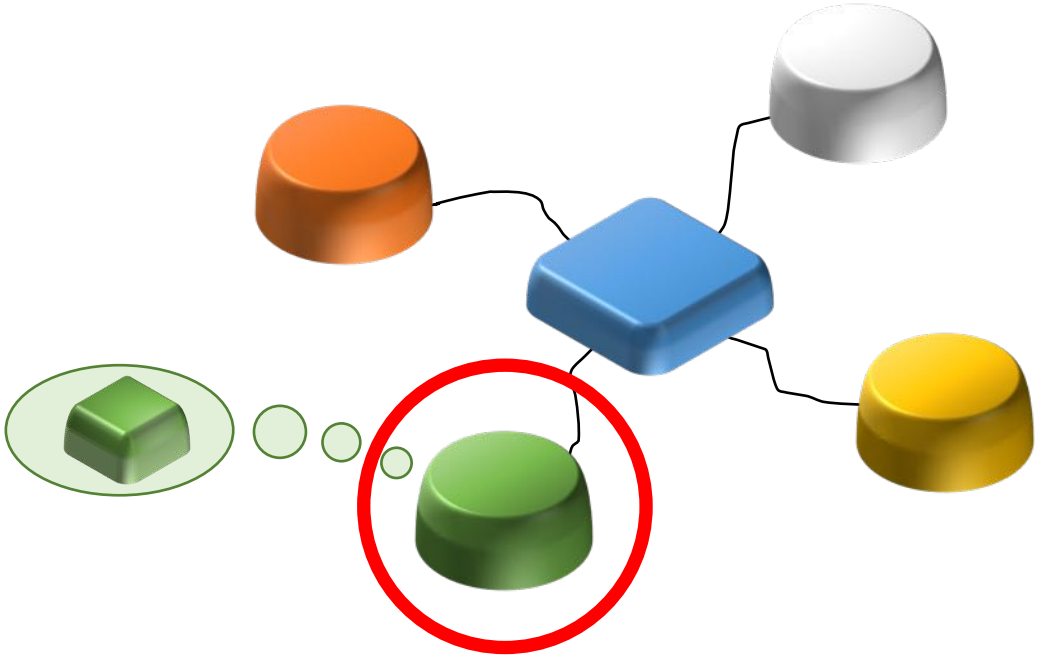
Multiple Processes



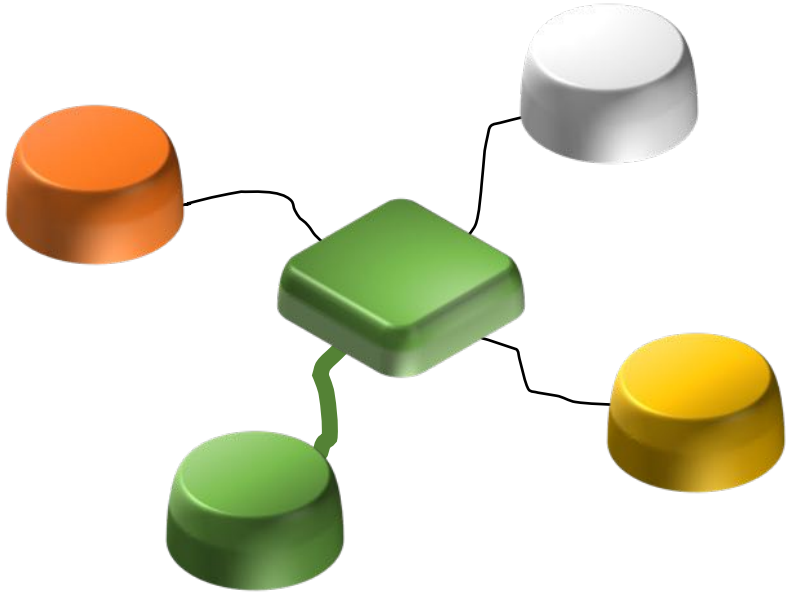
Sharing Resources (unsync.)



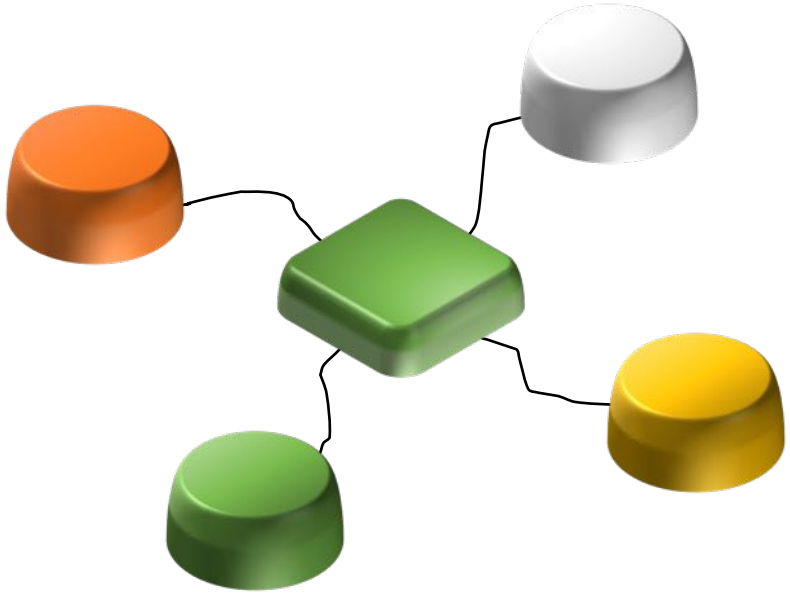
Sharing Resources (unsync.)



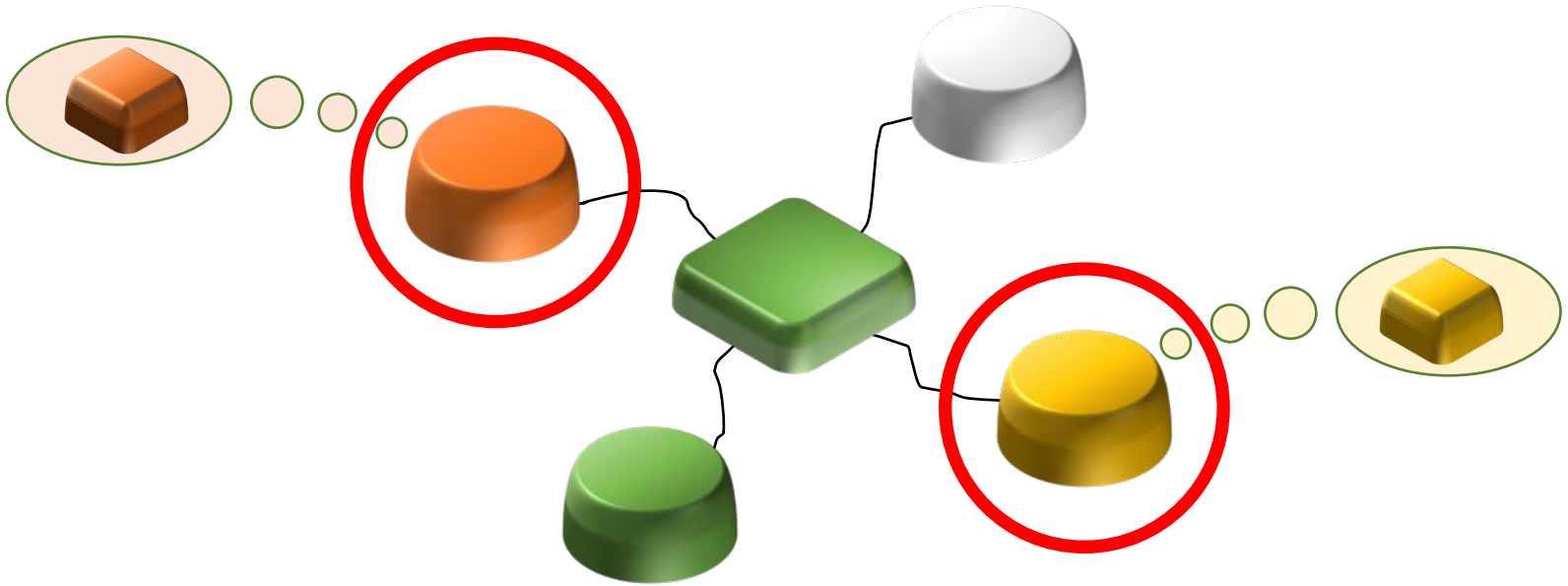
Sharing Resources (unsync.)



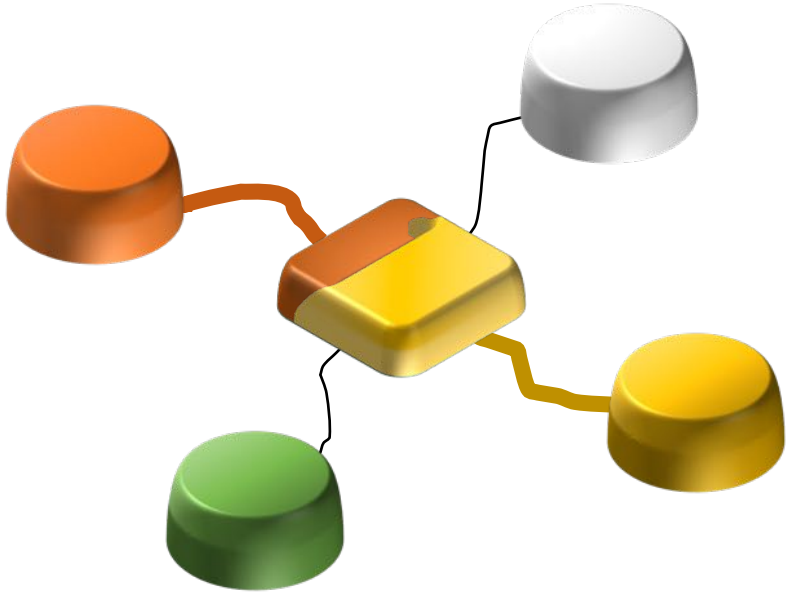
Sharing Resources (unsync.)



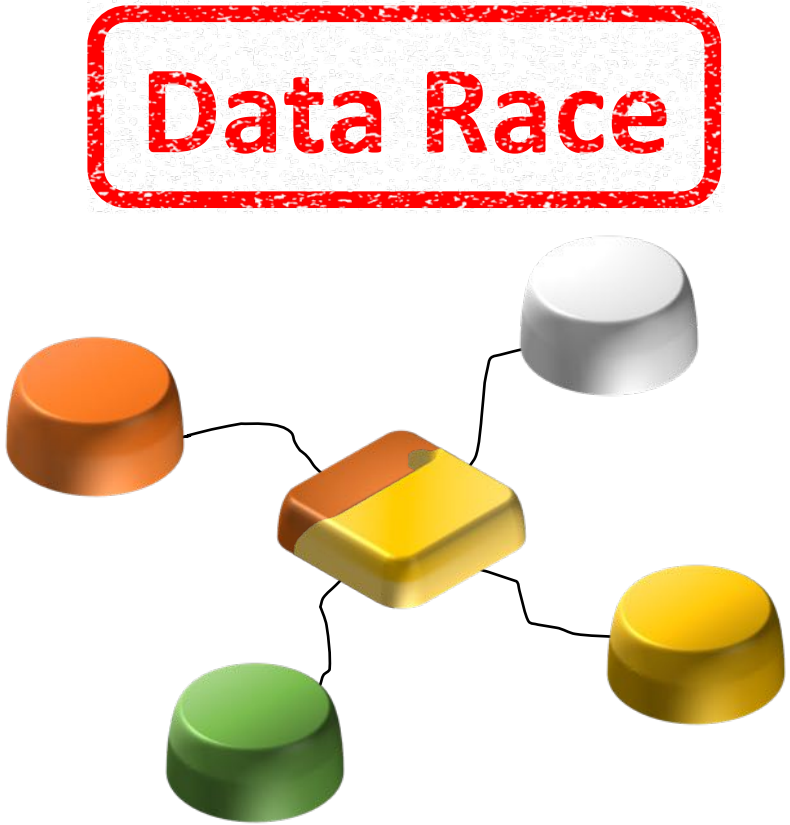
Sharing Resources (unsync.)



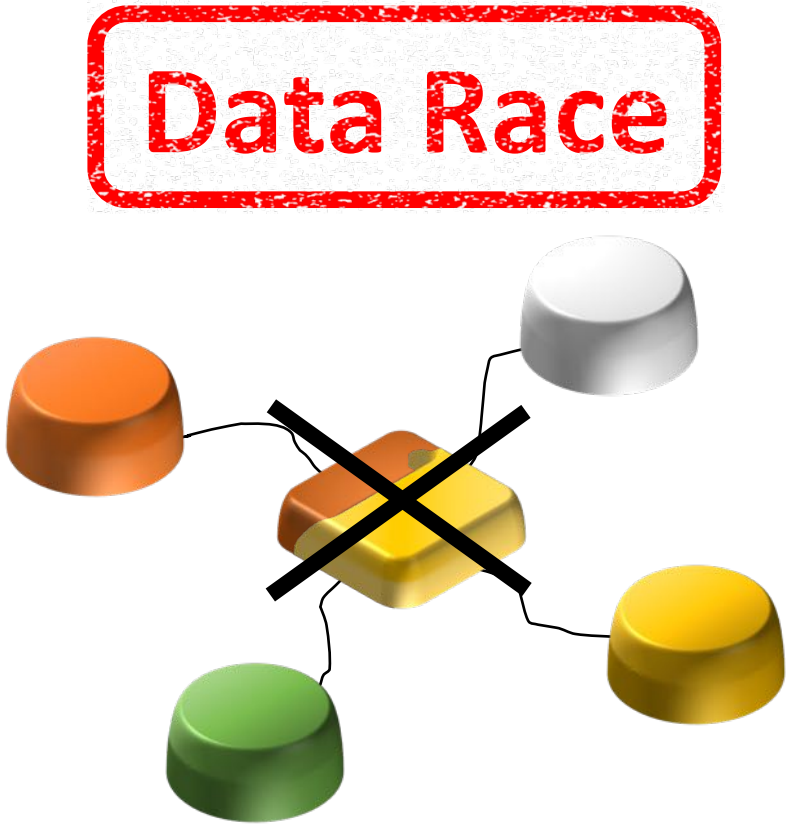
Sharing Resources (unsync.)



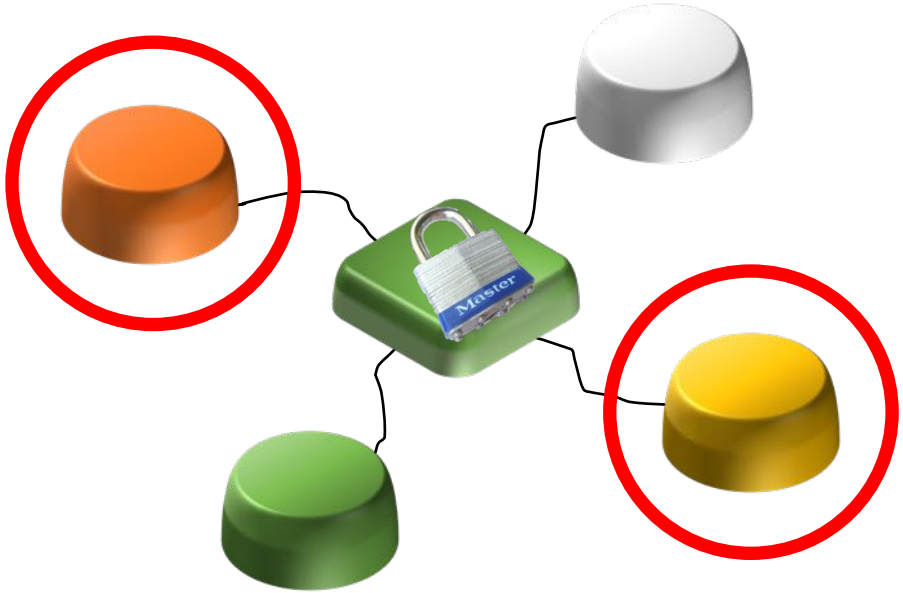
Sharing Resources (unsync.)



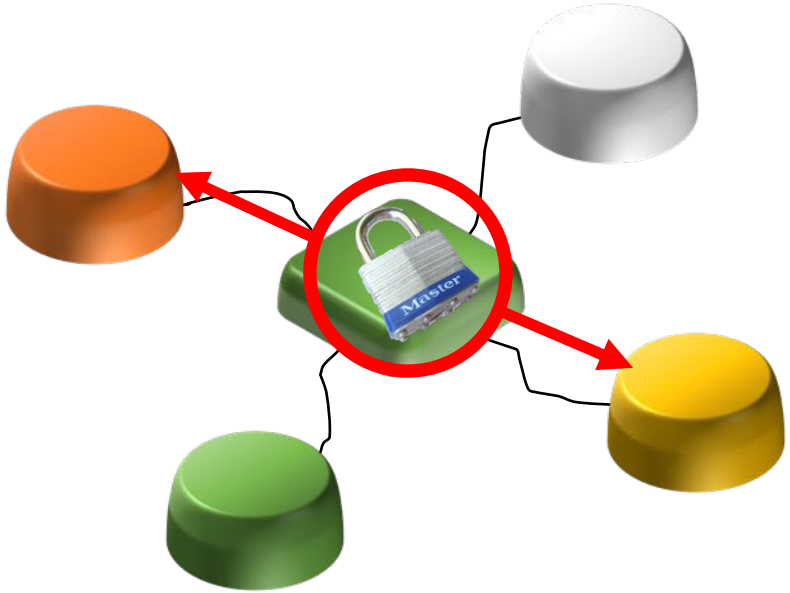
How to avoid Data Races?



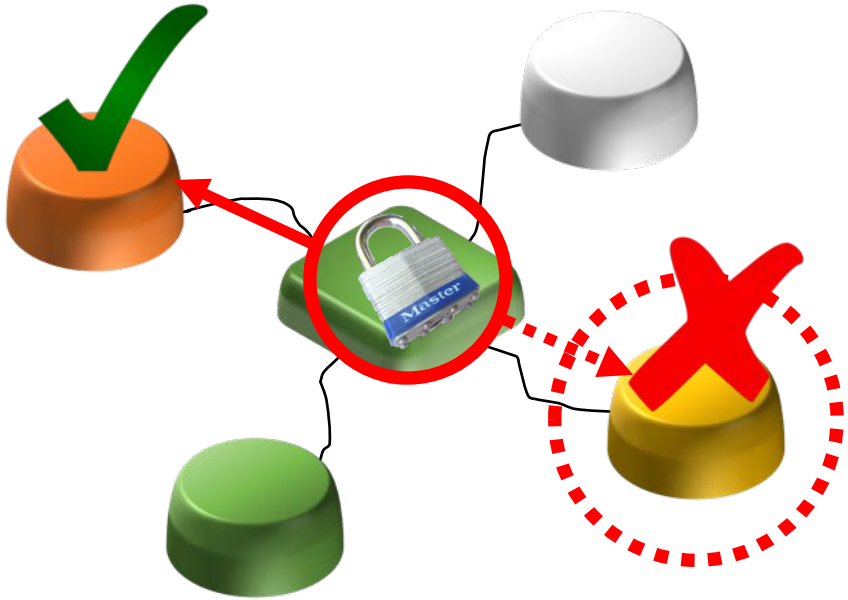
Sharing Resources (sync.)



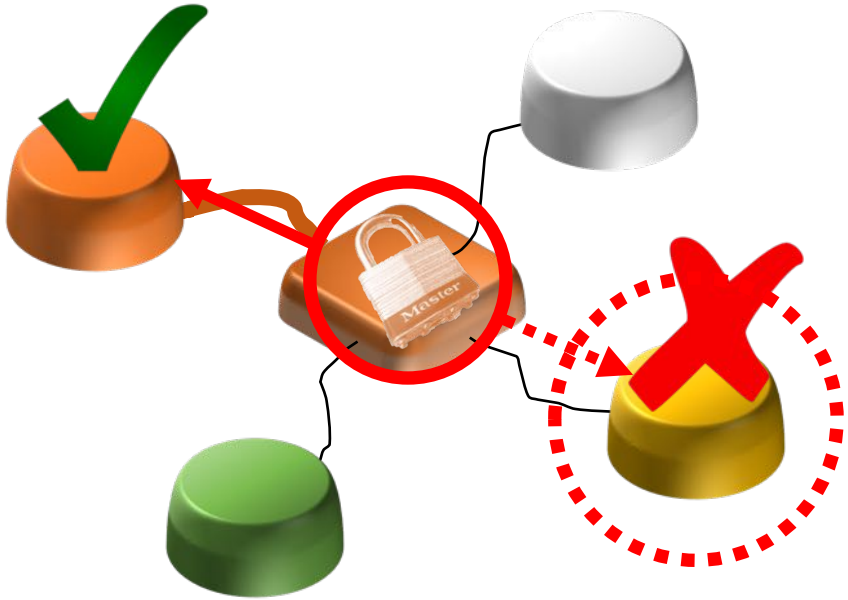
Sharing Resources (sync.)



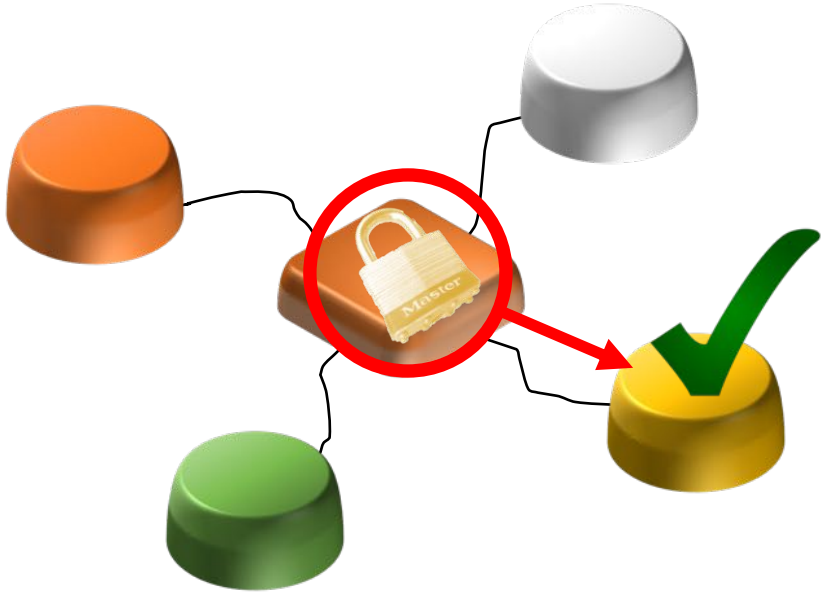
Sharing Resources (sync.)



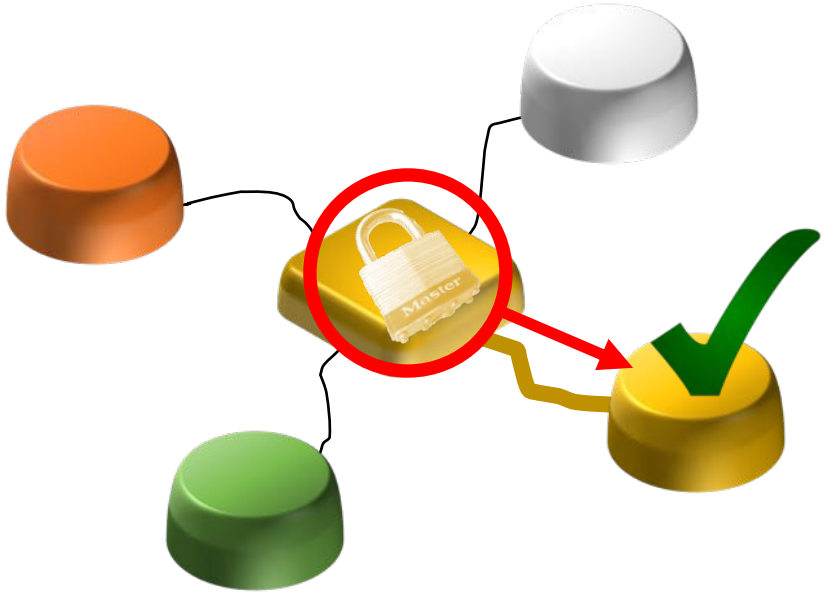
Sharing Resources (sync.)



Sharing Resources (sync.)

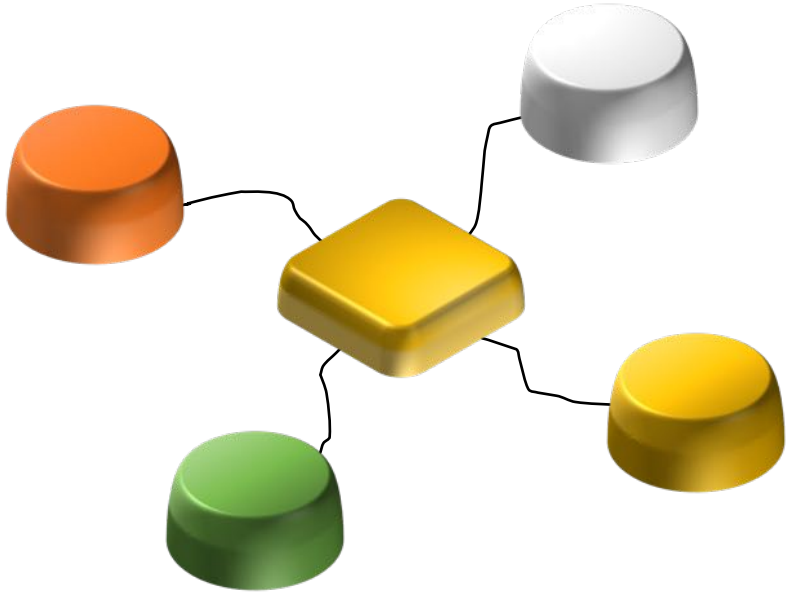


Sharing Resources (sync.)

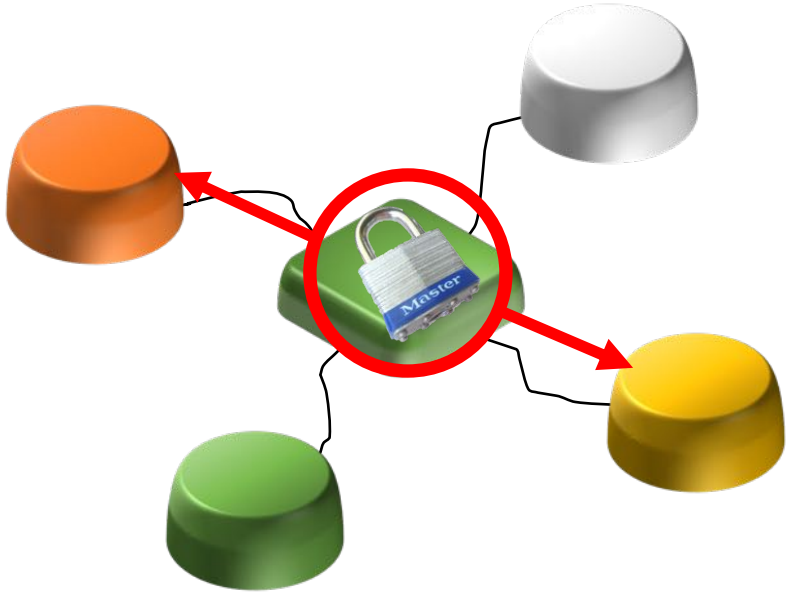


Sharing Resources (sync.)

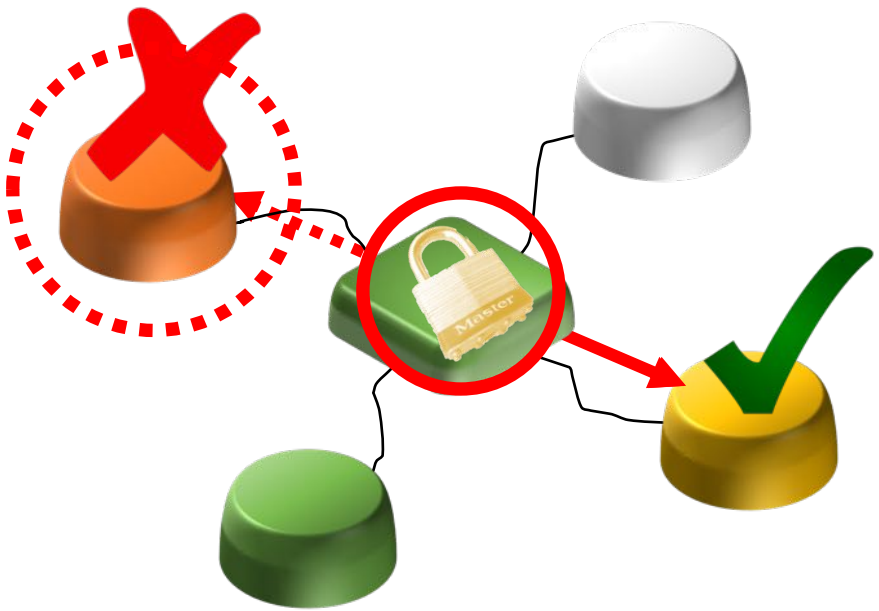
Sequential Consistency



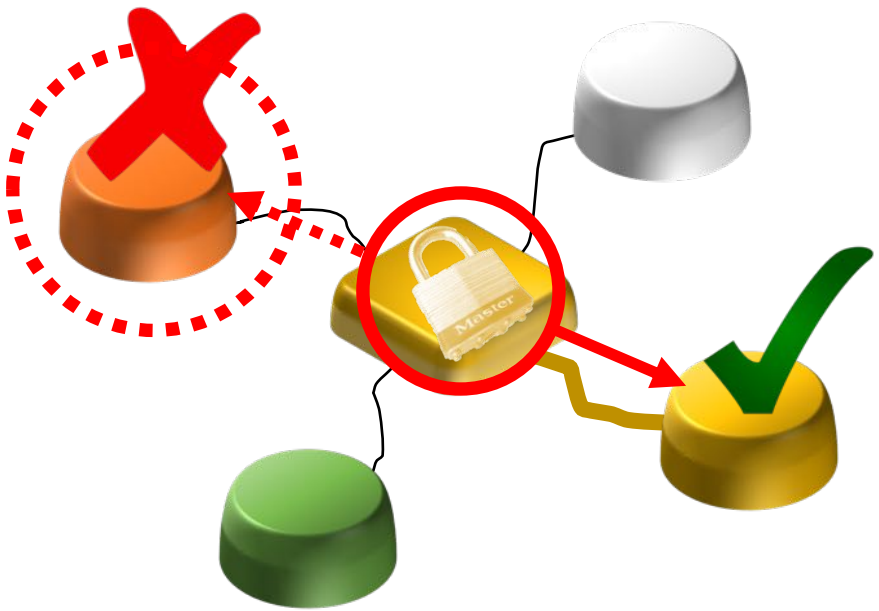
Sharing Resources (sync.)



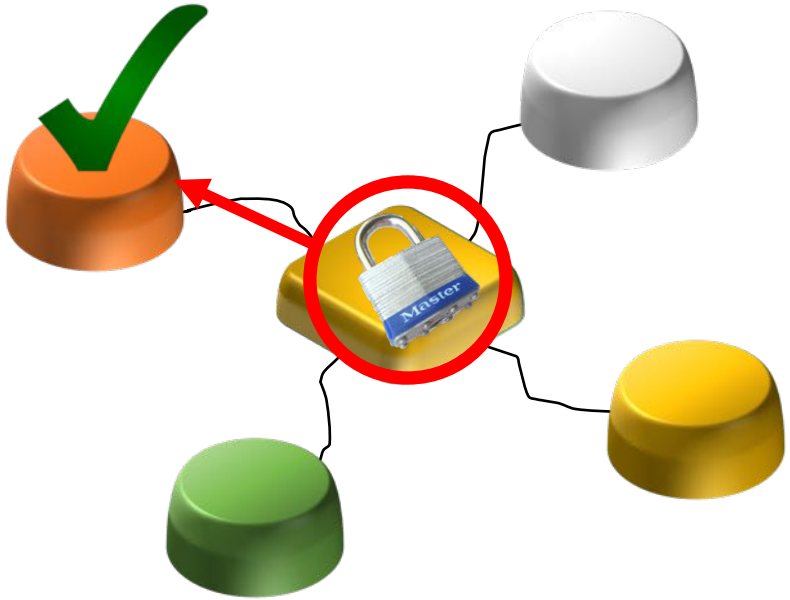
Sharing Resources (sync.)



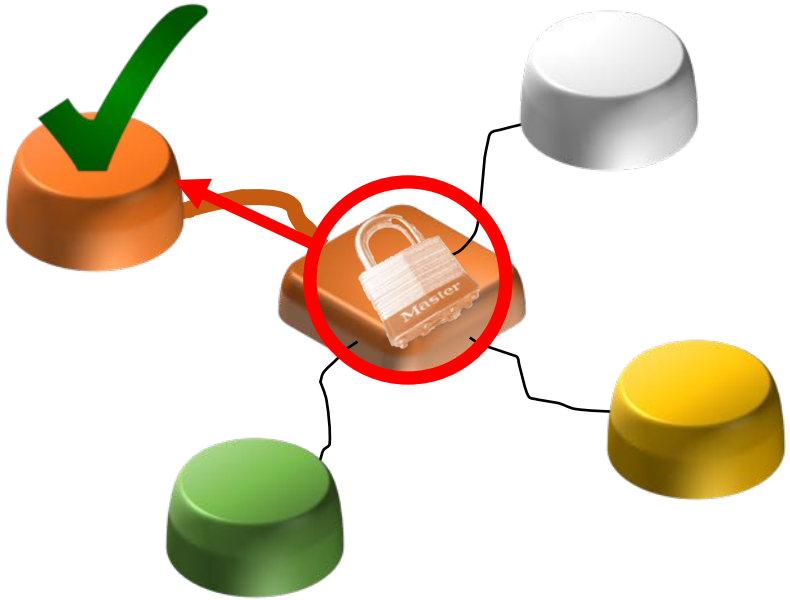
Sharing Resources (sync.)



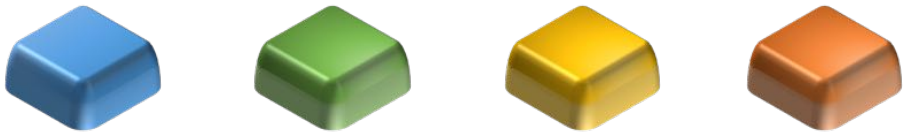
Sharing Resources (sync.)



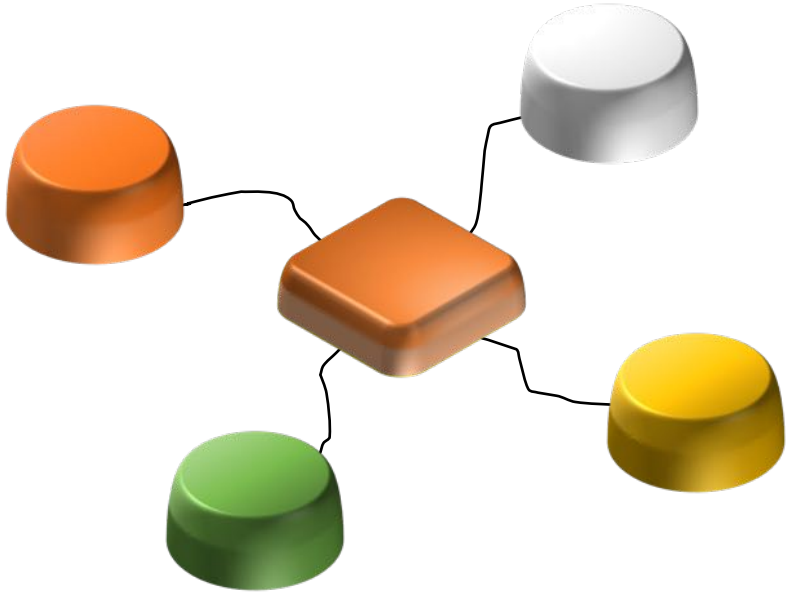
Sharing Resources (sync.)



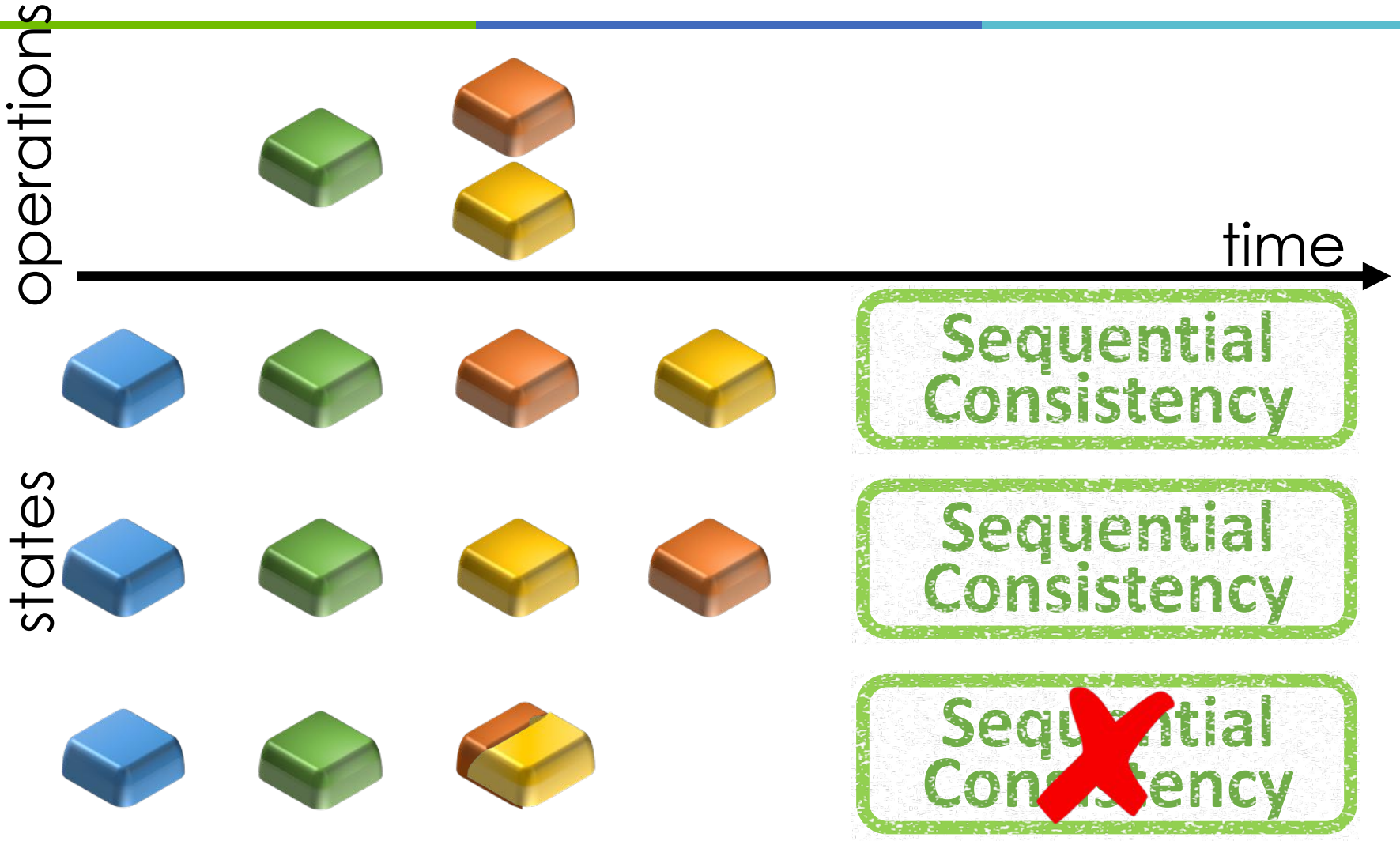
Sharing Resources (sync.)



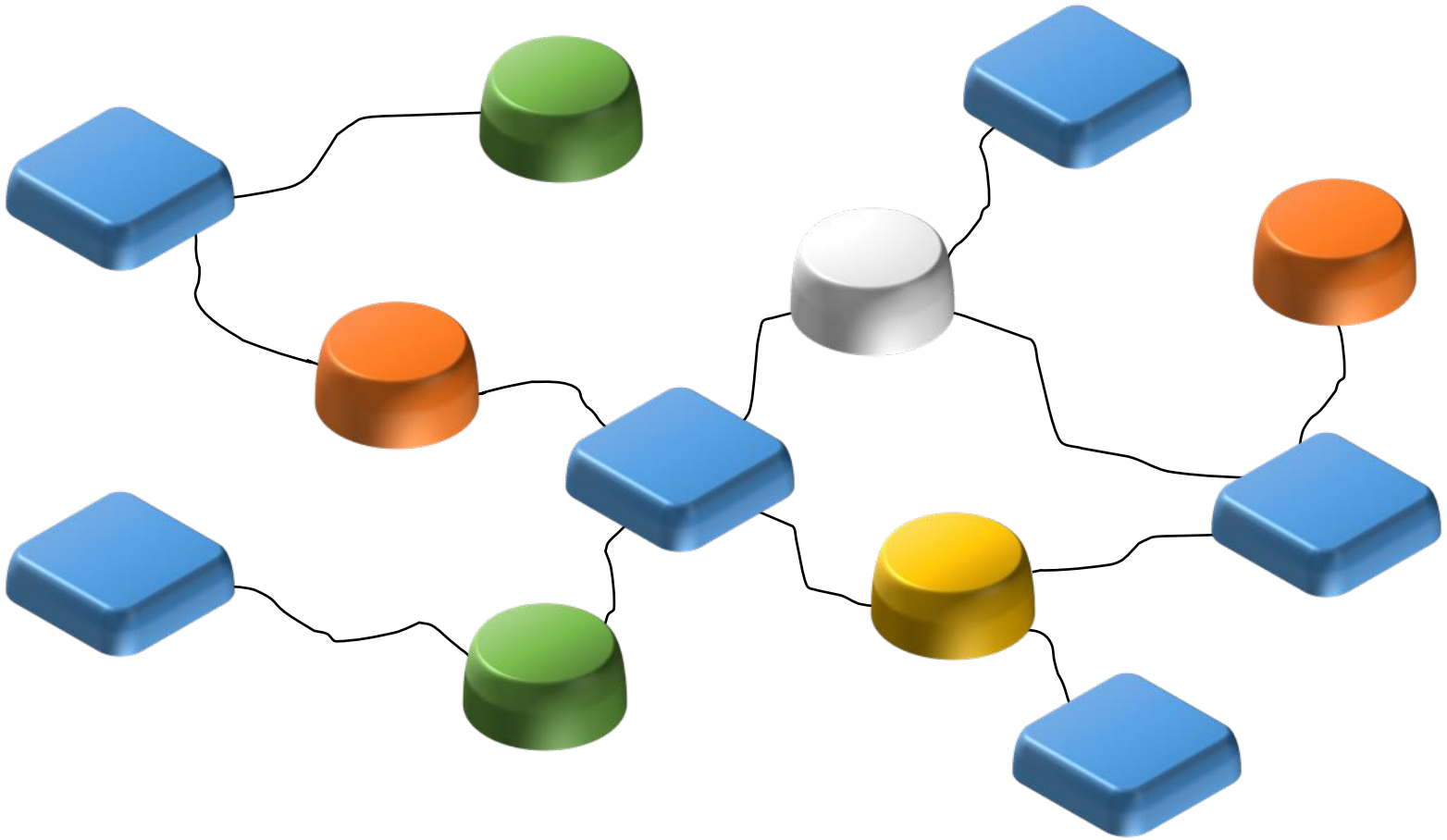
Sequential Consistency



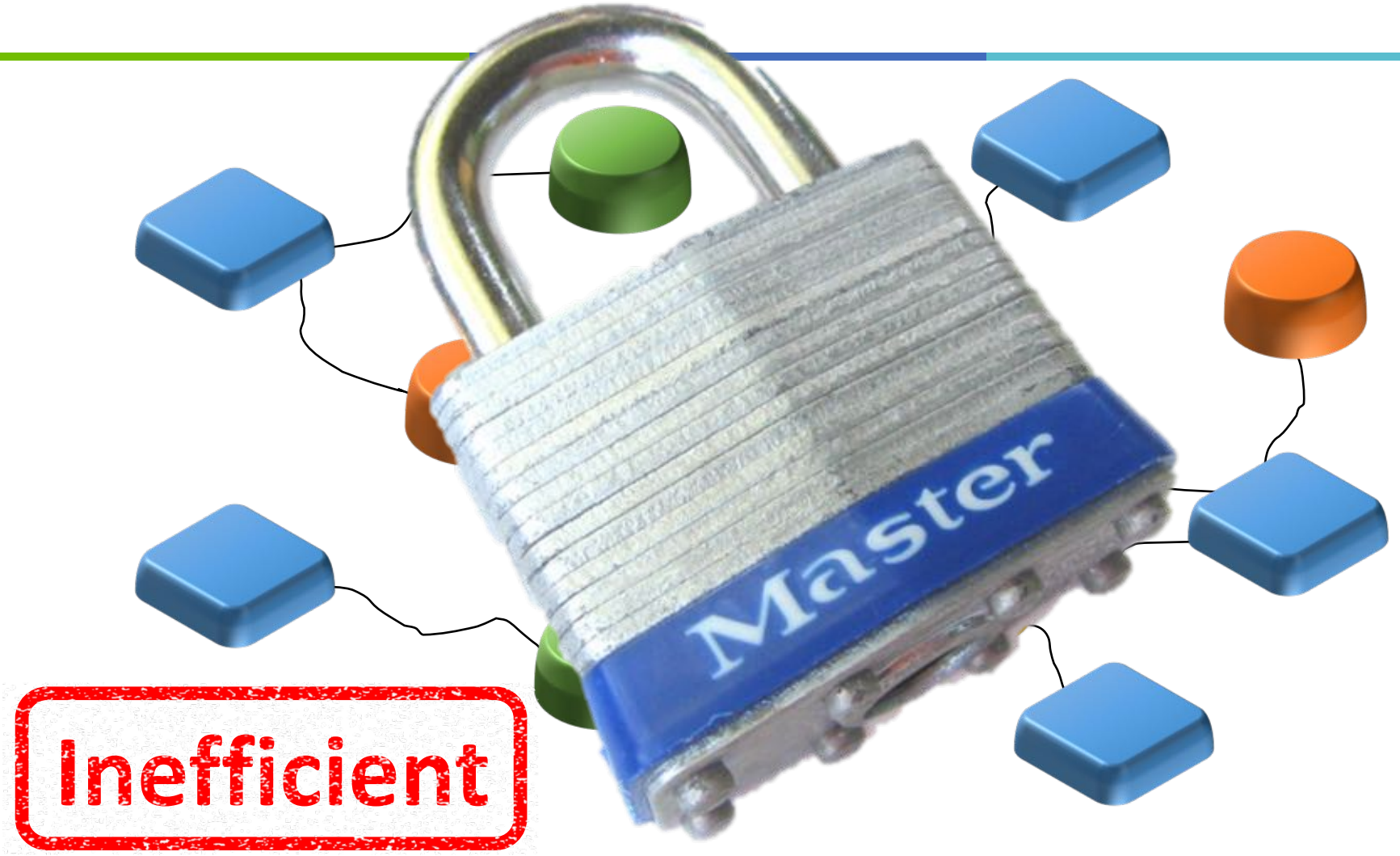
Sequential consistency



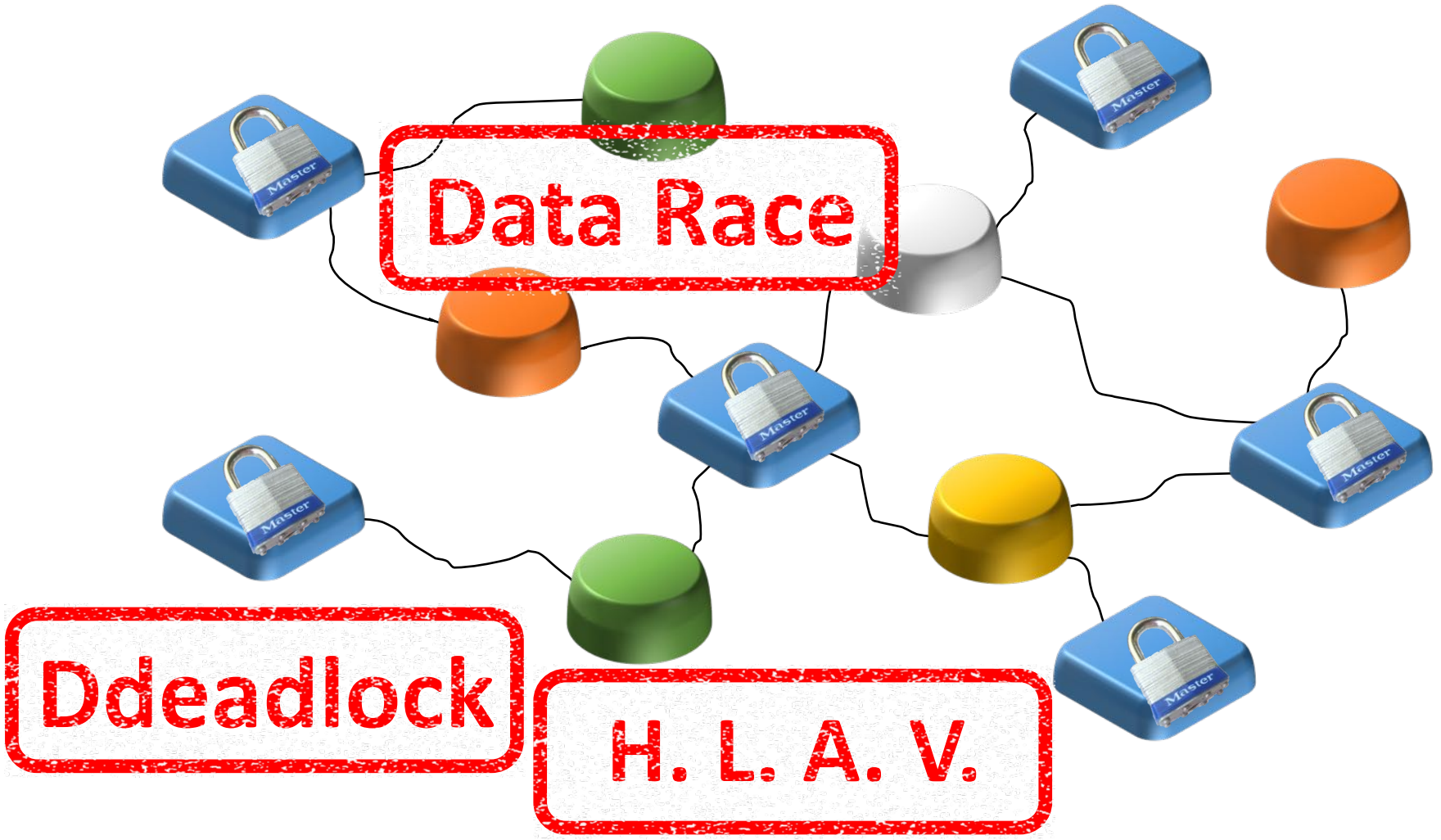
Sharing Multiple Resources



Sharing Multiple Resources



Sharing Multiple Resources



Assigning Semantics to Concurrent Programs

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

- What are the final values for 'X', 'Y', 'a' and 'b'?
  - X = 1, Y = 2

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

- What are the final values for 'X', 'Y', 'a' and 'b'?
  - X = 1, Y = 2, **a = ?, b = ?**

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

- What are the final values for 'X', 'Y', 'a' and 'b'?
  - X = 1, Y = 2, **a = ?, b = ?**
- Depends on the interleavings of the assignments
  - Sequential Consistency [Lamport'79]
  - Program behavior = set of interleavings

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1
Y = 2

a = Y
b = X

X = 1

Y = 2

a = Y

b = X

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1
Y = 2

a = Y
b = X

X = 1

Y = 2

a = Y

b = X

a=2, b=1

Assigning Semantics to Concurrent Programs

X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

X = 1	X = 1
Y = 2	a = Y
a = Y	b = X
b = X	Y = 2

a=2, b=1

Assigning Semantics to Concurrent Programs

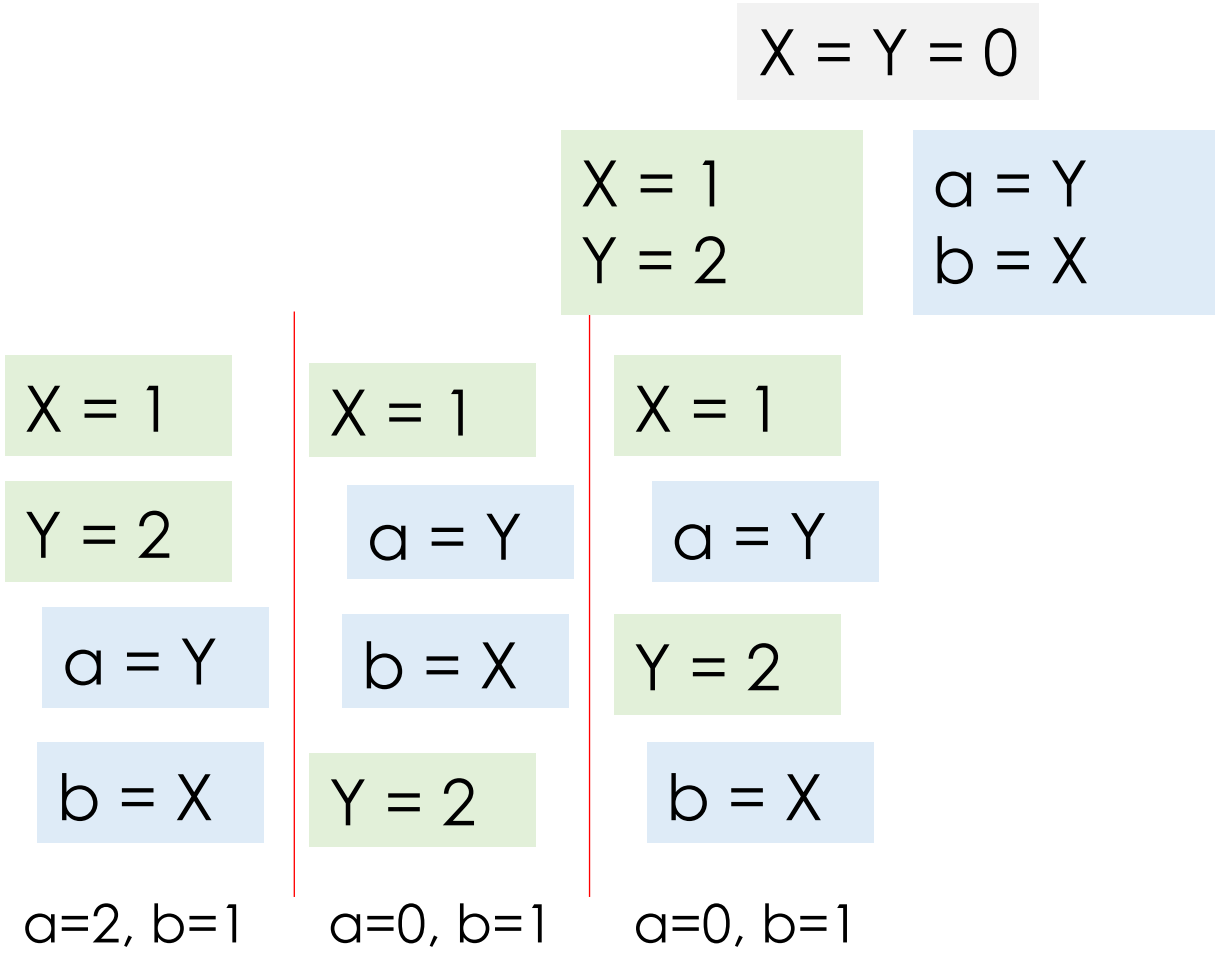
X = Y = 0

X,Y => Global Vars
a, b => Local Vars

X = 1 Y = 2	a = Y b = X

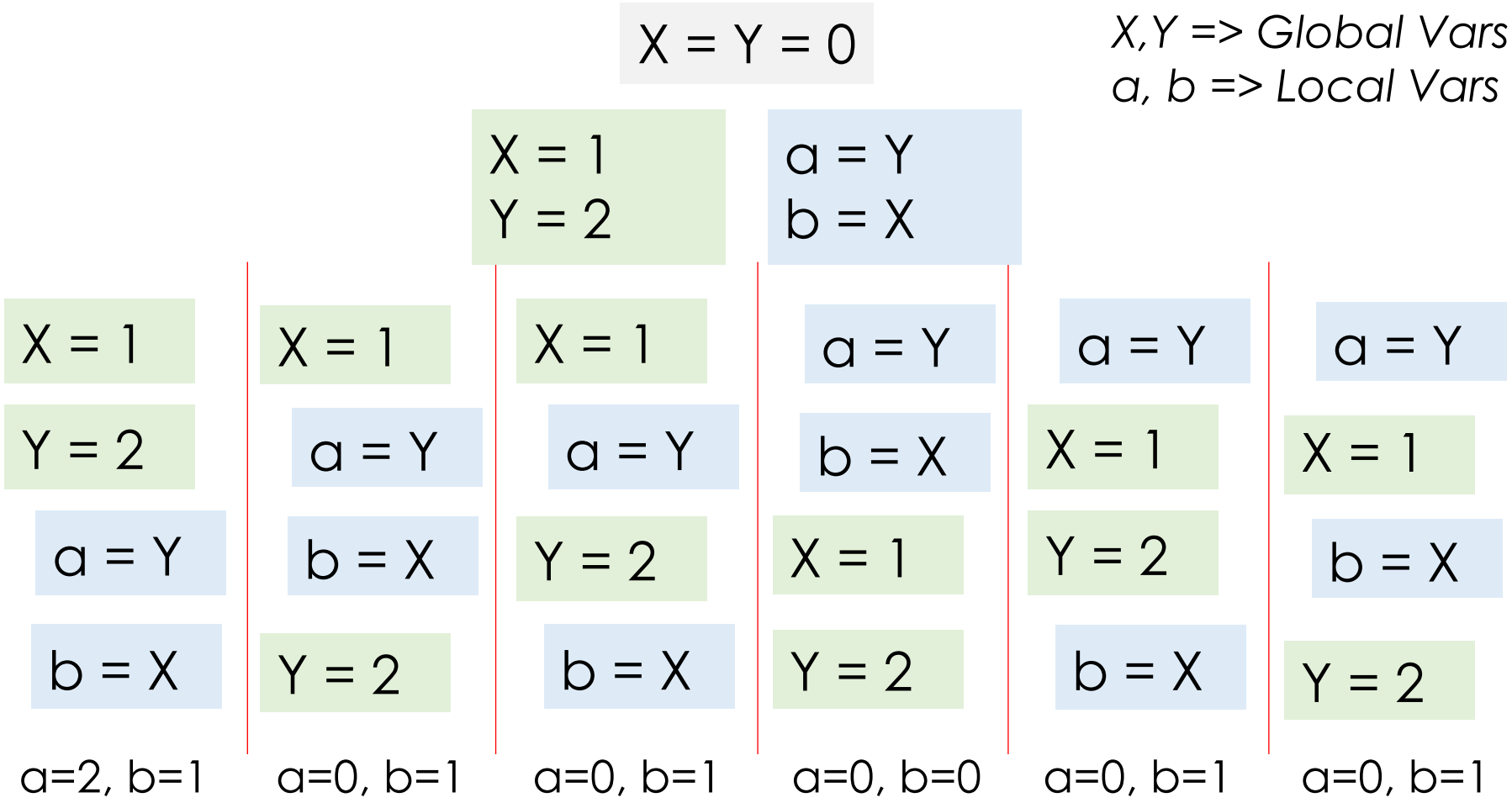
X = 1	X = 1
Y = 2	a = Y
a = Y	b = X
b = X	Y = 2
a=2, b=1	a=0, b=1

Assigning Semantics to Concurrent Programs

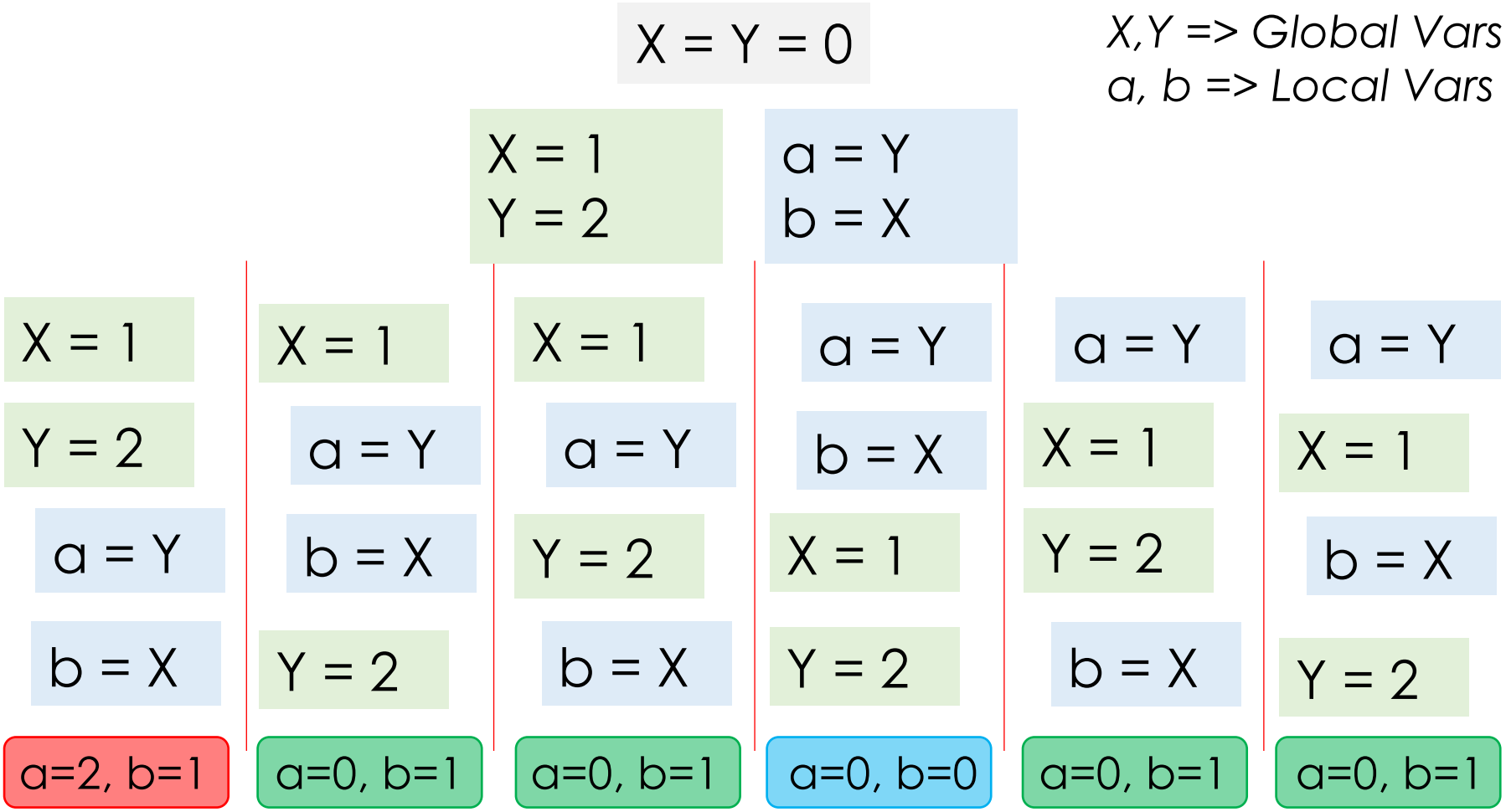


X,Y => Global Vars
a, b => Local Vars

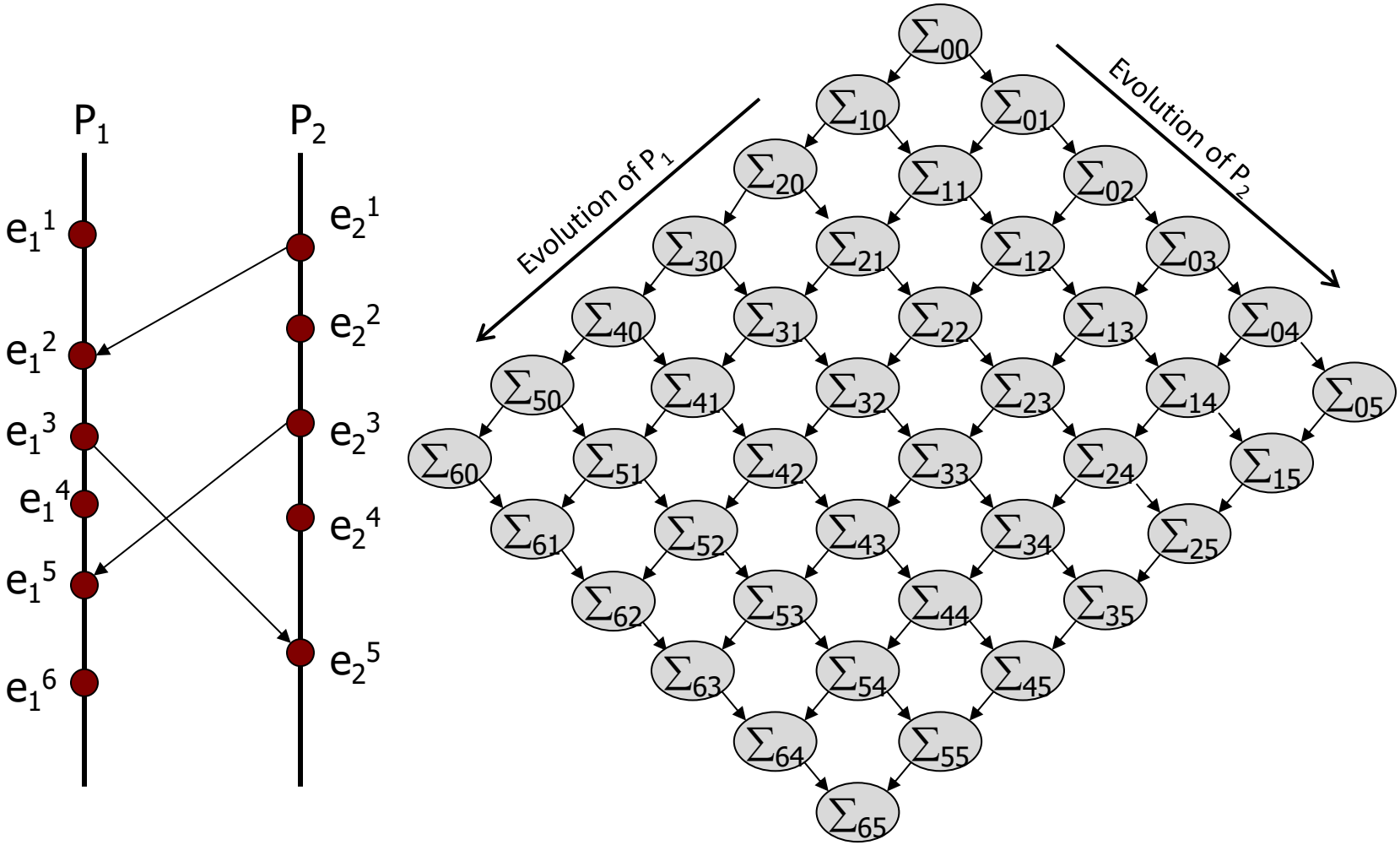
Assigning Semantics to Concurrent Programs



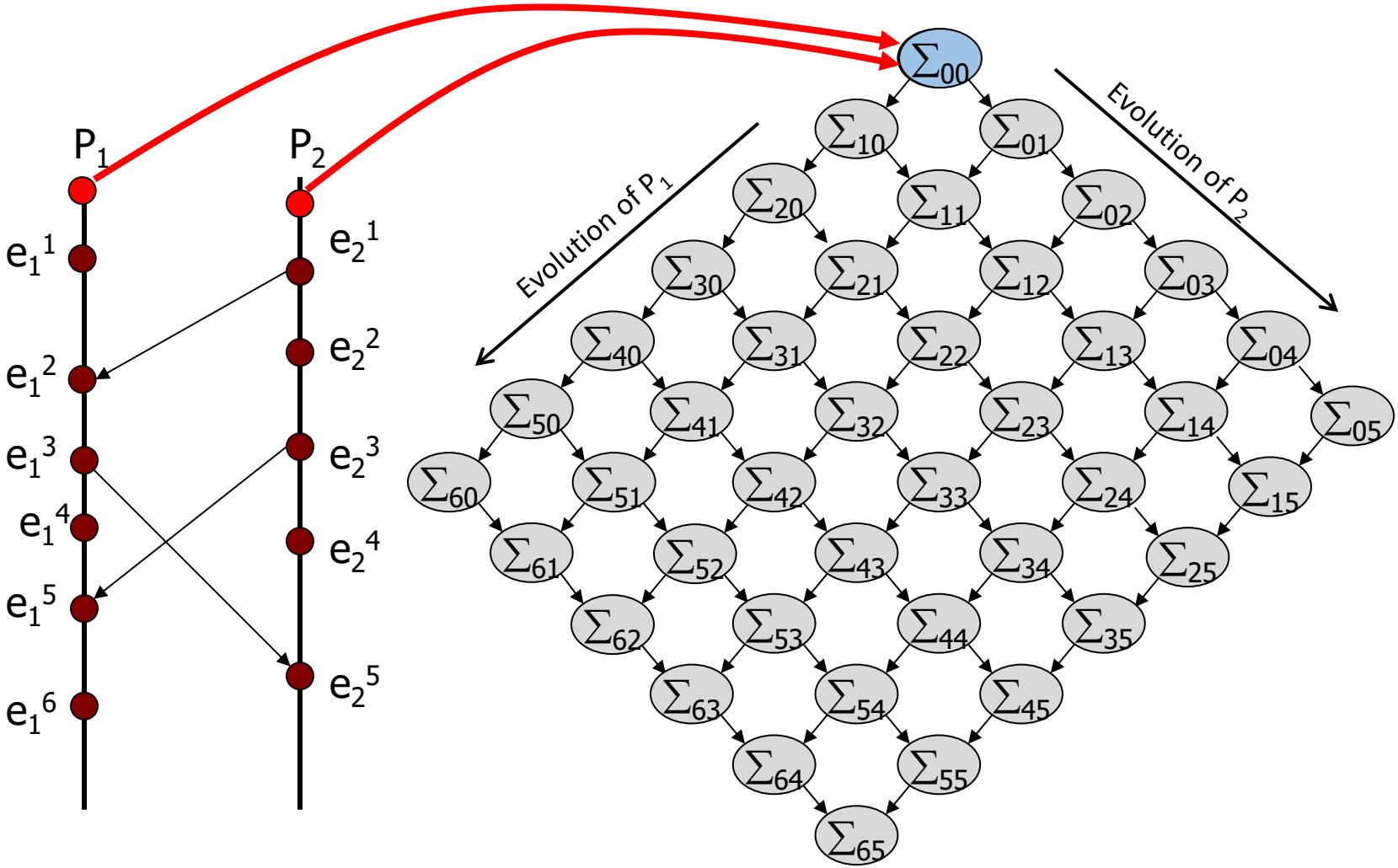
Assigning Semantics to Concurrent Programs



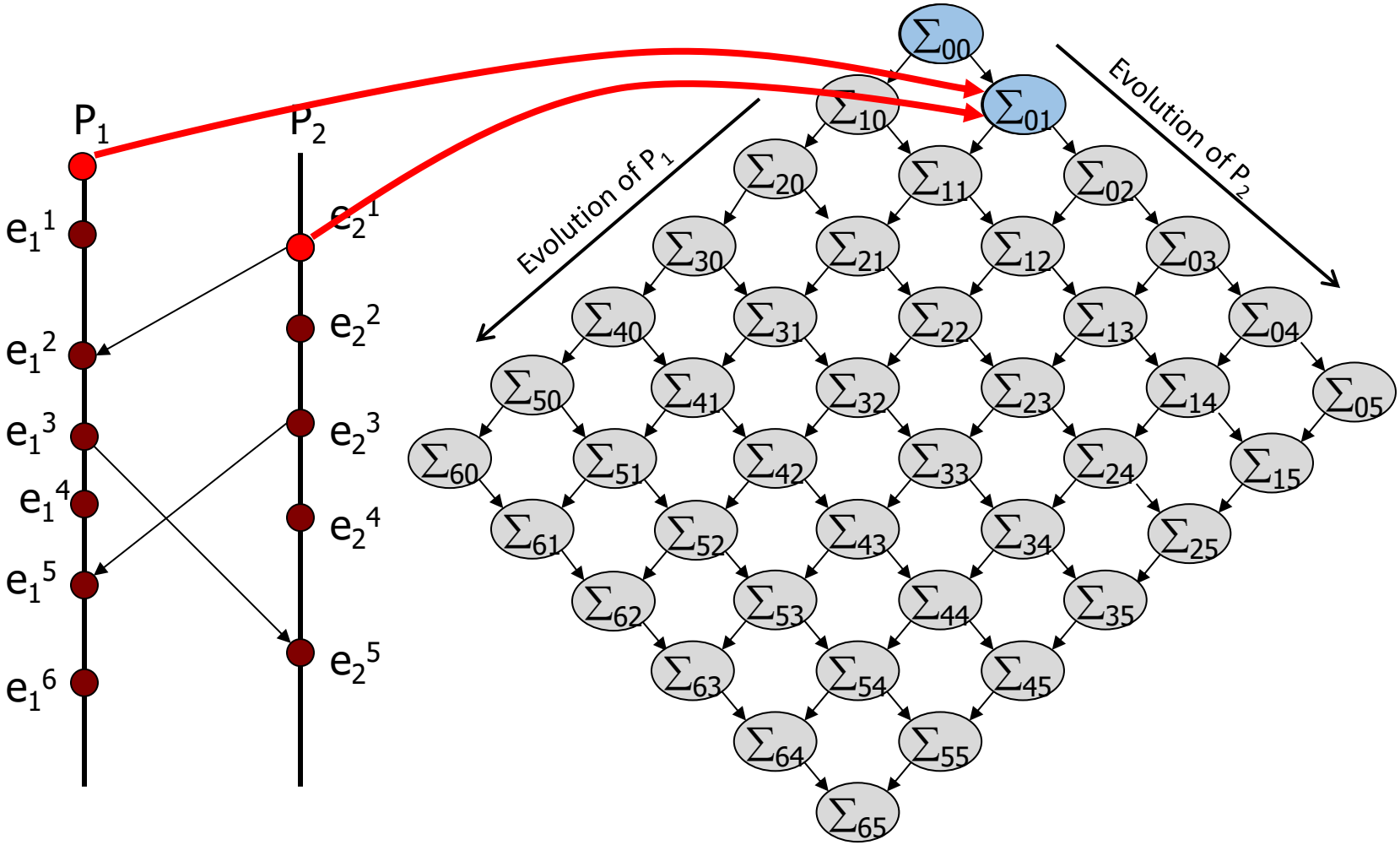
State explosion in concurrent programs



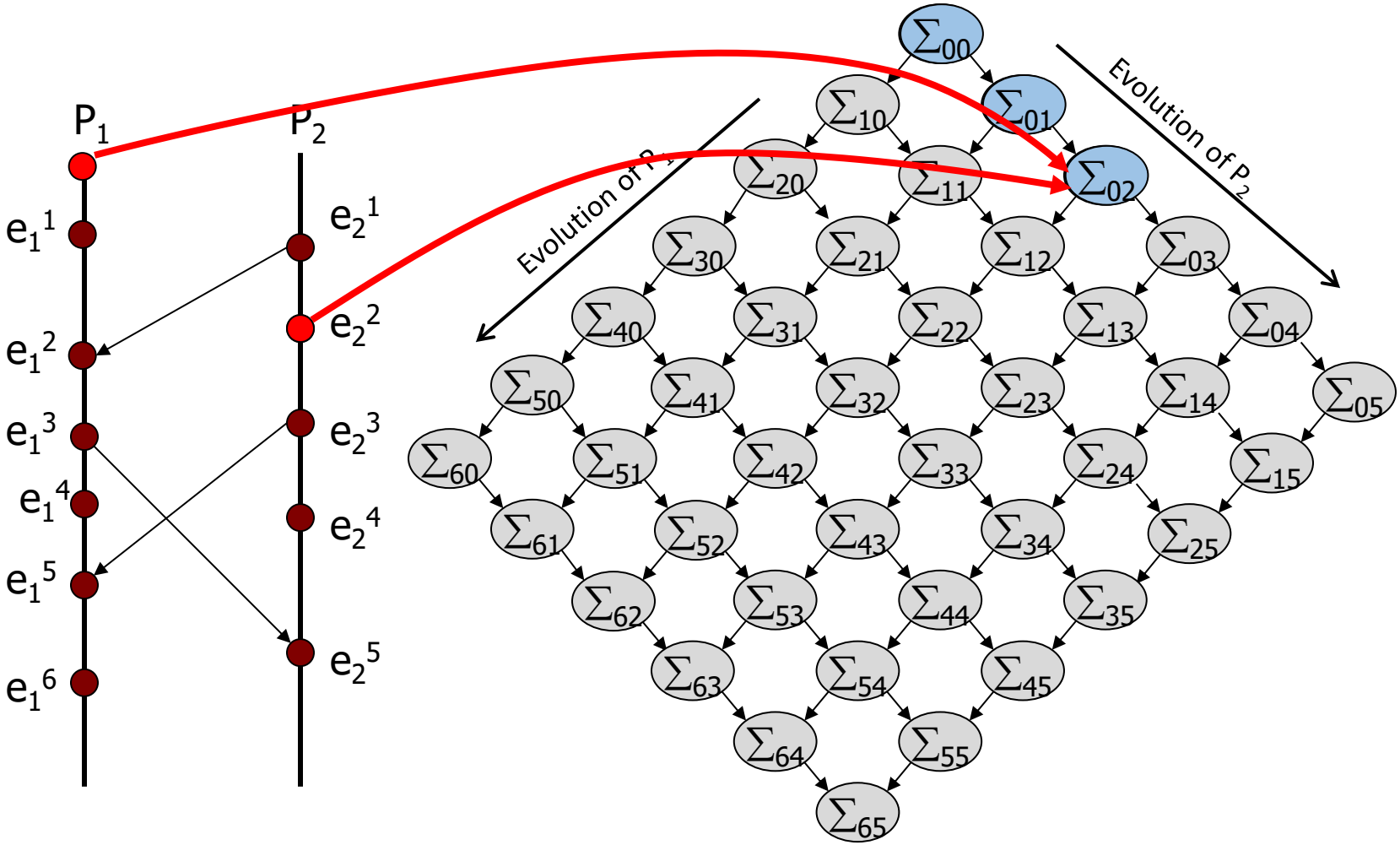
State explosion in concurrent programs



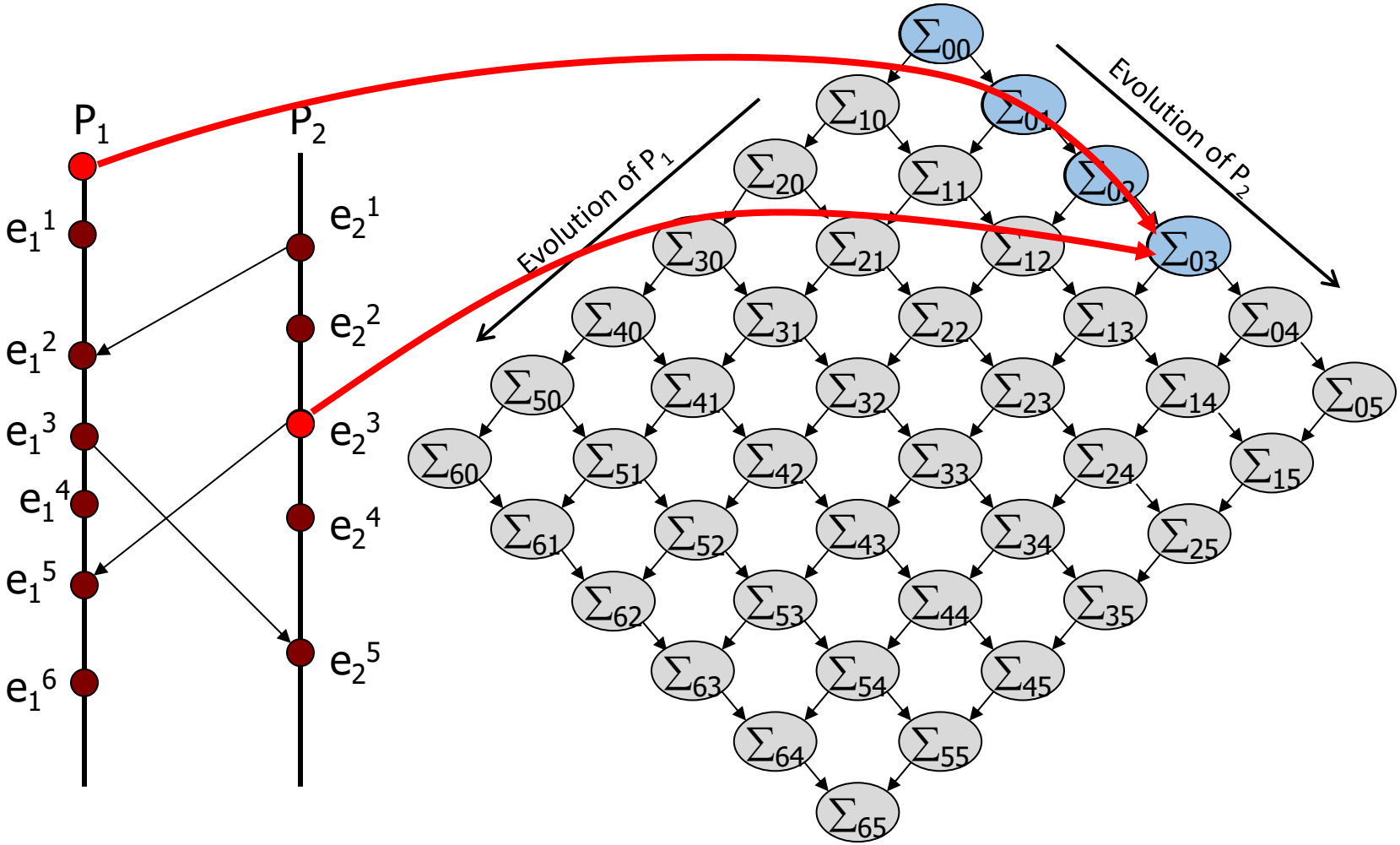
State explosion in concurrent programs



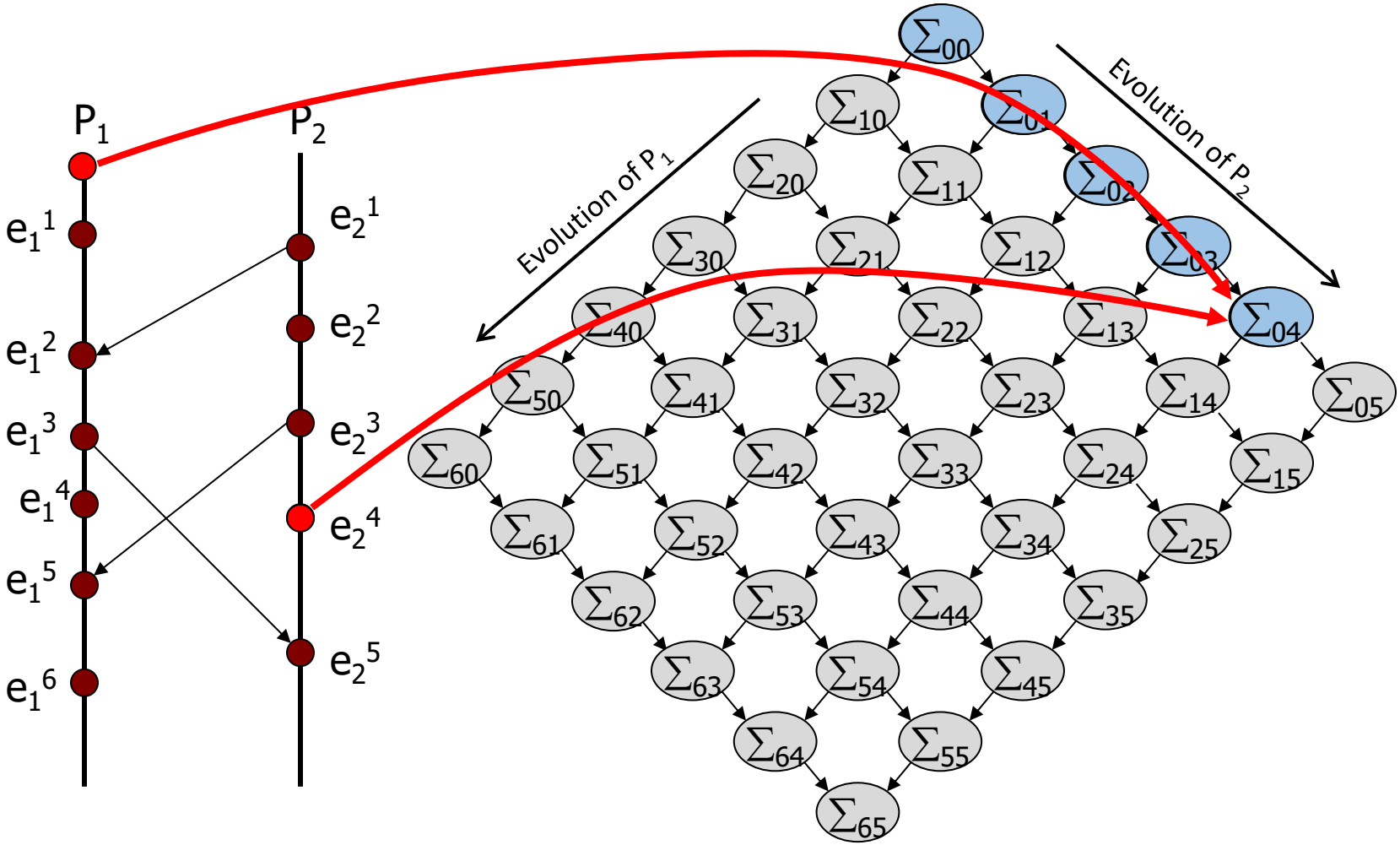
State explosion in concurrent programs



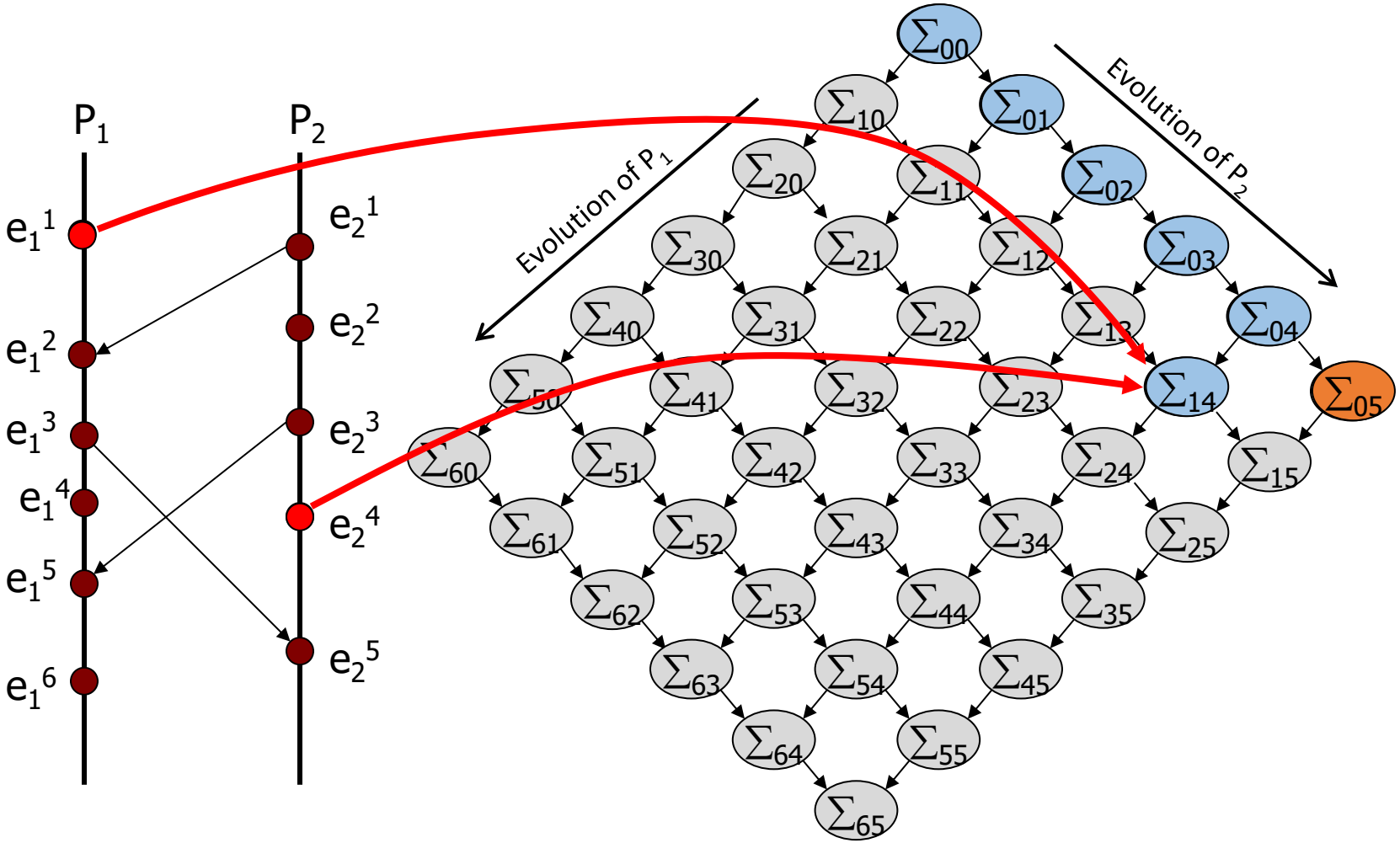
State explosion in concurrent programs



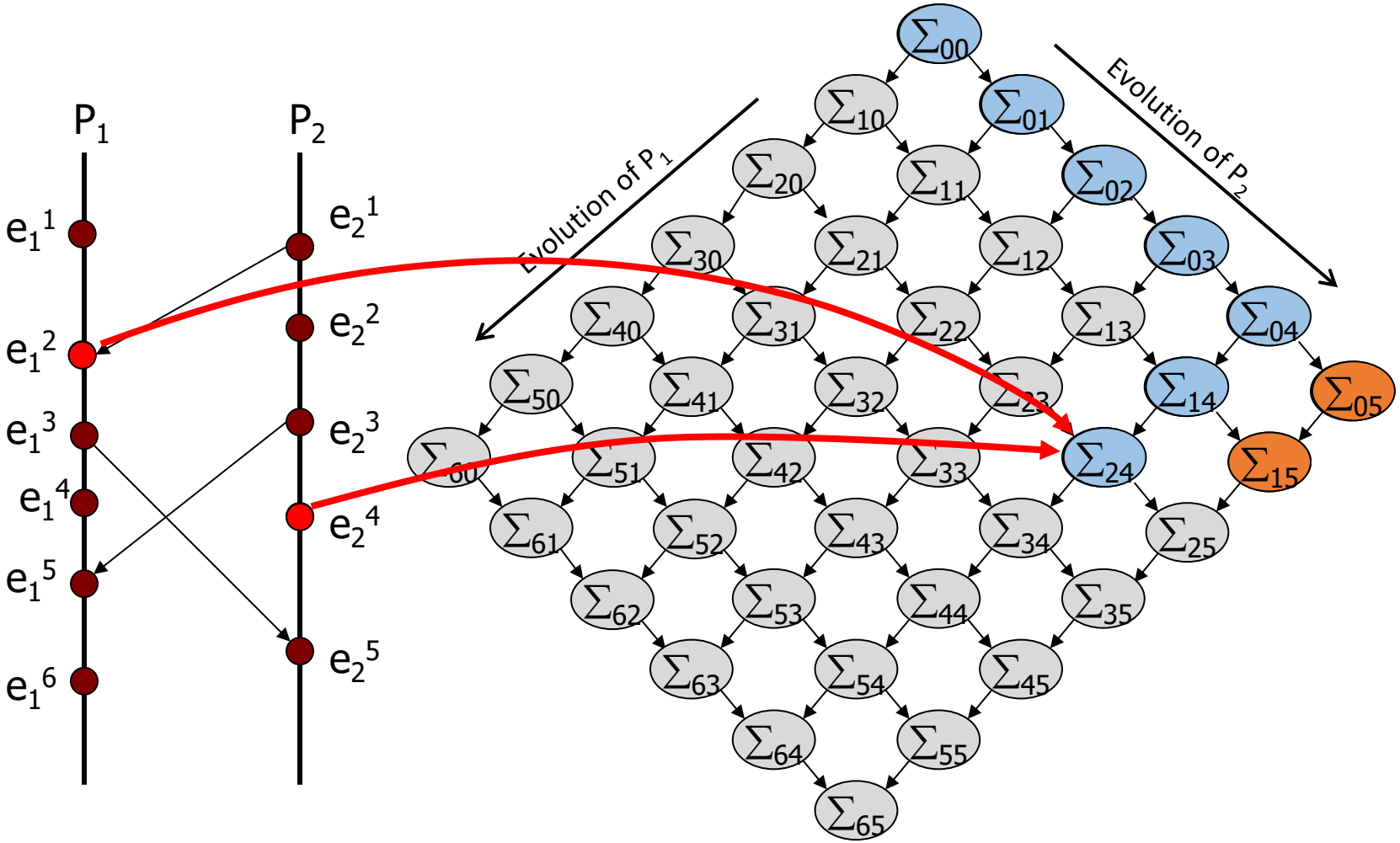
State explosion in concurrent programs



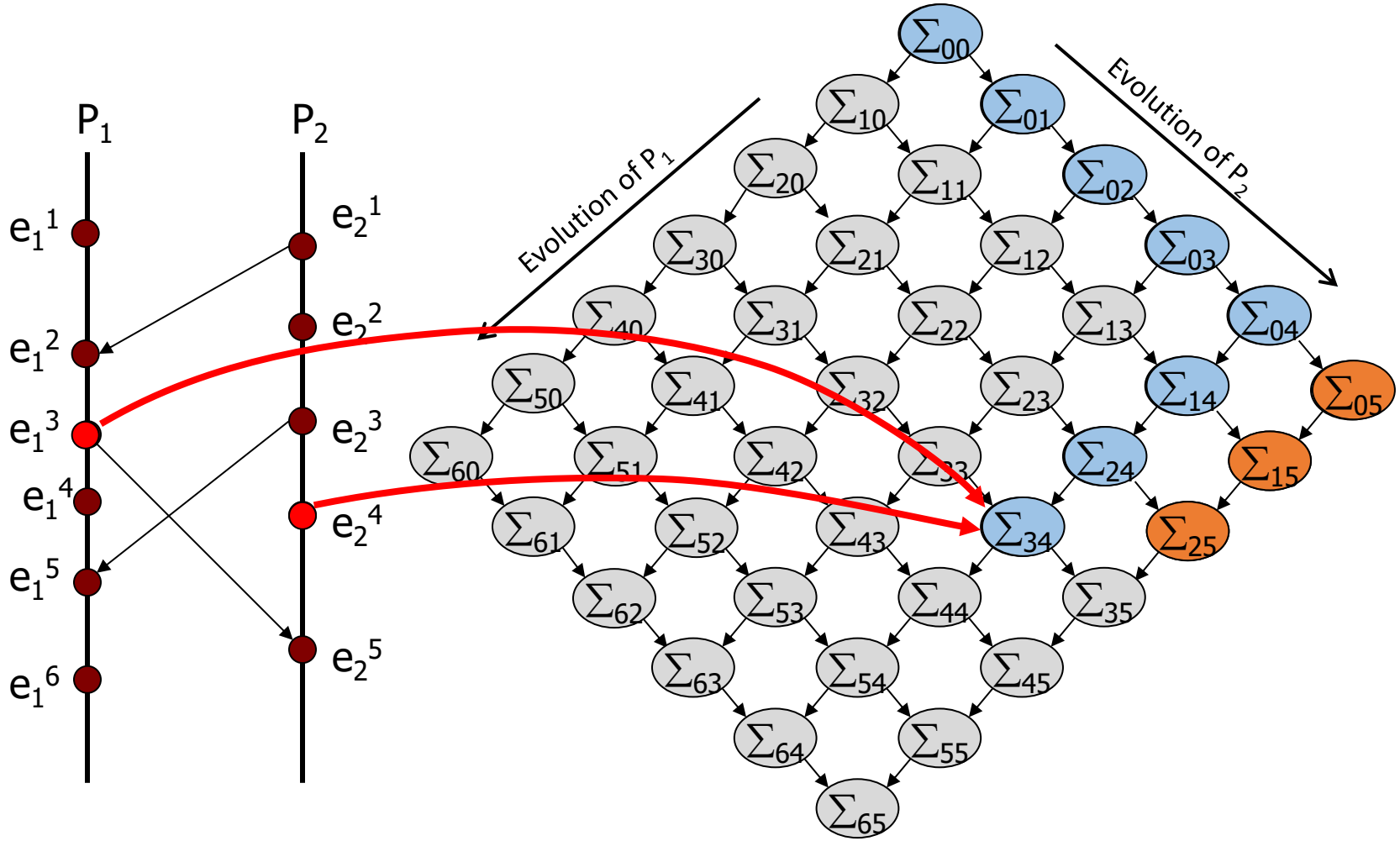
State explosion in concurrent programs



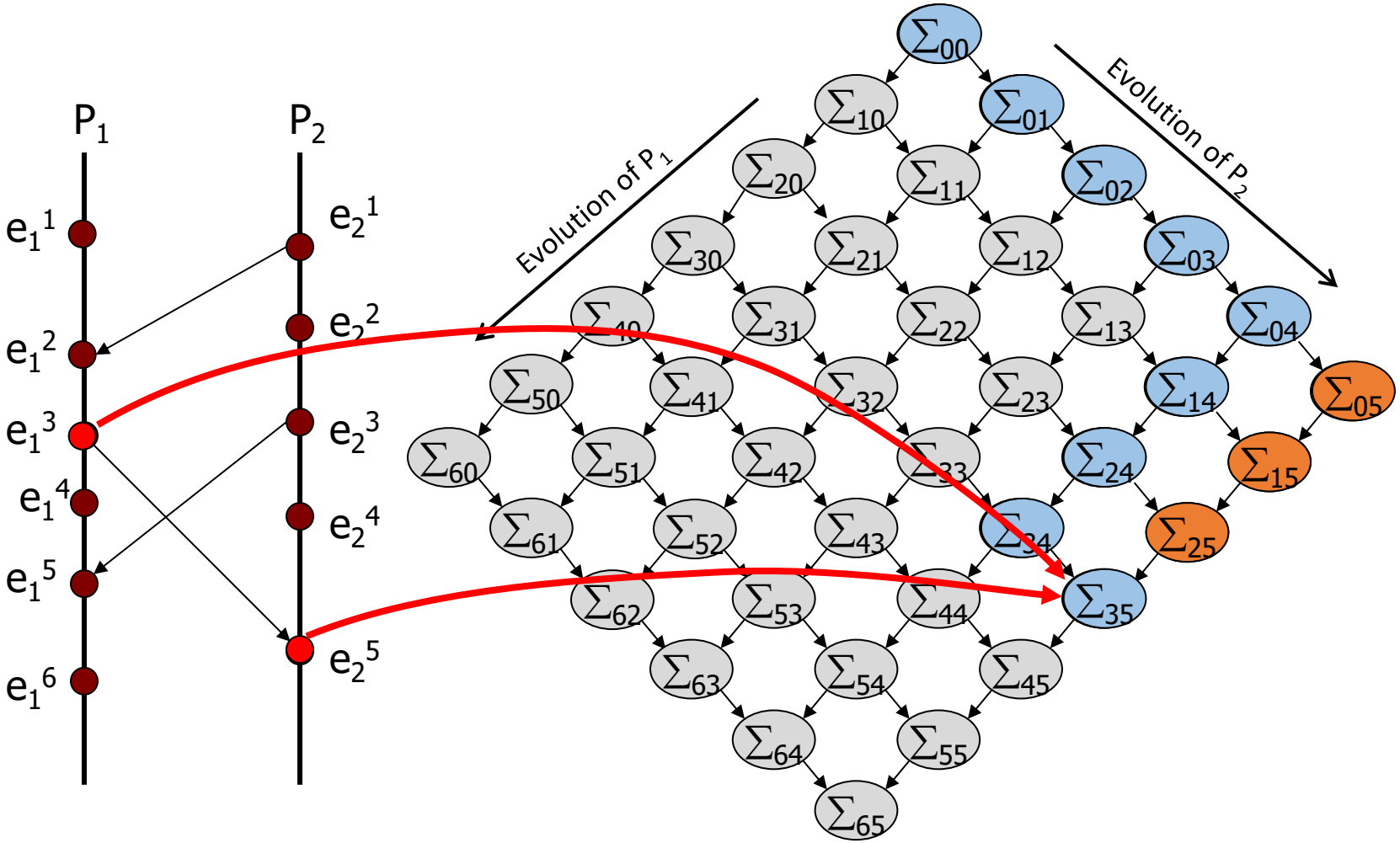
State explosion in concurrent programs



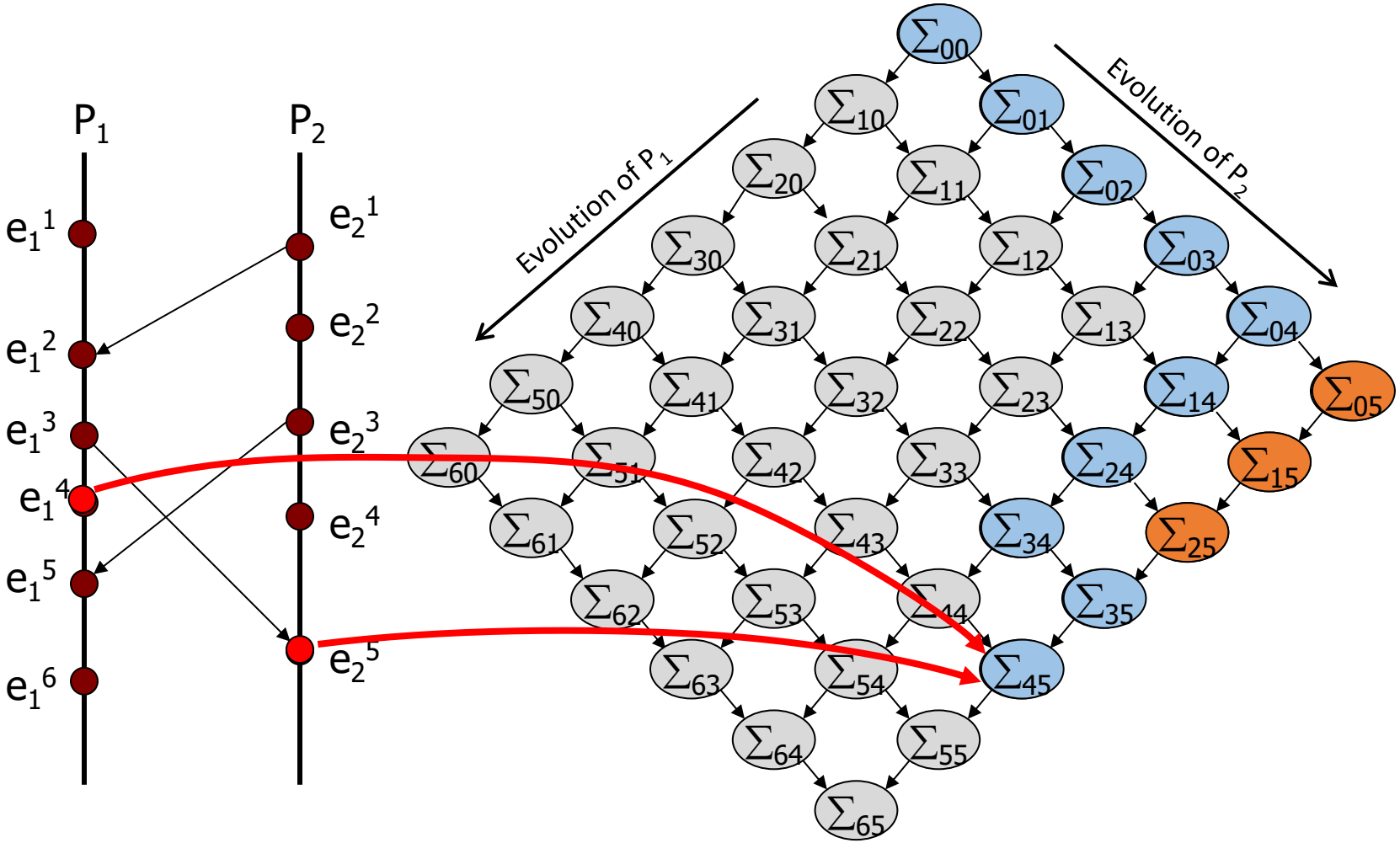
State explosion in concurrent programs



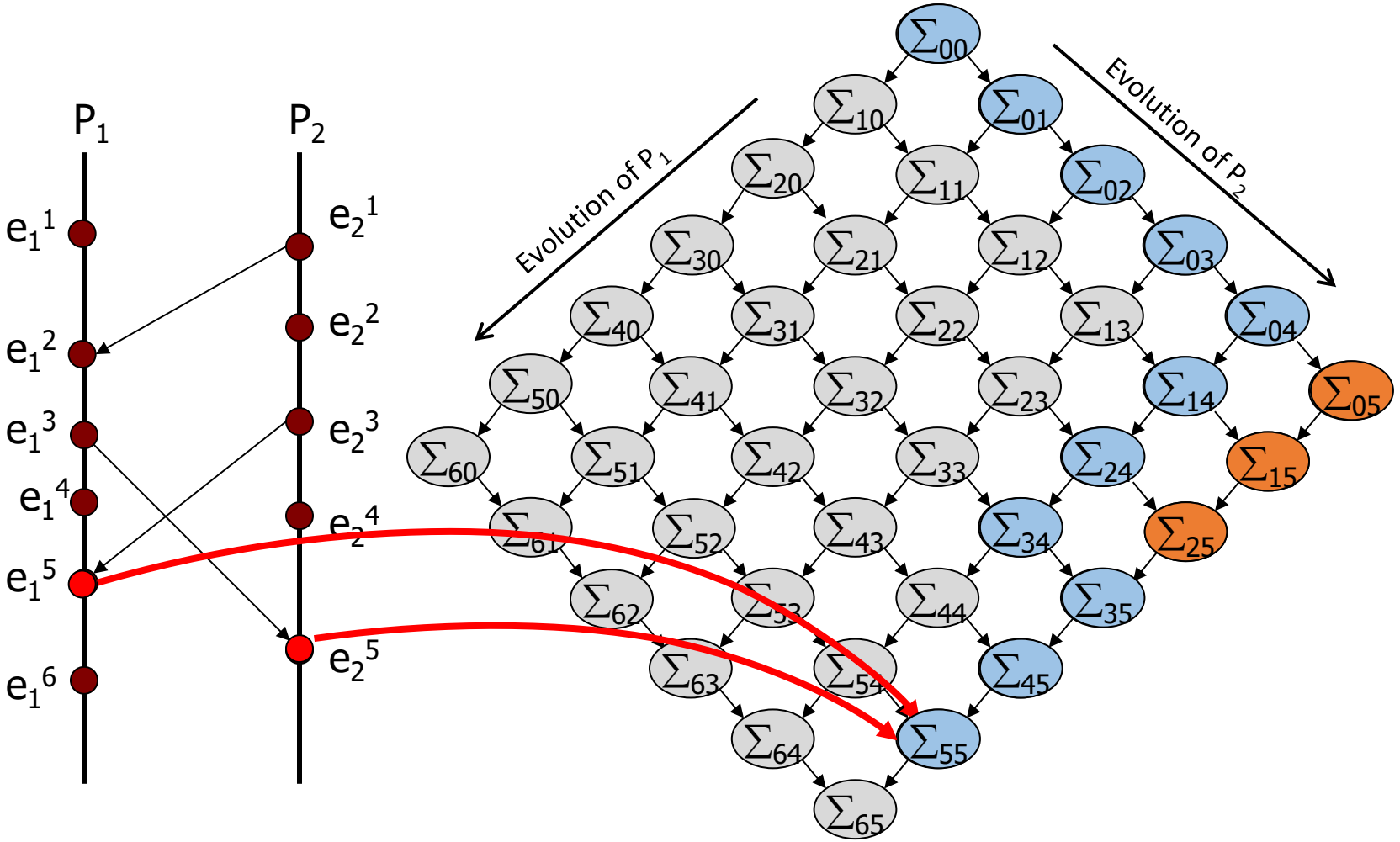
State explosion in concurrent programs



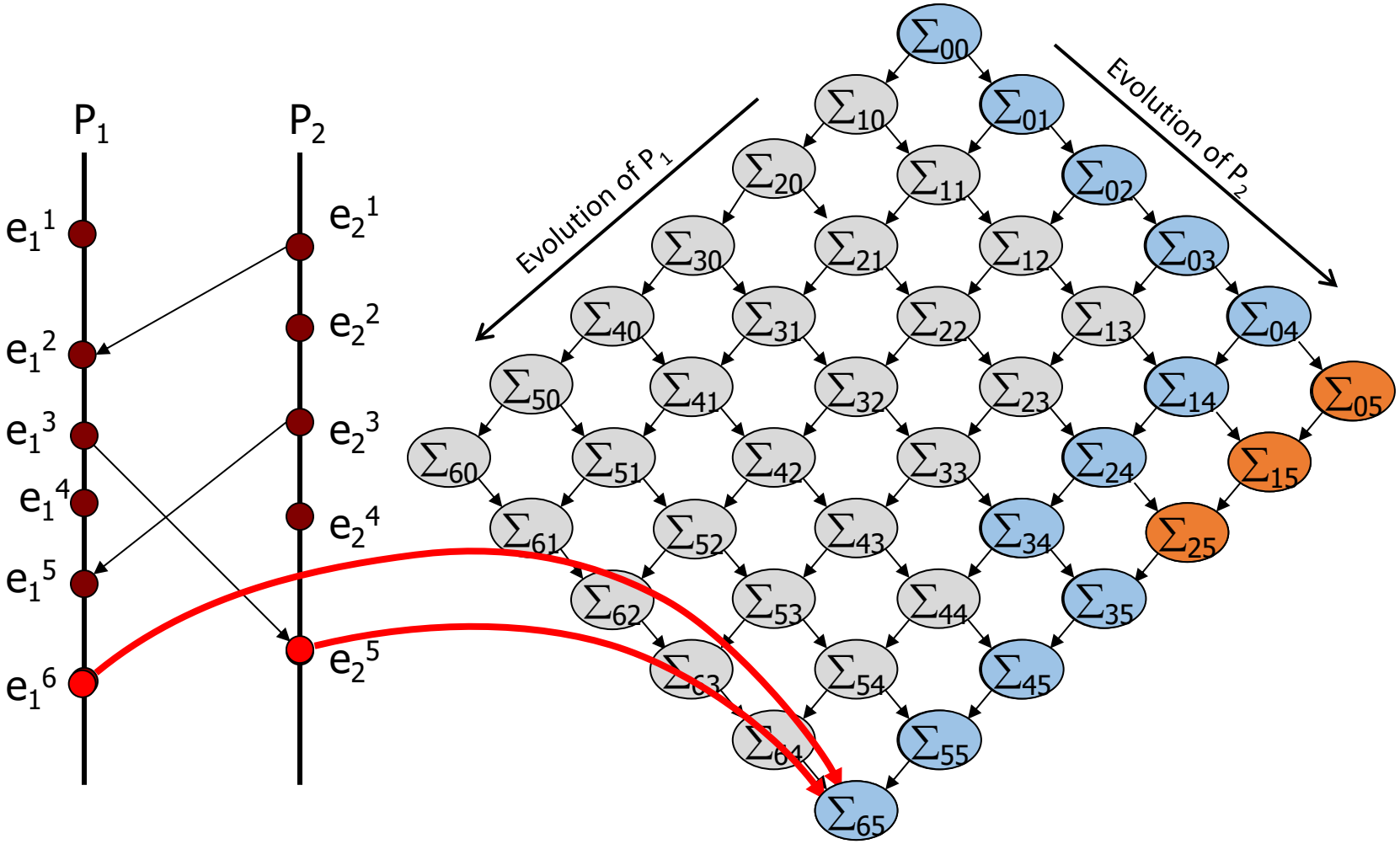
State explosion in concurrent programs



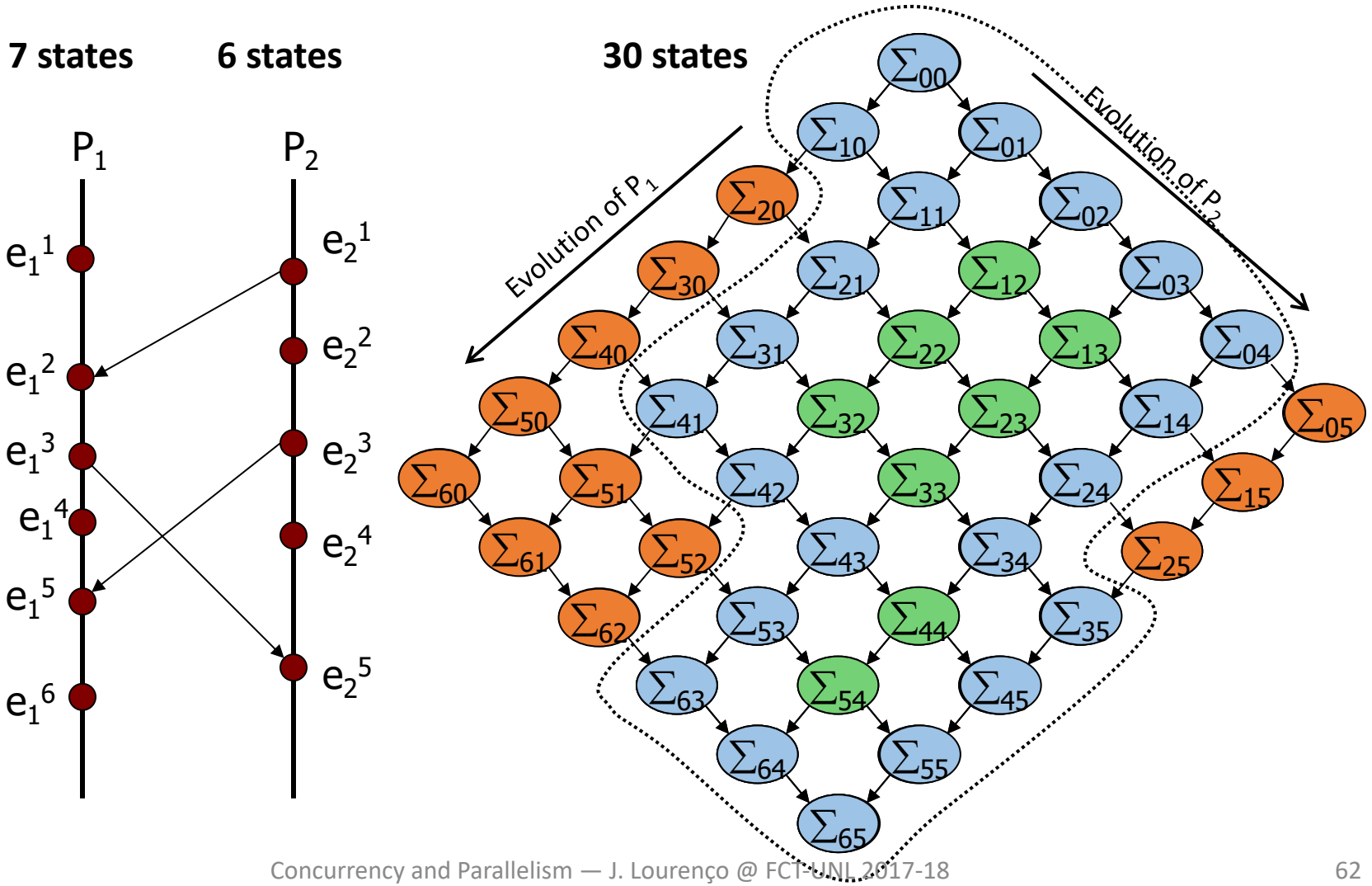
State explosion in concurrent programs



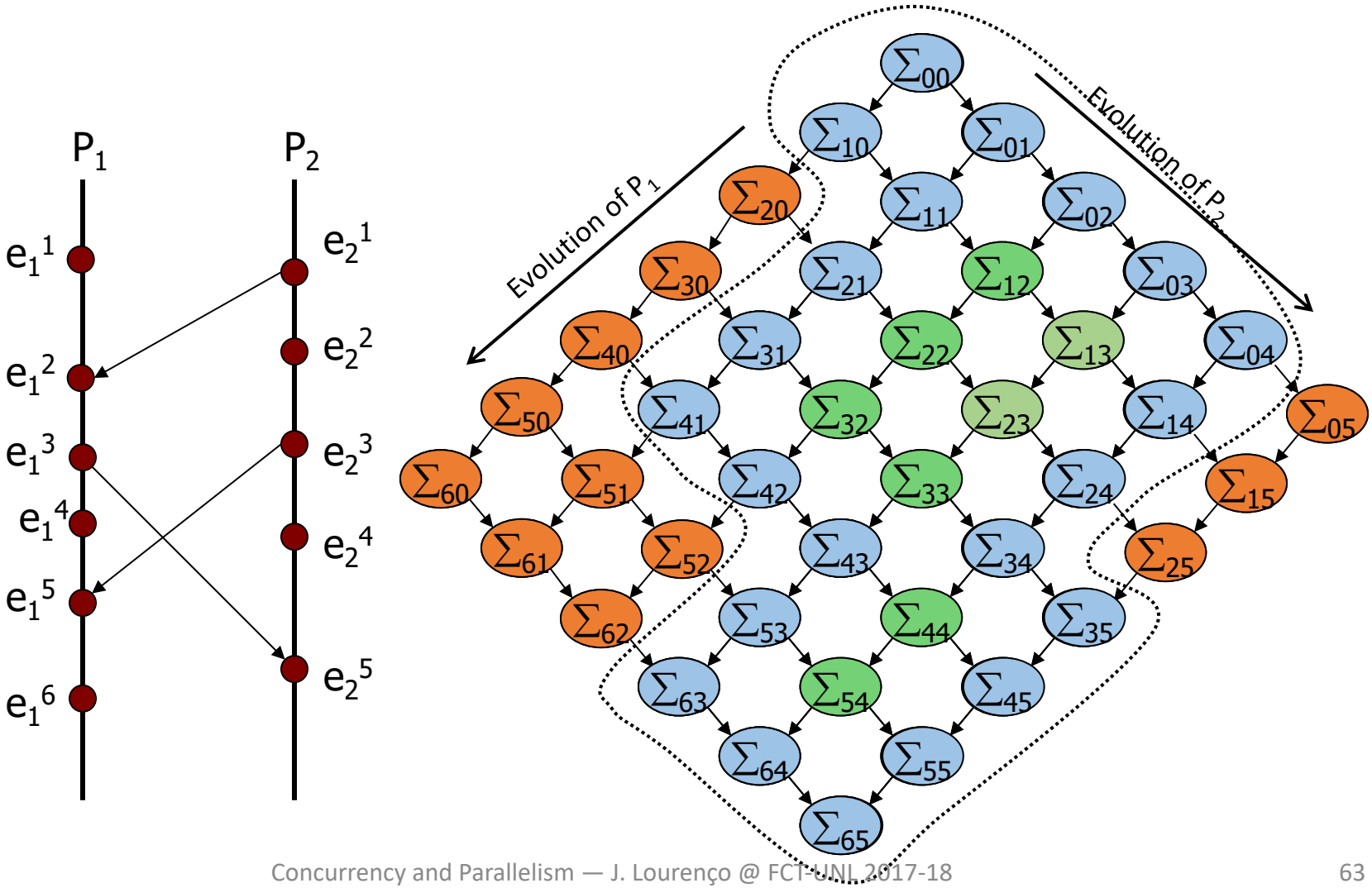
State explosion in concurrent programs



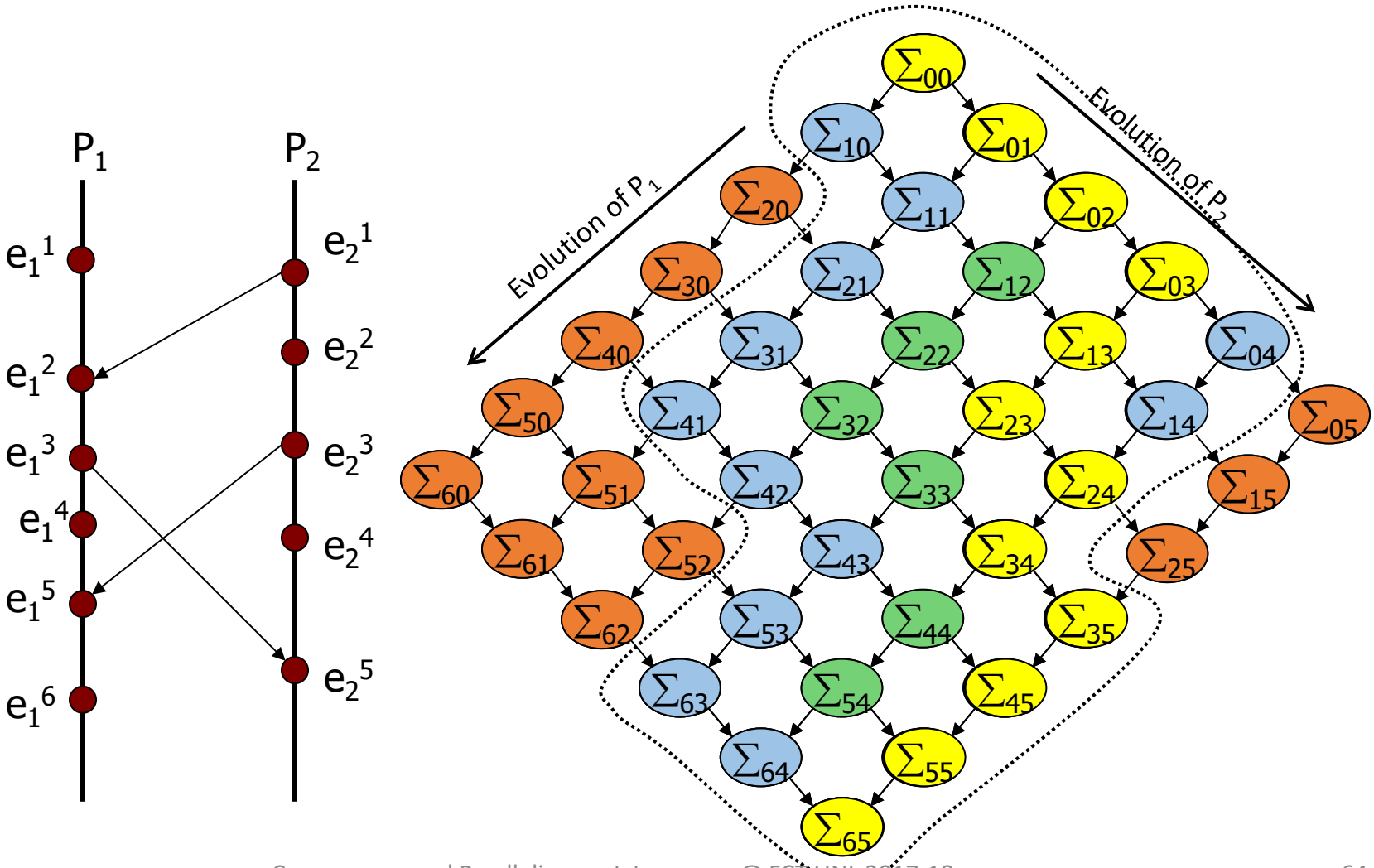
State explosion in concurrent programs



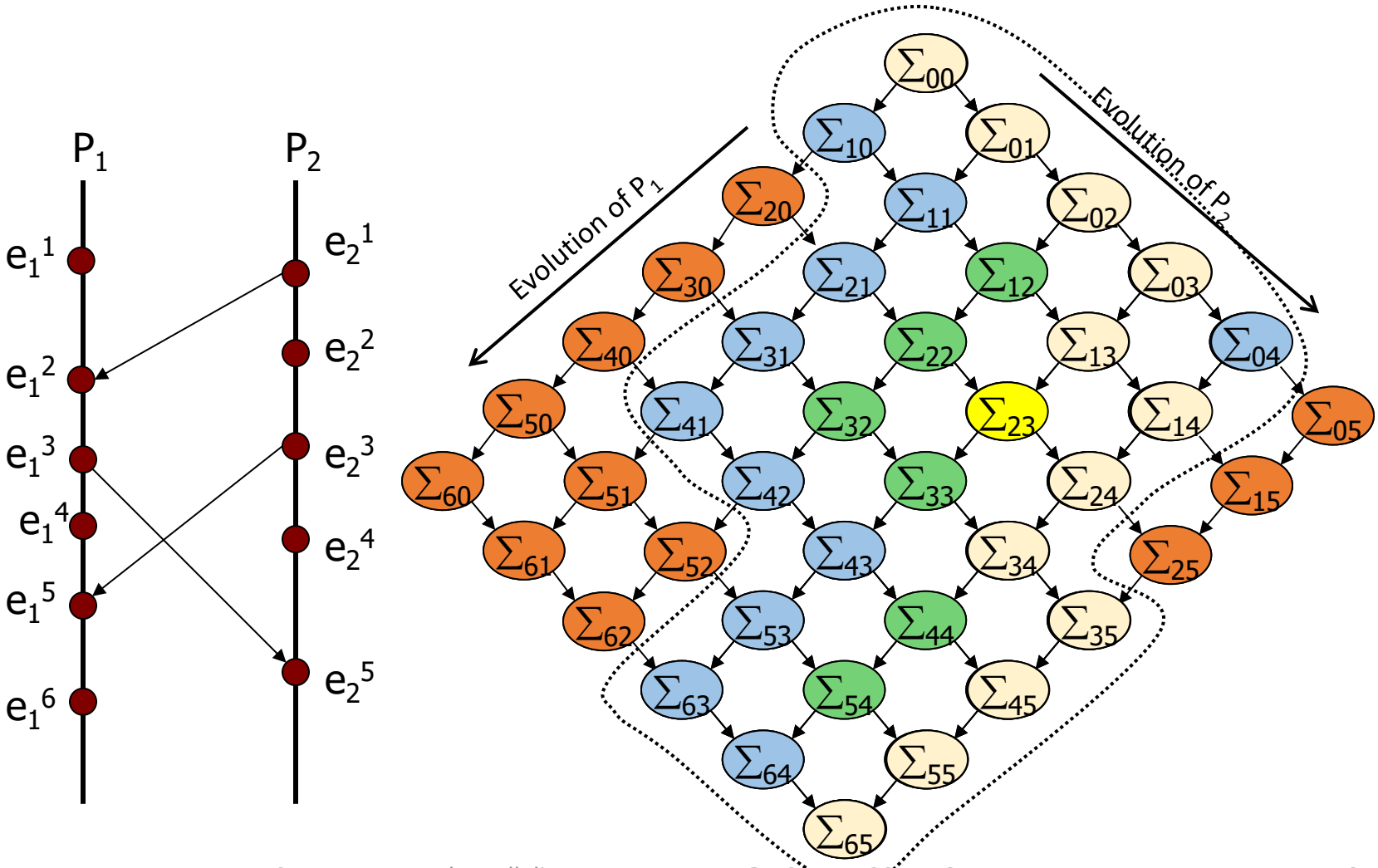
Consistent run



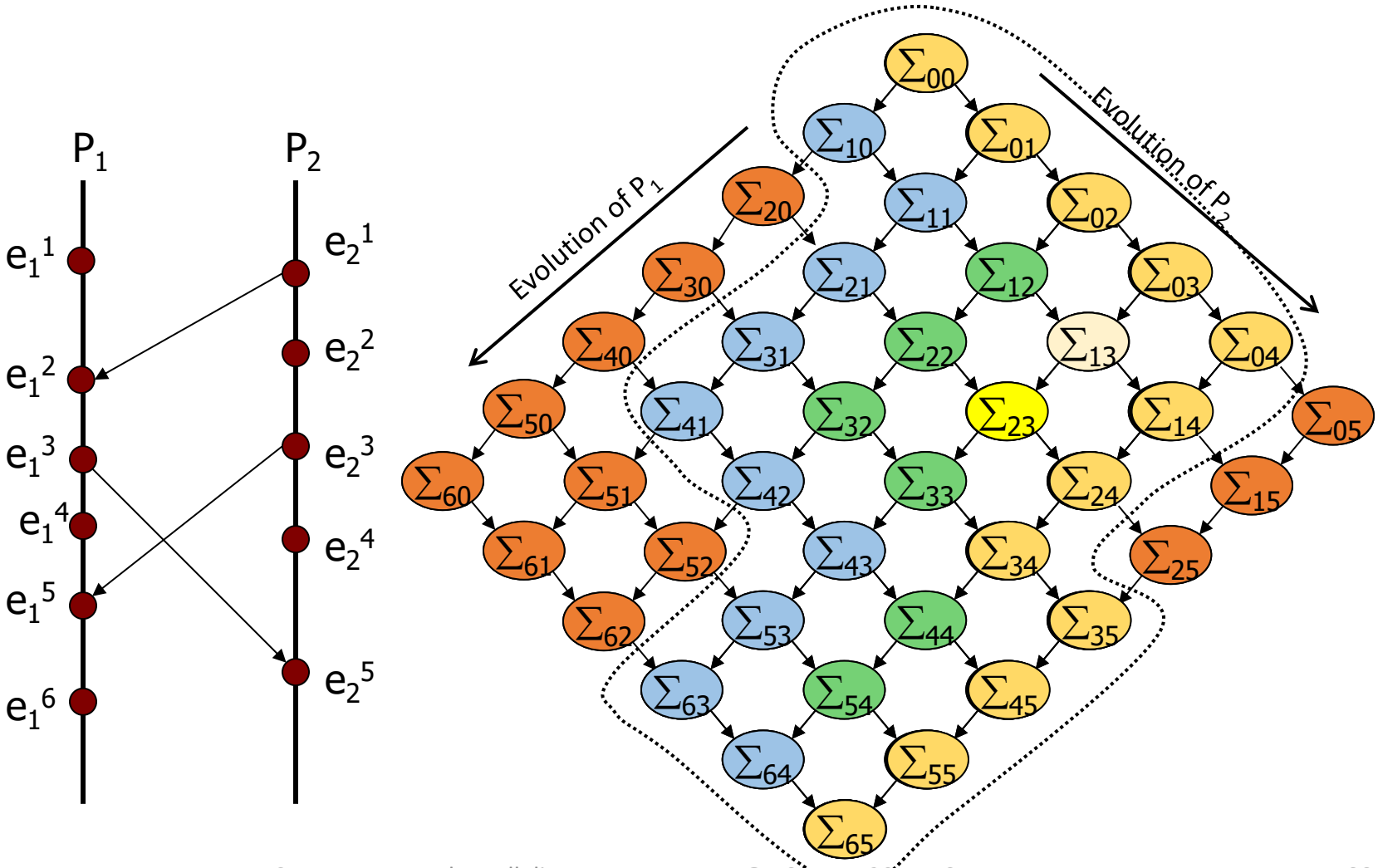
Consistent run



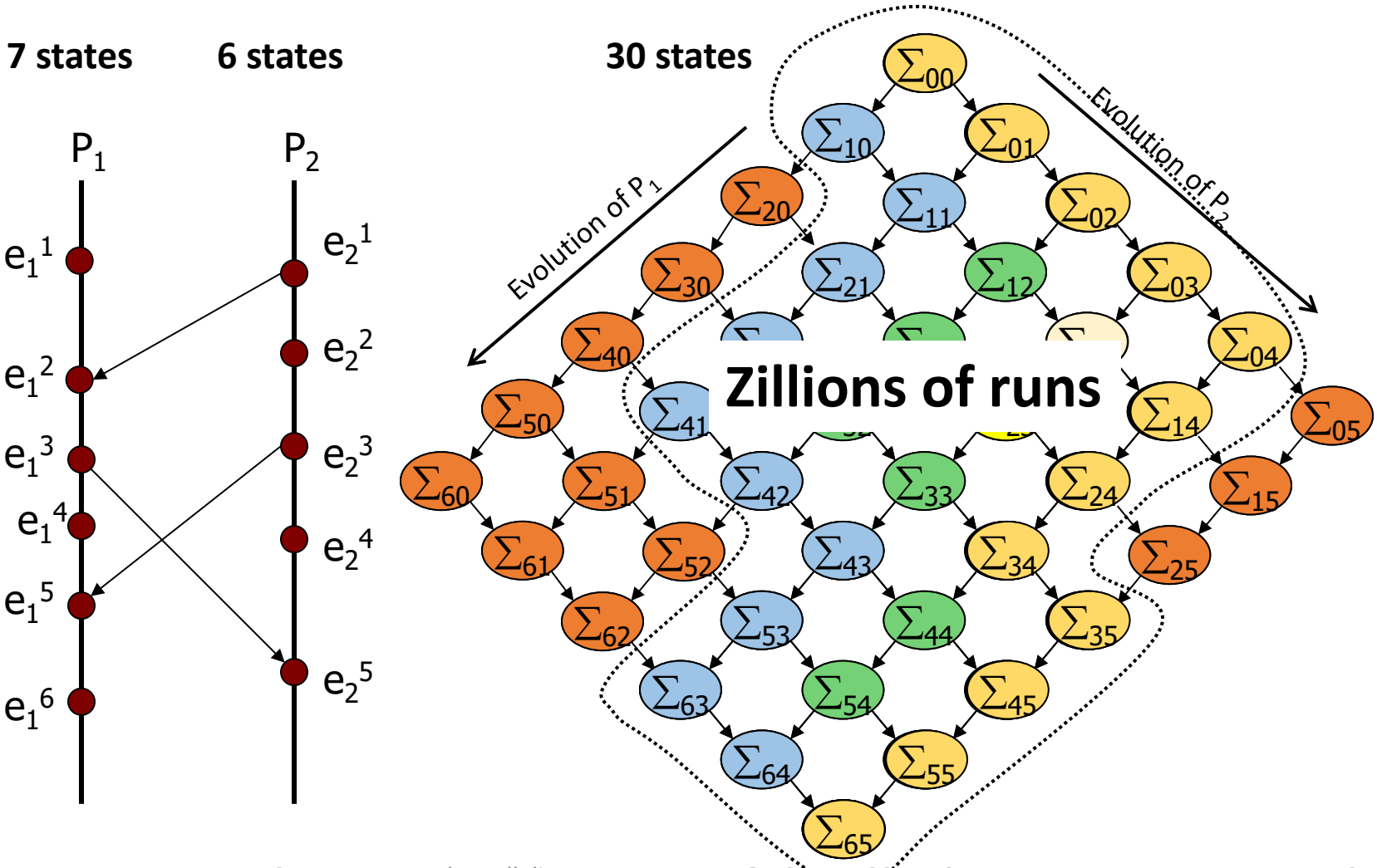
Consistent run



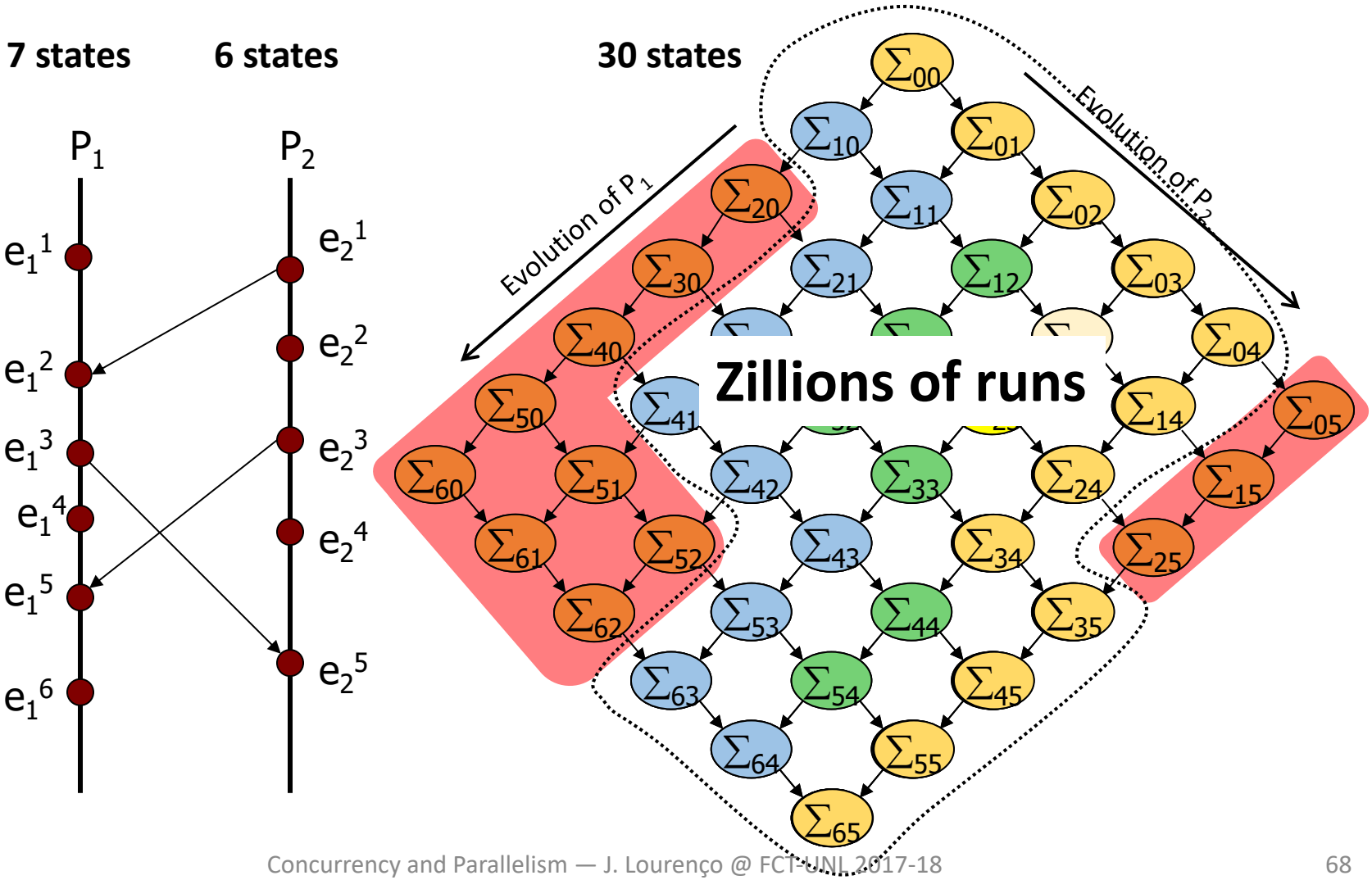
Consistent run



Consistent run



What does it mean the program has an error?



Concurrency Errors

Common Concurrency Errors

- Data races (atomicity violations)
- Ordering violations
- Unintended sharing
- High-level atomicity violations
- Deadlocks and livelocks

Potential Data Race

- Code is supposed to execute atomically
  - Multiple dependent instructions manipulate some shared data
- Interleaving with instructions of another thread that access the same data

Thread 1

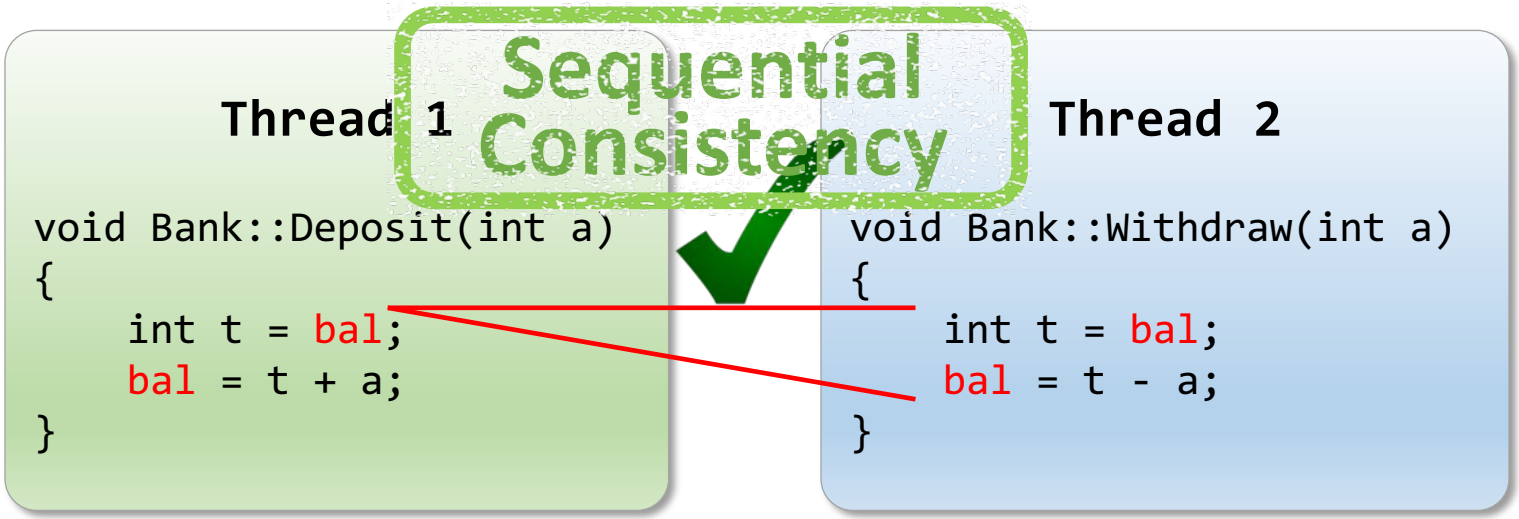
```
void Bank::Deposit(int a)
{
    int t = bal;
    bal = t + a;
}
```

Thread 2

```
void Bank::Withdraw(int a)
{
    int t = bal;
    bal = t - a;
}
```

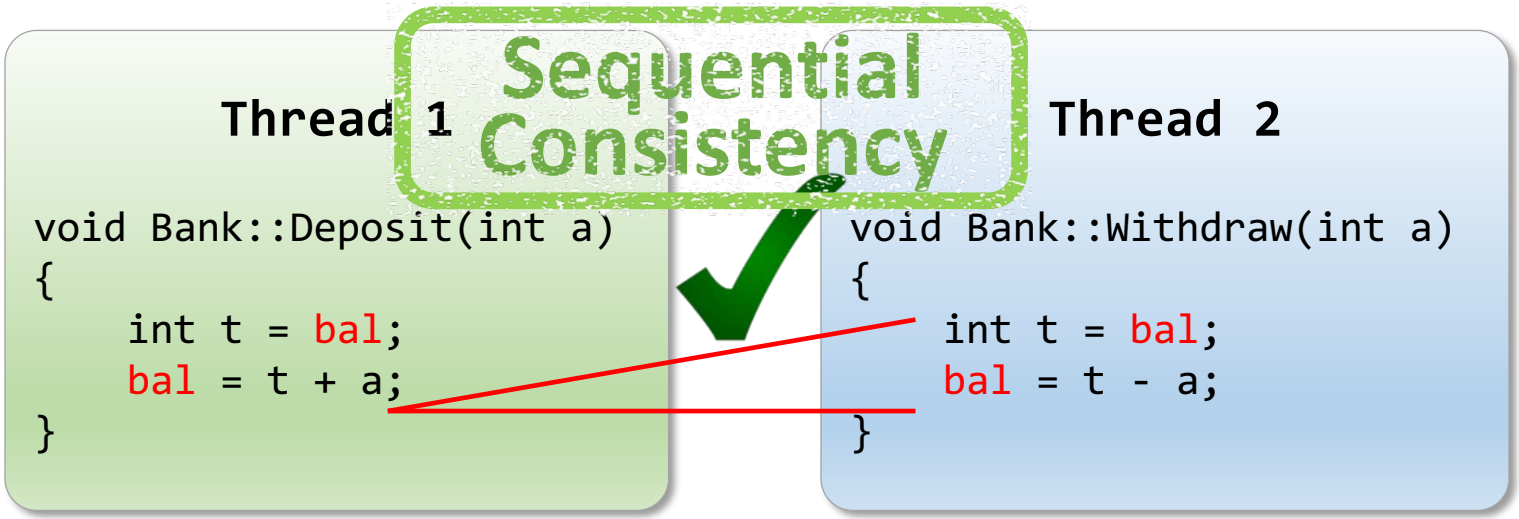
Potential data Race

- Code is supposed to execute atomically
  - Multiple dependent instructions manipulate some shared data
- Interleaving with instructions of another thread that access the same data



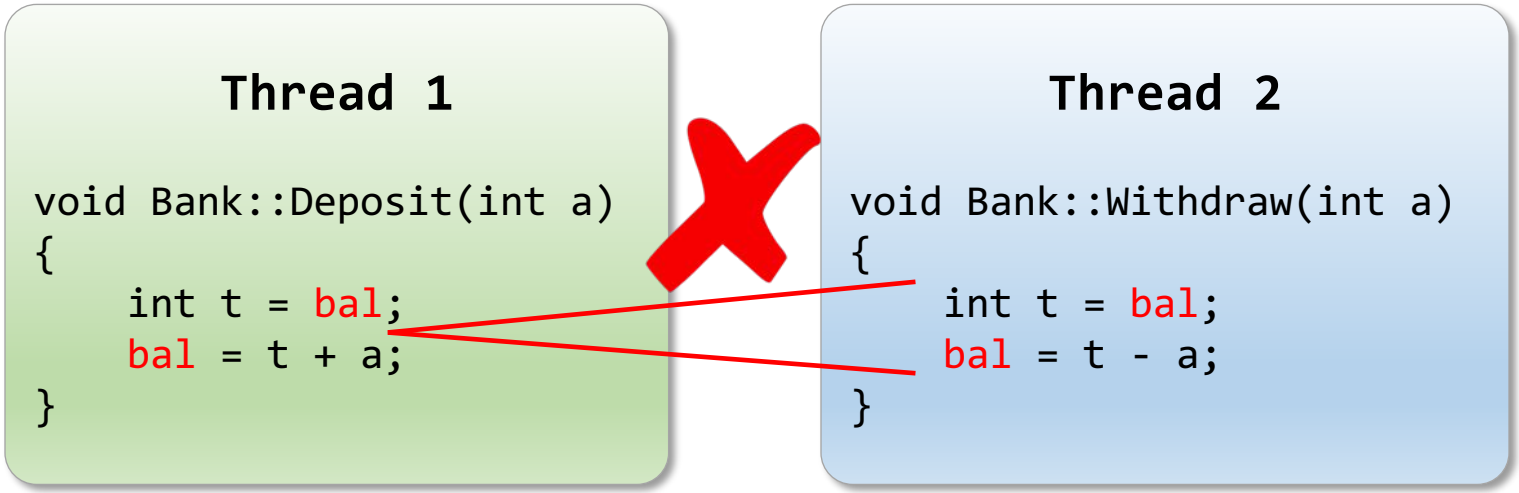
Potential data Race

- Code is supposed to execute atomically
  - Multiple dependent instructions manipulate some shared data
- Interleaving with instructions of another thread that access the same data



Data Race

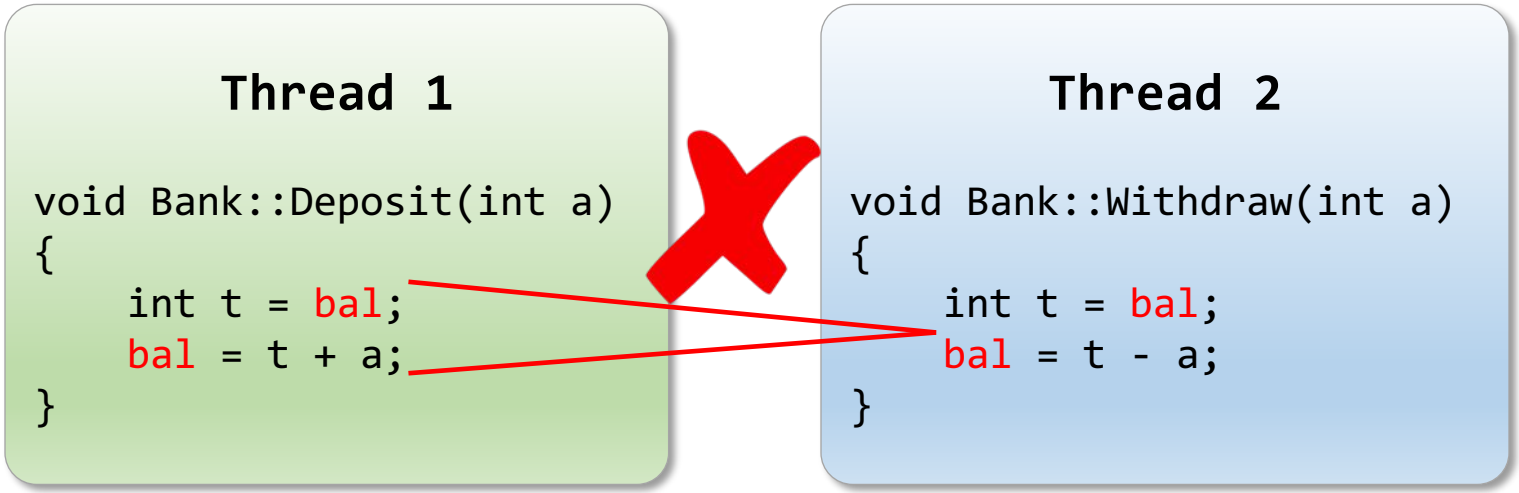
- Code is supposed to execute atomically
  - Multiple dependent instructions manipulate some shared data
- Interleaving with instructions of another thread that access the same data



The withdraw is not reflected in the final balance!

Data Race

- Code is supposed to execute atomically
  - Multiple dependent instructions manipulate some shared data
- Interleaving with instructions of another thread that access the same data



The deposit is not reflected in the final balance!

Ordering Violation

- Missing or incorrect synchronization between two processes
  (e.g., a producer and a consumer)

Thread 1

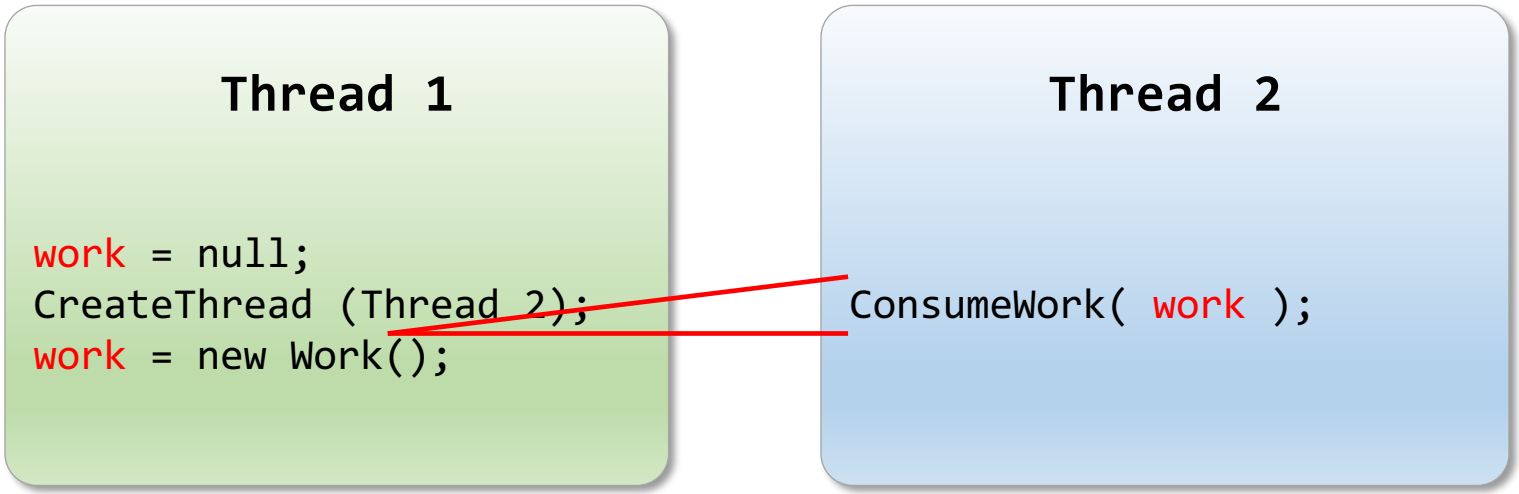
```
work = null;
CreateThread (Thread 2);
work = new Work();
```

Thread 2

```
ConsumeWork( work );
```

Ordering Violation

- Missing or incorrect synchronization between two processes
  (e.g., a producer and a consumer)



'work' is not initialized yet!

Unintended Sharing

- Processes accidentally share data
  - 'work()' is executed by two threads concurrently

```
void work() {
   static int local = 0;
   …
   local += 10;
   printf("%d", local);
}
```

Thread 1

```
…
work();
…
```

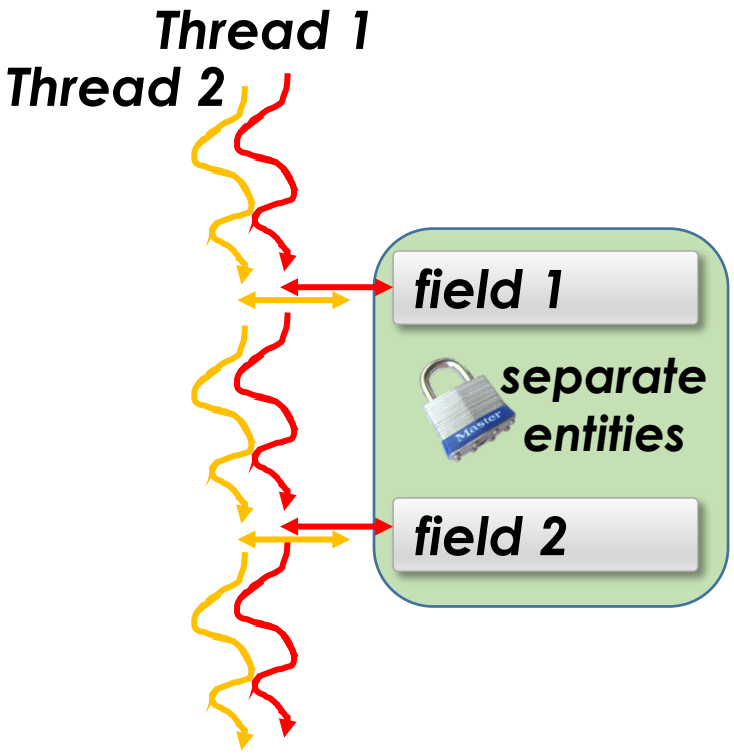
Thread 2

```
…
work();
…
```

Output is "10" and "20"

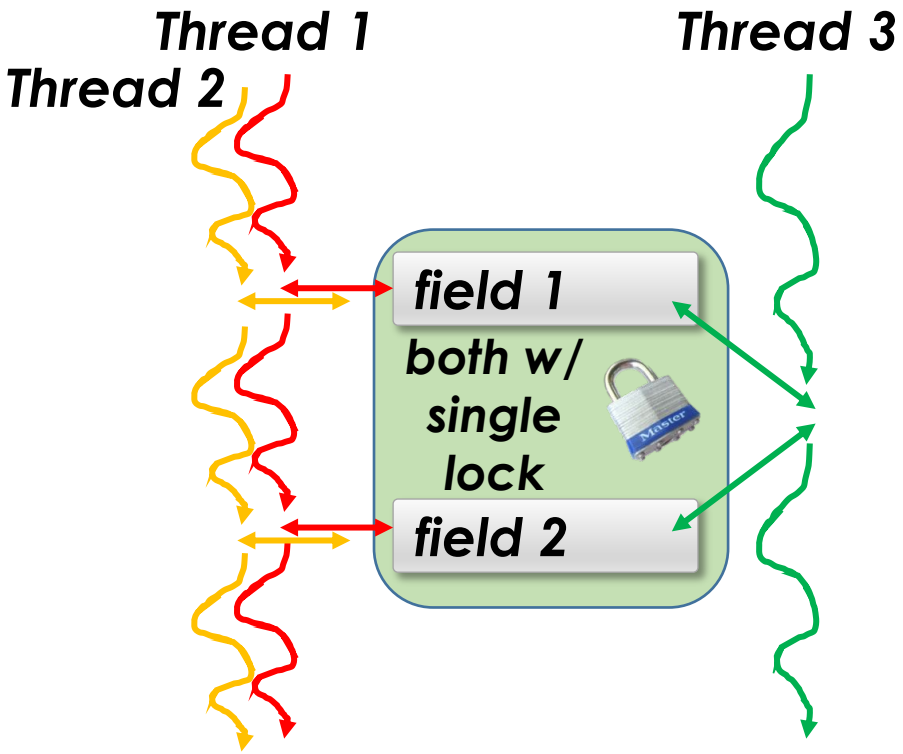
High-Level Data Race

- Wrongly defined atomic blocks



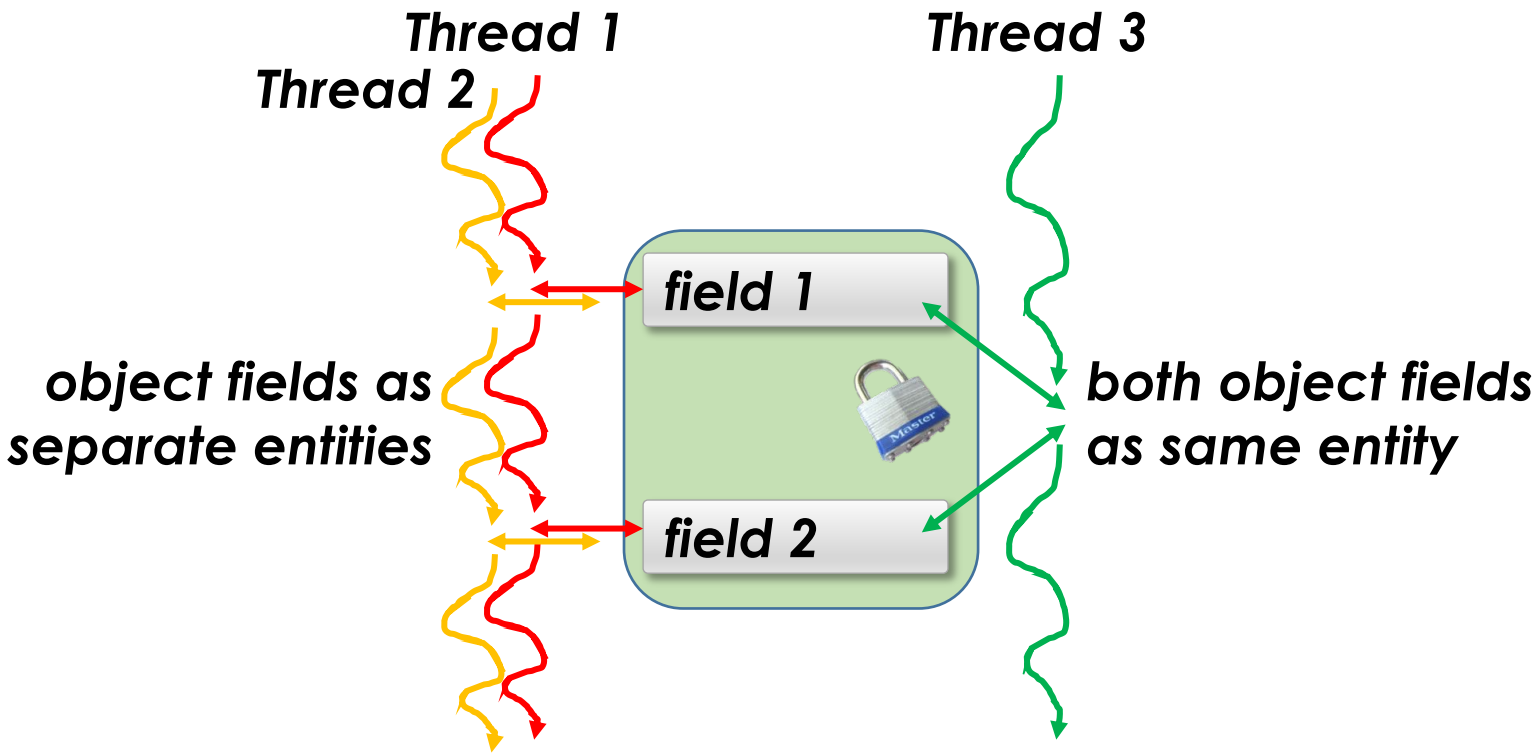
High-Level Data Race

- Wrongly defined atomic blocks



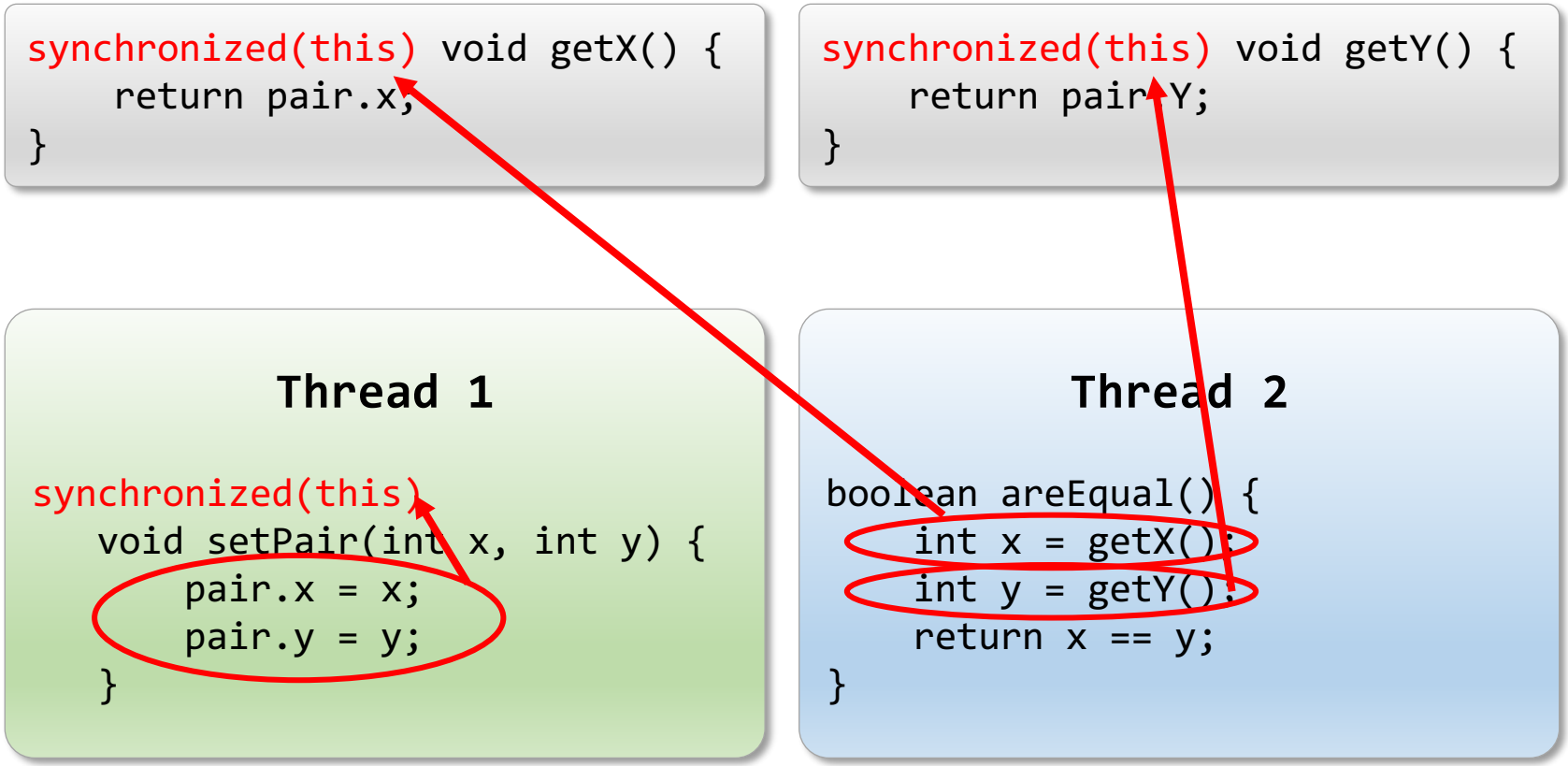
High-Level Data Race

- Wrongly defined atomic blocks



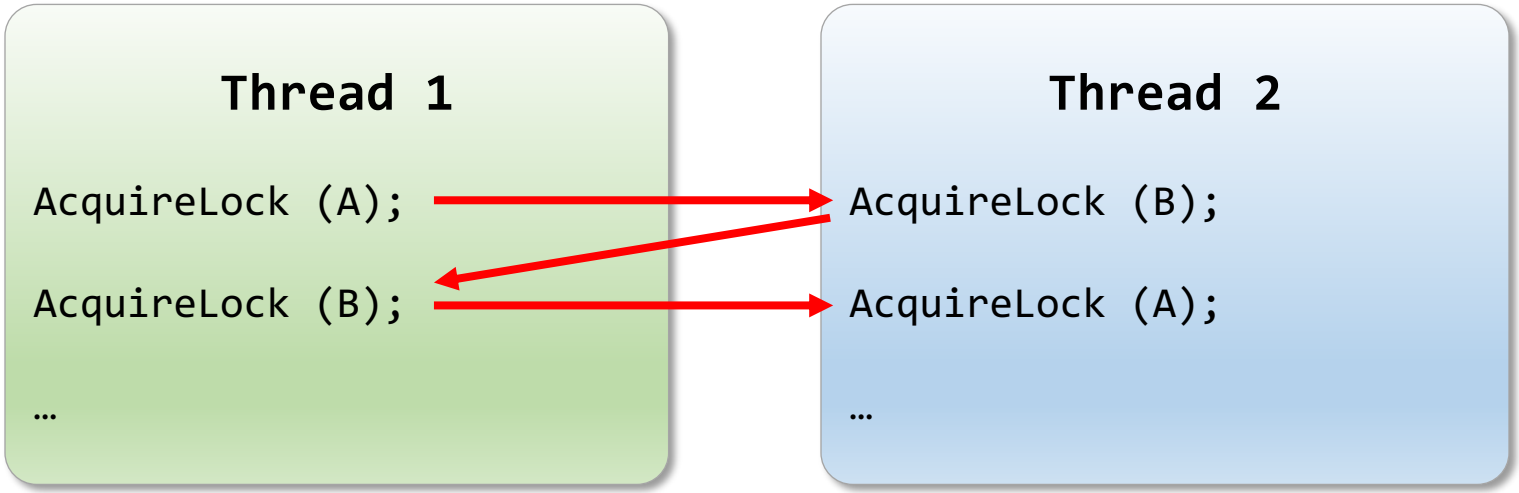
High-Level Data Race

- Wrongly defined atomic blocks



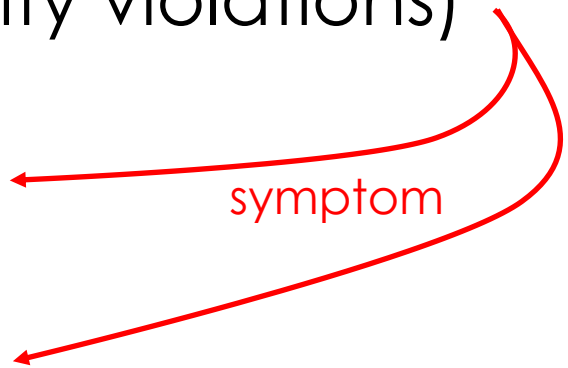
Deadlock

- Processes are waiting forever for each other



Common Concurrency Errors

- Data races (atomicity violations)
- Ordering violations
- Unintended sharing
- High-level atomicity violations
- Deadlocks and livelocks



Common Concurrency Errors

- Data races (atomicity violations)
- Ordering violations
- Unintended sharing
- High-level atomicity violations
- Deadlocks and livelocks

Concurrency Errors

Data Races

What is a Data Race?

- Two conflicting memory accesses happening concurrently

What is a Data Race?

- Two **conflicting memory accesses** happening concurrently

What is a Data Race?

- Two conflicting memory accesses **happening concurrently**

What is a Data Race?

- Two **conflicting memory accesses** happening concurrently
- Conflicting memory accesses means:
  - They access the same memory location

What is a Data Race?

- Two **conflicting memory accesses** happening concurrently
- Conflicting memory accesses means:
  - They access the same memory location
  - At least one is an update (write)

- Write — Write
- Write — Read
- Read — Write
- Read — Read

What is a Data Race?

- Two conflicting memory accesses **happening concurrently**

What means "Happens Concurrently"?

- Two events A and B happen concurrently if both

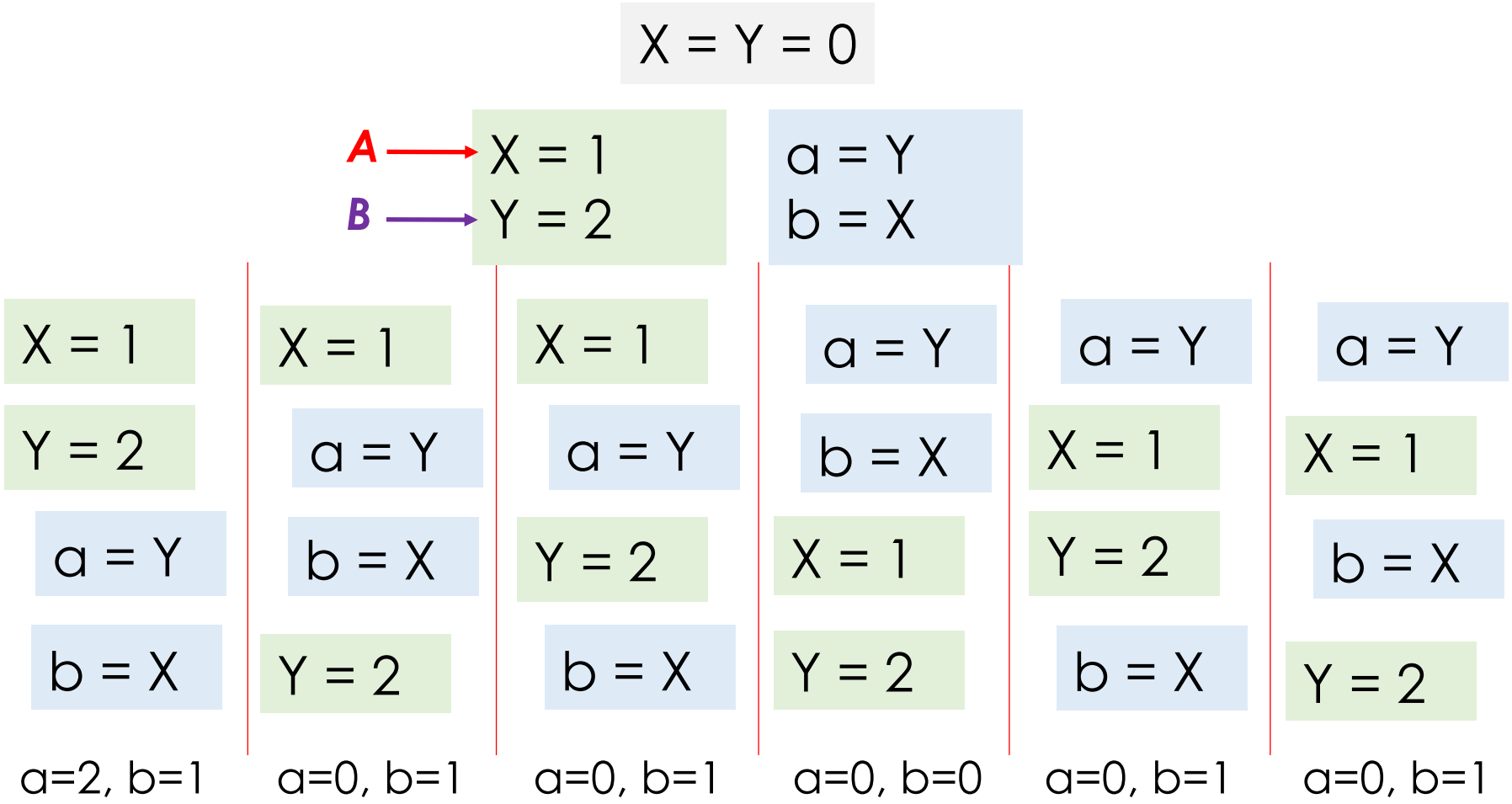
  A, B

  and

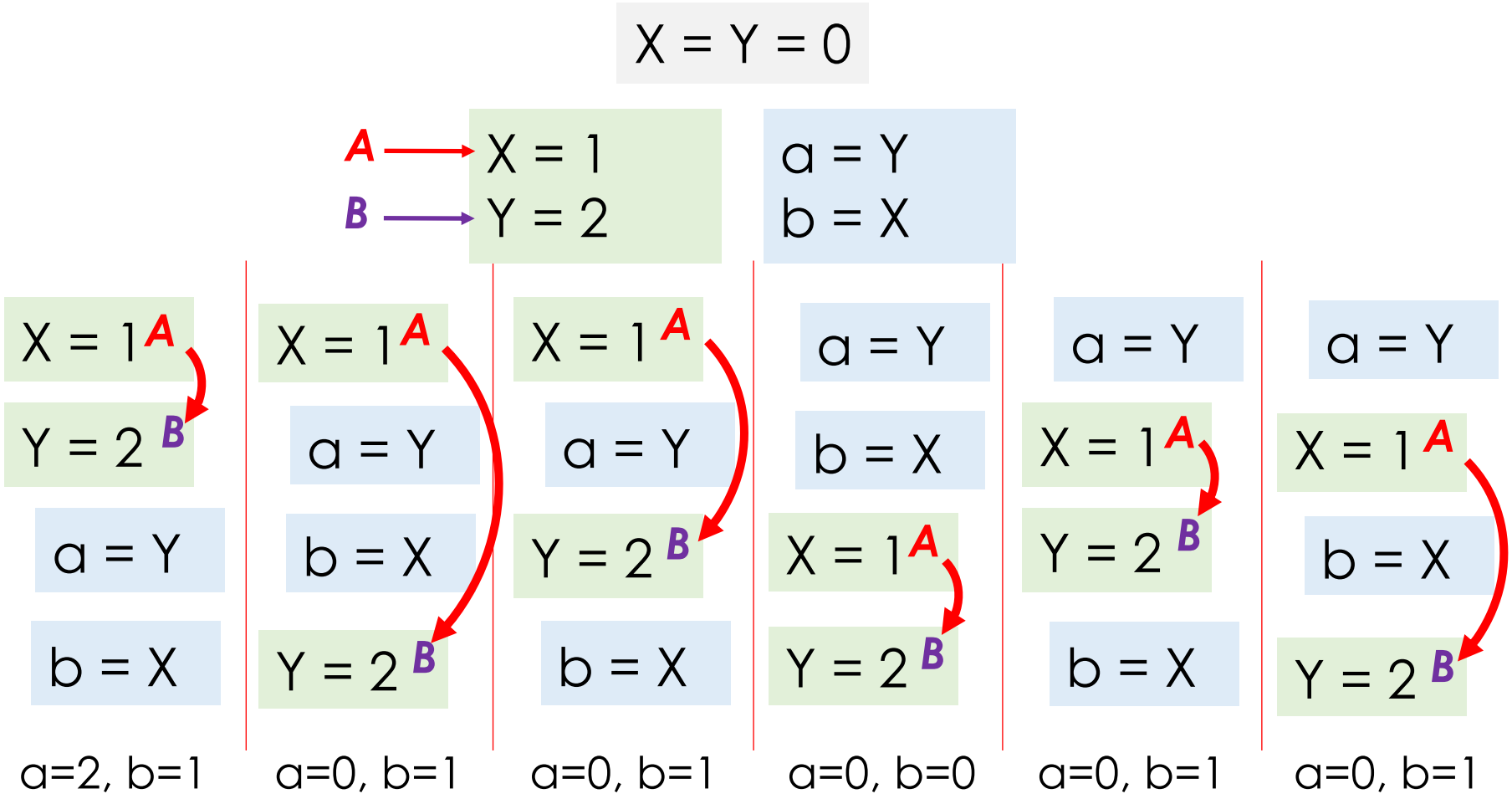
  B, A

  are possible sequentially consistent executions of those events

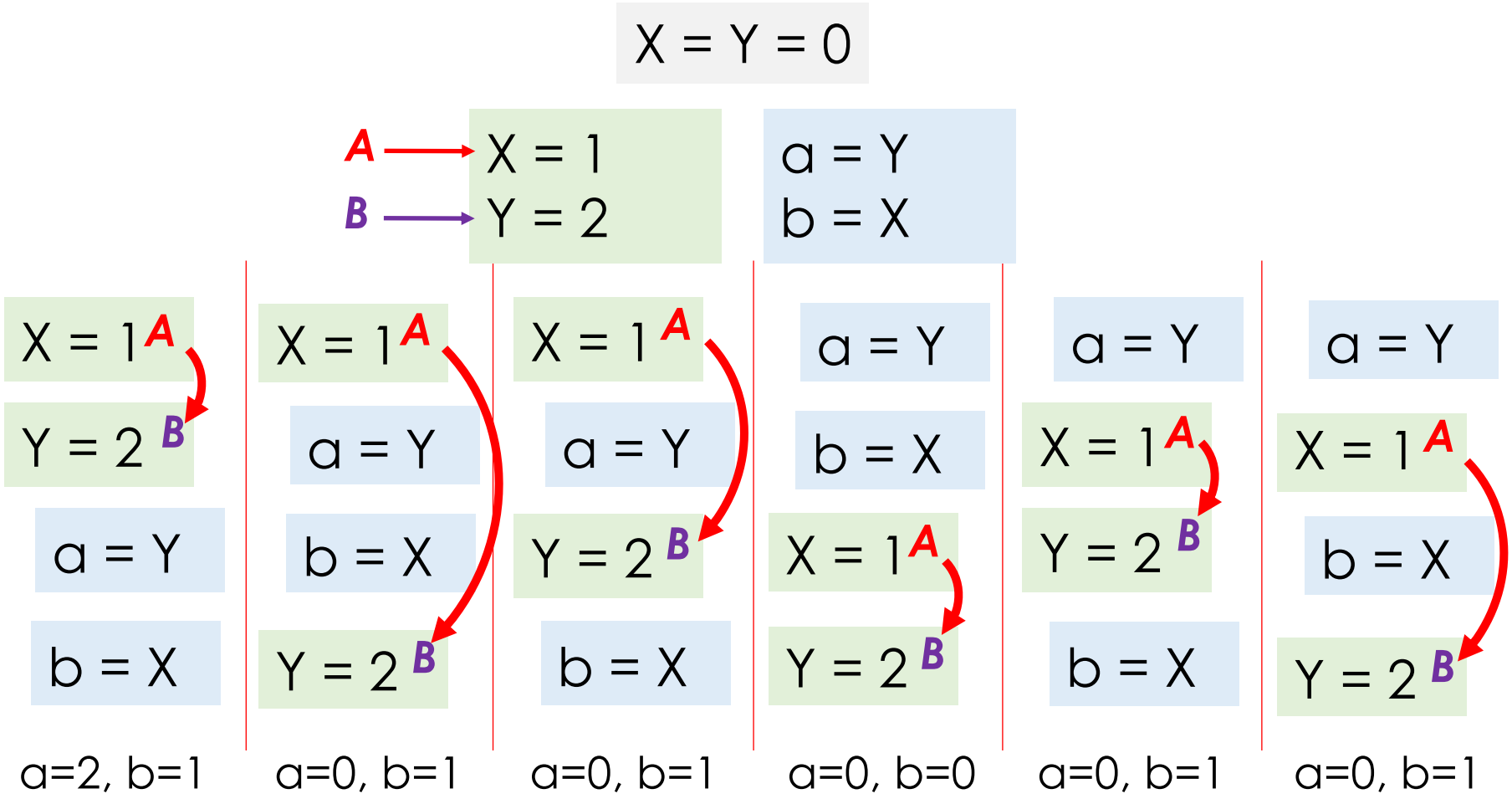
Assigning Semantics to Concurrent Programs



Assigning Semantics to Concurrent Programs

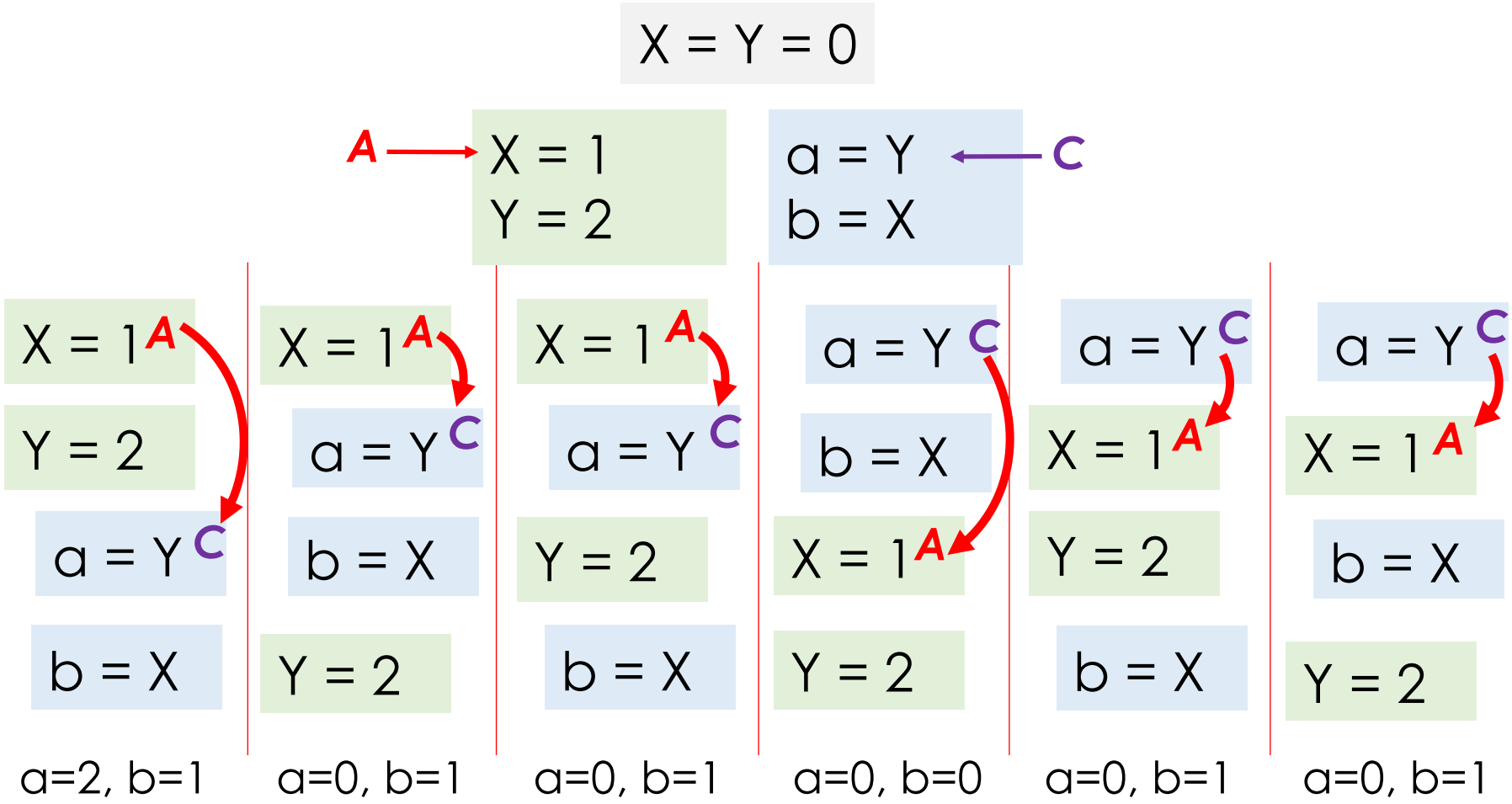


Assigning Semantics to Concurrent Programs



Order is always "A, B". Events 'A' and 'B' ***are not*** *concurrent!*

Assigning Semantics to Concurrent Programs



Order may be "A, C" or "C, A". Events 'A' and 'C' ***are*** *concurrent!*

Question

x is a shared variable, initially 0

Q: Knowing that processes A and B execute concurrently, what are the possible values for 'x' after both processes terminate?

Process A

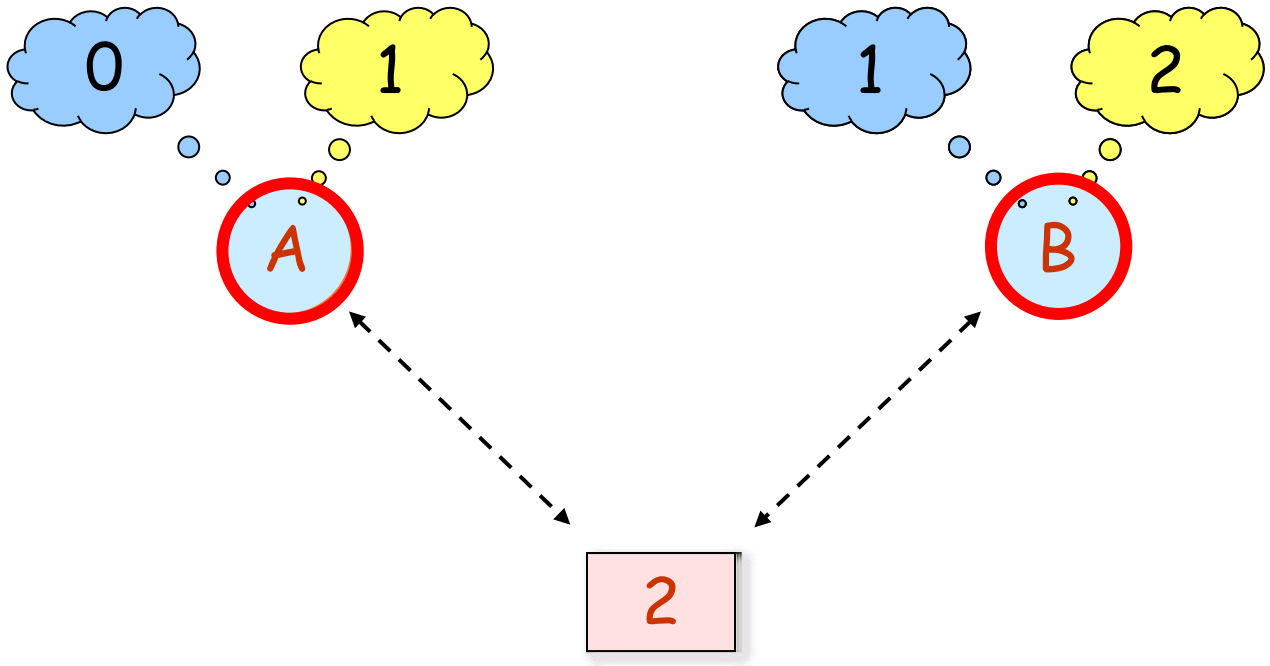
```
for (i = 0; i < 5;  i++) {
    x = x + 1
}
```

Process B

```
for (j = 0; j < 5;  j++) {
    x = x + 1
}
```

A) Any value in the range 2 to 10
B) Any value in the range 5 to 10
C) Always 10

The smallest value is 2



How to Detect a Data Race?

- Two concurrent accesses to a shared memory location
- At least one of them is a write

- How to monitor memory accesses?
- How to detect if two accesses are (or may be) concurrent?

Concurrency Errors

Data Race Detection

Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants

- Example Tools:
  - RCC/Java — type-based
  - ESC/Java — "functional verification" (theorem proving-based)

Static Data Race Detection

- Advantages:
  - Reason about all inputs/interleavings
  - No run-time overhead
  - Adapt well-understood static-analysis techniques
  - Annotations to document concurrency invariants

- Disadvantages of static:
  - Tools produce "false positives" and/or "false negatives"
  - May be slow, require programmer annotations
  - May be hard to interpret results
  - May not scale to large or complex programs

Dynamic Data Race Detection

- Advantages
  - Soundness
    - Every actual data race is reported (for the current *run*)
  - Completeness
    - All reported warnings are actually races (avoid "false positives")

- Disadvantages
  - Run-time overhead (5-20x slower for best tools)
  - Memory overhead for analysis state
  - Reasons only about observed executions
    - sensitive to test coverage
    - (some generalization possible...)

Approaches

- Happens-Before

- Lock-set algorithm
  - Learns which shared memory locations are protected by which locks
  - Issues warning if finds no lock protects a shared memory location

Concurrency Errors

Data Race Detection Using Happens-before [Lamport '78]

Lock Definition

- **Lock**: a synchronization object that is either available (free), or owned (by a thread)
  - Operations: **lock(mu)** and **unlock(mu)**
    - (*We are assuming no explicit initialize operation*)
  - A lock can only be unlocked by its current owner
  - The **lock()** operation is blocking if the lock is owned by another thread

The Happens-before Relation

- *happens-before* defines a partial order for events in a set of concurrent threads
  - In a single thread, *happens-before* reflects the temporal order of event occurrence
  - Between threads, **A** happens before **B** if A is a unlock access in one thread, and the next access to that lock (event B) is in a **different** thread, and if the accesses obey the semantics of the lock (can't have two successive locks, or two successive unlocks, or a lock in one thread and an unlock in a different thread)

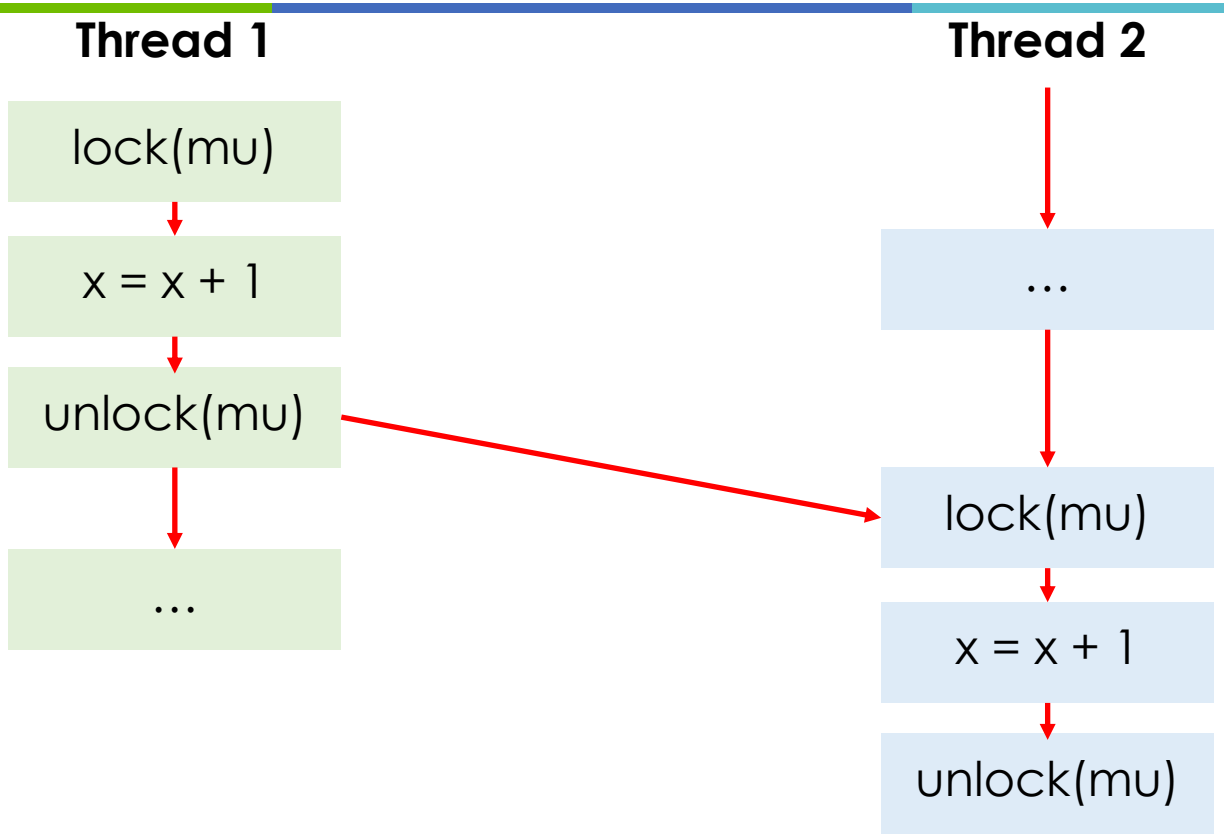
The Happens-before Relation

- Let **event a** be in thread 1 and **event b** be in thread 2

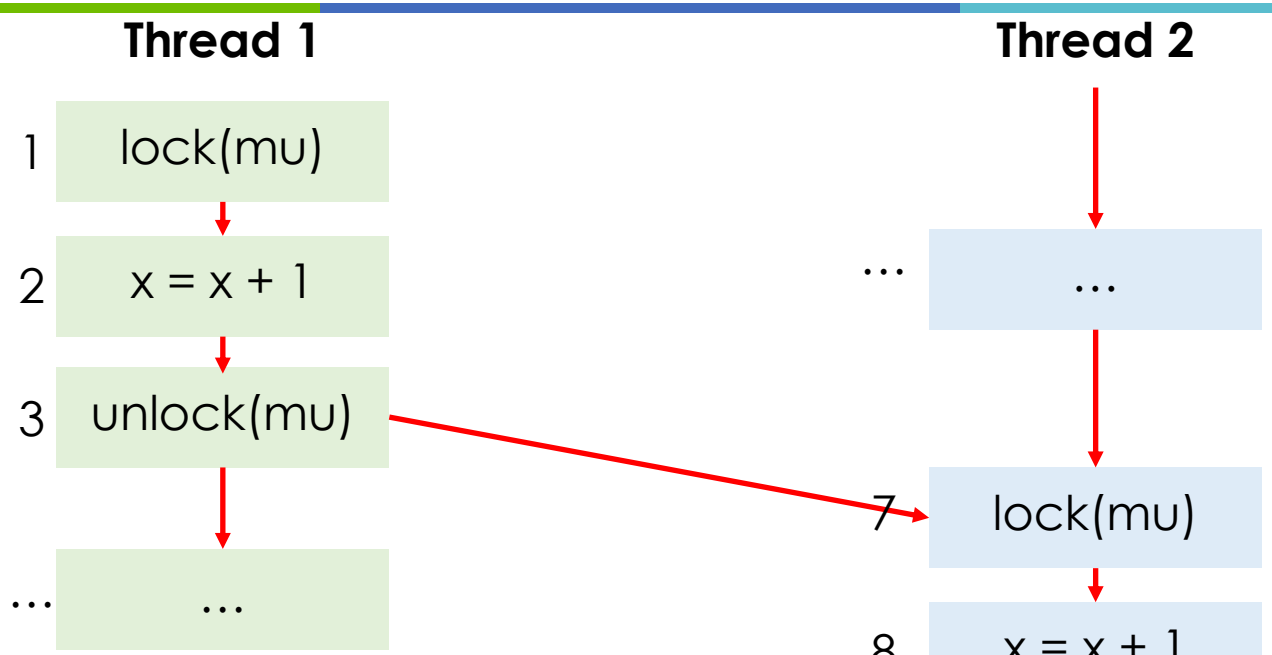
If a = unlock(mu) and b = lock(mu) then
a → b (a happens-before b)

Data races between threads are ***possible*** if accesses to shared variables are not ordered by the *happens-before* relation

Example 1



Example 1



ev1 → *ev2* → *ev3*

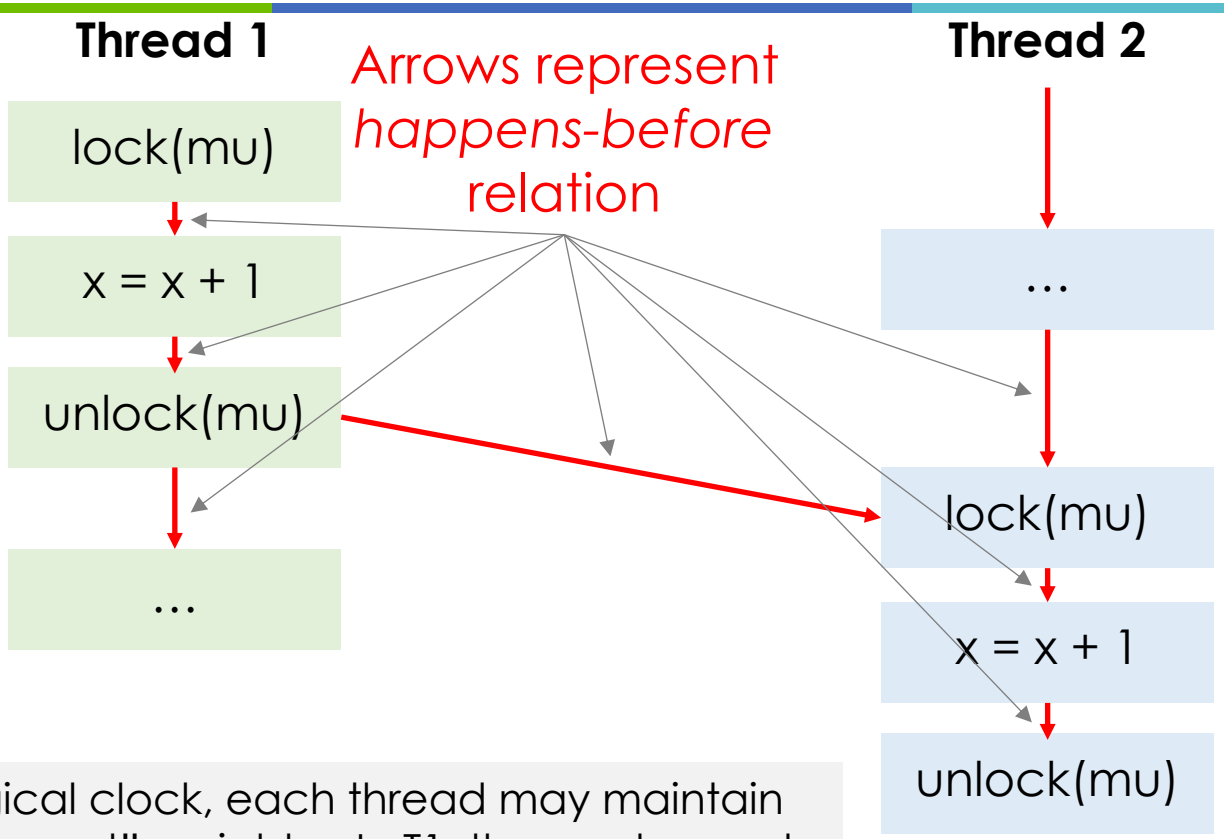
ev7 → *ev8* → *ev9*

ev3 → *ev7*

ev2 → *ev8 (by transitivity)*

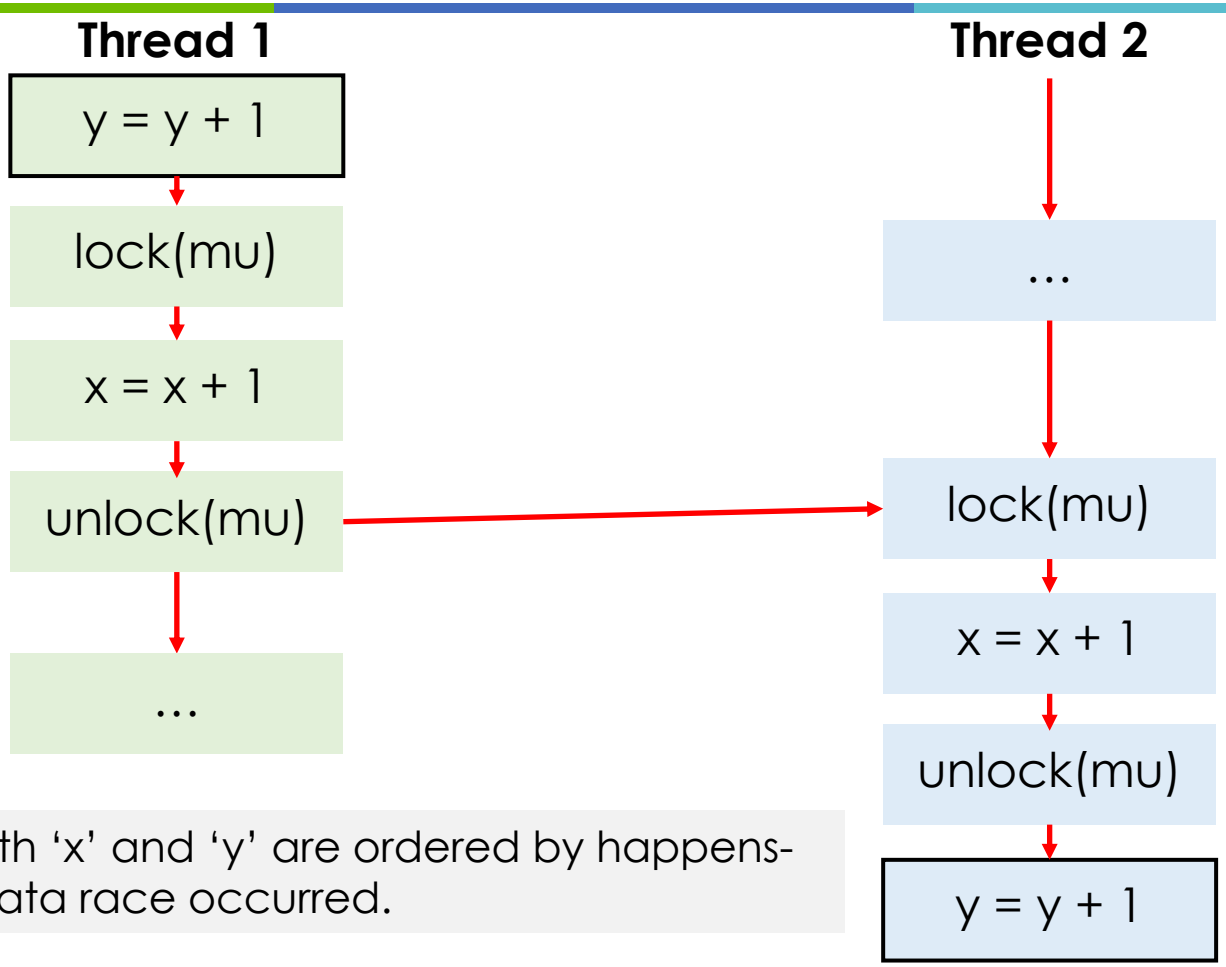
No Data Race

Example 1



Instead of a logical clock, each thread may maintain a "most recent event" variable. In T1, the most recent event is unlock(mu); when T2 executes lock(mu) the system can establish the happens-before relation.

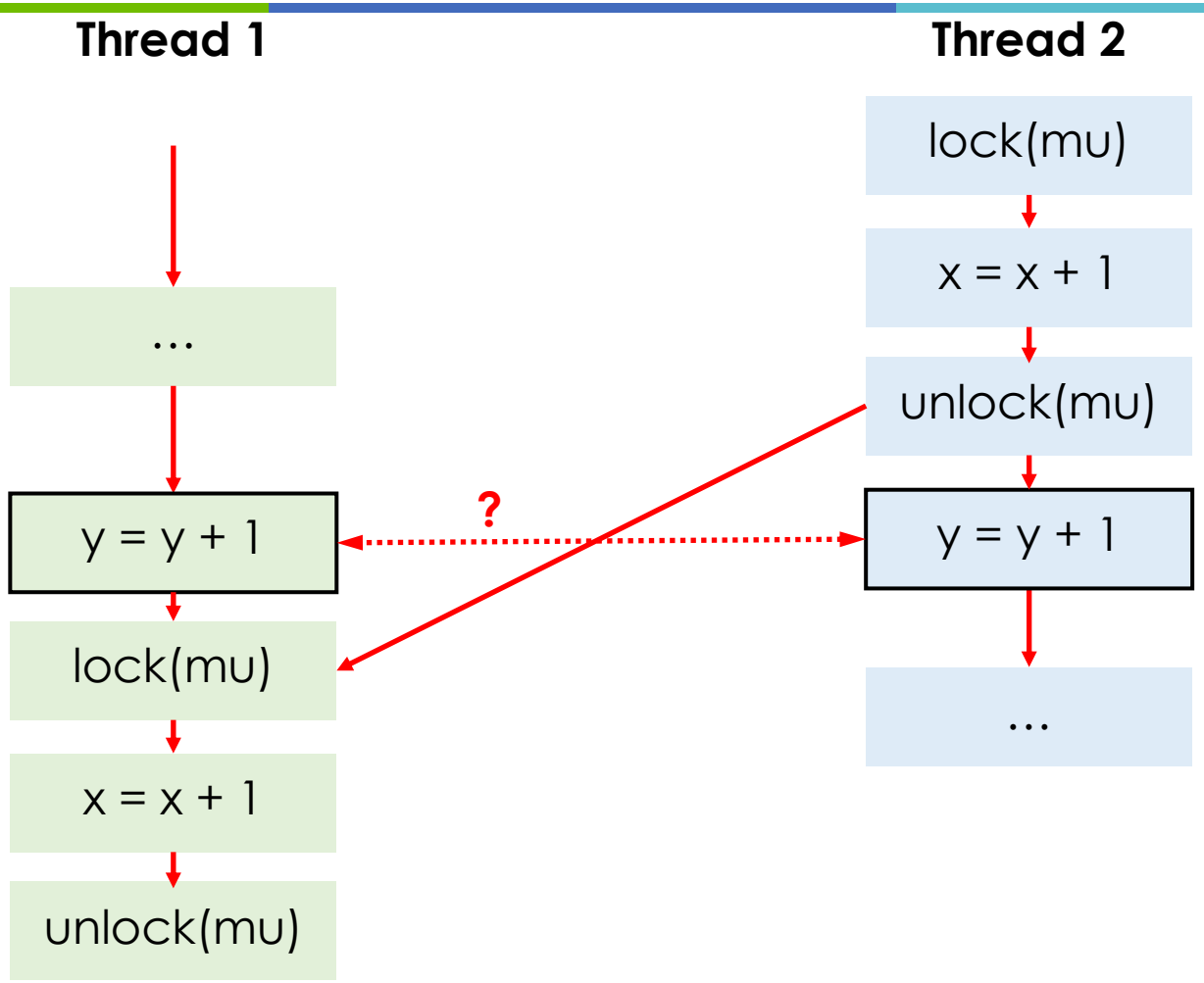
Example 2



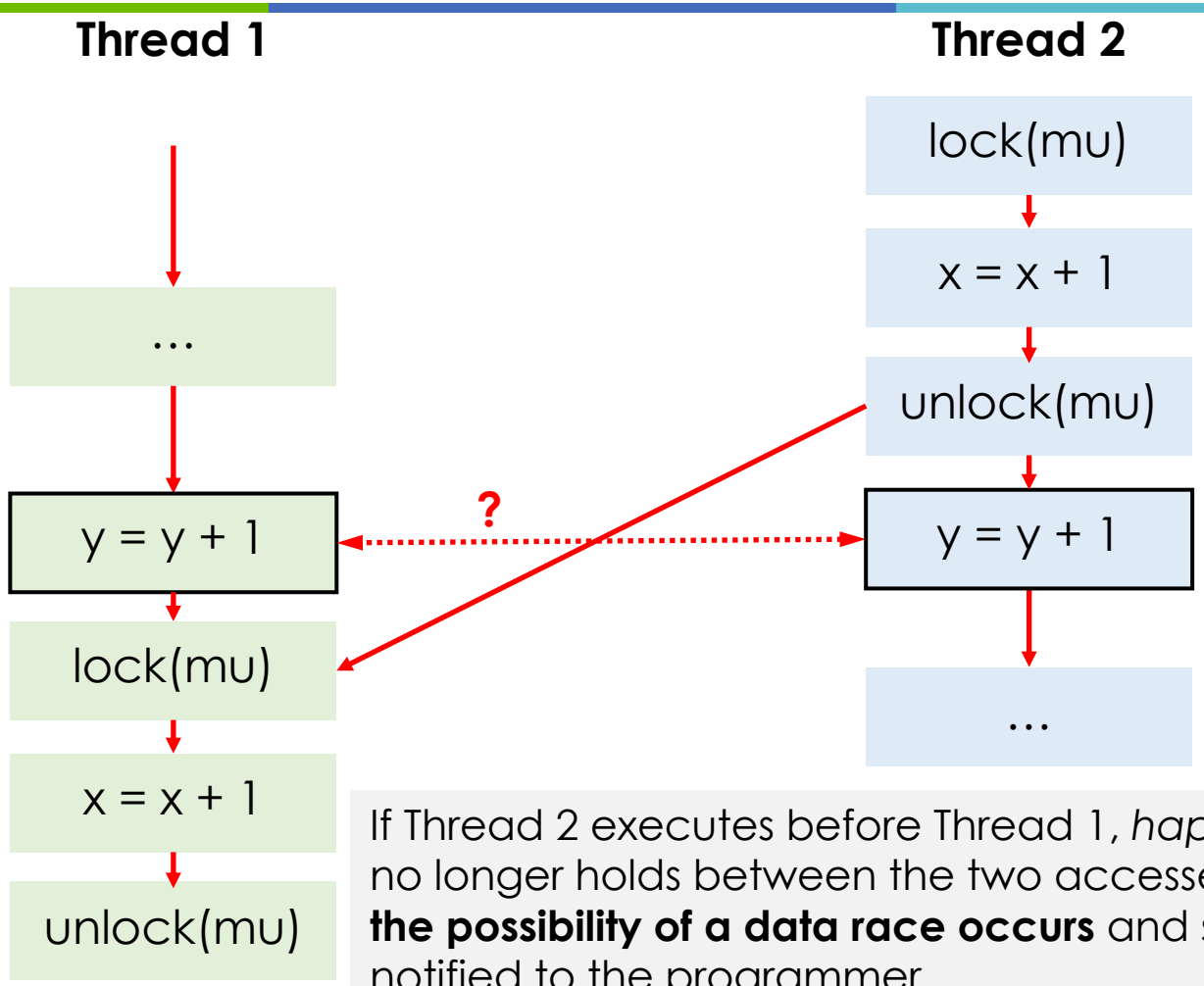
Accesses to both ‘x’ and ‘y’ are ordered by happens-before, so no data race occurred.

But … a different execution ordering could get different results?

Example 3


Thread 1
Thread 2
lock(mu)
x = x + 1
unlock(mu)
...
y = y + 1
?
y = y + 1
lock(mu)
...
x = x + 1
unlock(mu)

Example 3



If Thread 2 executes before Thread 1, *happens-before* no longer holds between the two accesses to 'y', so **the possibility of a data race occurs** and should be notified to the programmer.

Happens Before (summary)

- Happens-before defines a partial order for events in a set of concurrent threads
- Data races are possible if accesses to shared variables are not ordered by the happens-before relation
- Happens-before only detects data races if the incorrect order shows up in an execution trace

Concurrency Errors

The Lock-Set Algorithm — Eraser [Savage et.al. '97]

Approaches

- Checks a sufficient condition for data-race freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure are made while holding the lock

Thread 1

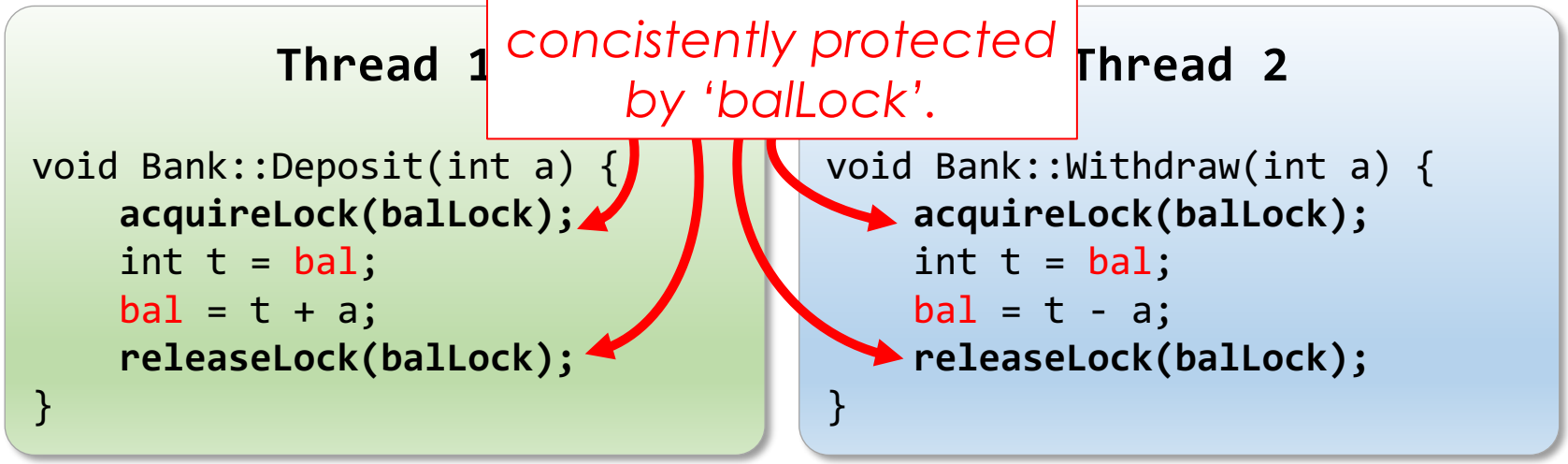
```
void Bank::Deposit(int a) {
    acquireLock(balLock);
    int t = bal;
    bal = t + a;
    releaseLock(balLock);
}
```

Thread 2

```
void Bank::Withdraw(int a) {
    acquireLock(balLock);
    int t = bal;
    bal = t - a;
    releaseLock(balLock);
}
```

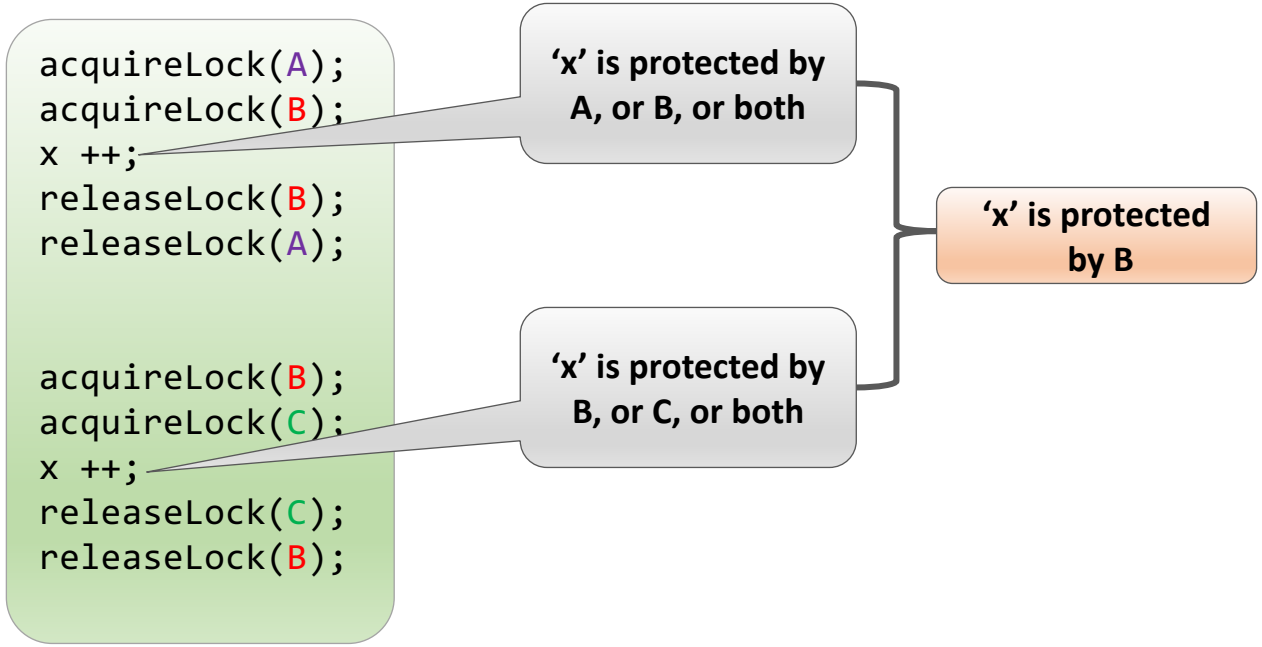
Approach

- Checks a sufficient condition for data-race freedom
- Consistent locking discipline
  - Every data structure is protected by a single lock
  - All accesses to the data structure are made while holding the lock



Approach

- How to know which locks protect each memory location?
  - Ask the programmer? Cumbersome!
  - Infer from the program code? Is it effective?



The Lock-Set Algorithm

- Two data structures:
  - LocksHeld(t) ≈ set of locks held currently by thread t
    - Initially set to Empty
  - LockSet(x). ≈ set of locks that could potentially be protecting x
    - Initially set to the universal set (all lock IDs)
- When thread ‘t’ acquires lock ‘l’
  - LocksHeld(t) = LocksHeld(t) ∪ {l}
- When thread ‘t’ releases lock ‘l’
  - LocksHeld(t) = LocksHeld(t) \ {l}
- When thread ‘t’ accesses location ‘x’
  - LockSet(x) = LockSet(x) ∩ LocksHeld(t)
- “Data race” warning if LockSet(x) becomes empty

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)		
lock(m2)		
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	**{m1}**	{m1, m2}
lock(m2)		
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	**{m1, m2}**	{m1, m2}
v = v + 1		
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

Another Example

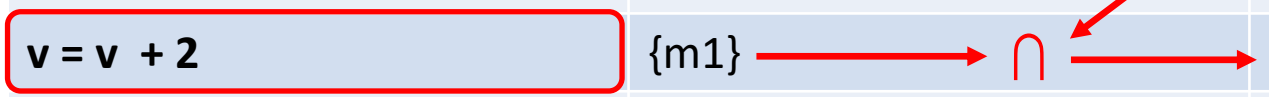
Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2} → ∩ →	**{m1, m2}**
unlock(m2)		
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	**{m1}**	{m1, m2}
v = v + 2		
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		

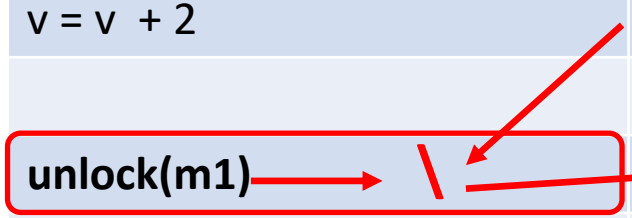
Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1} ∩	**{m1}**
unlock(m1)		
lock(m2)		
v = v + 1		
unlock(m2)		



Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	**{ }**	{m1}
lock(m2)		
v = v + 1		
unlock(m2)		



Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2) → U	**{m2}**	{m1}
v = v + 1		
unlock(m2)		

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)	{m2}	{m1}
v = v + 1	{m2} ∩	**{ }**
unlock(m2)		

Another Example

Program Code	LocksHeld	LockSet
	{ }	{m1, m2}
lock (m1)	{m1}	{m1, m2}
lock(m2)	{m1, m2}	{m1, m2}
v = v + 1	{m1, m2}	{m1, m2}
unlock(m2)	{m1}	{m1, m2}
v = v + 2	{m1}	{m1}
unlock(m1)	{ }	{m1}
lock(m2)	{m2}	{m1}
v = v + 1		**{ } — ALARM**
unlock(m2)		

Lockset Algorithm Guarantees

- No warnings => no data races on the current execution
  - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
  - Thread-local initialization or Bad locking discipline

Lockset Algorithm Guarantees

- No warnings => no data races on the current execution
  - The program followed consistent locking discipline in this execution
- Warnings does not imply a data race
  - Thread-local initialization or **Bad locking discipline**

Thread 1

```
acquireLock(m1);
acquireLock(m2);
x = x + 1;
releaseLock(m2);
releaseLock(m1);
```

Thread 2

```
acquireLock(m2);
acquireLock(m3);
x = x + 1;
releaseLock(m3);
releaseLock(m2);
```

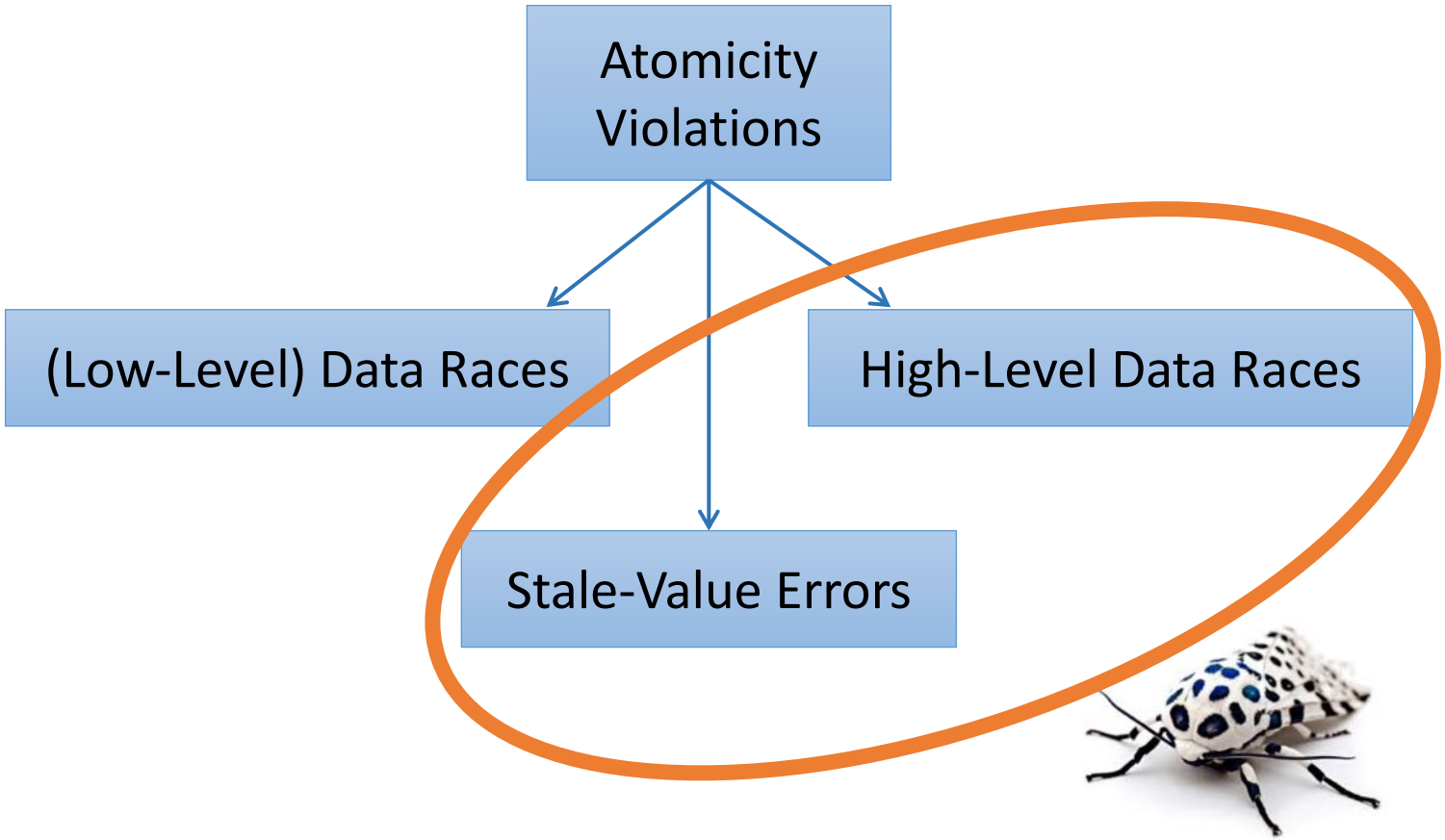
Thread 3

```
acquireLock(m1);
acquireLock(m3);
x = x + 1;
releaseLock(m3);
releaseLock(m1);
```

Concurrency Errors

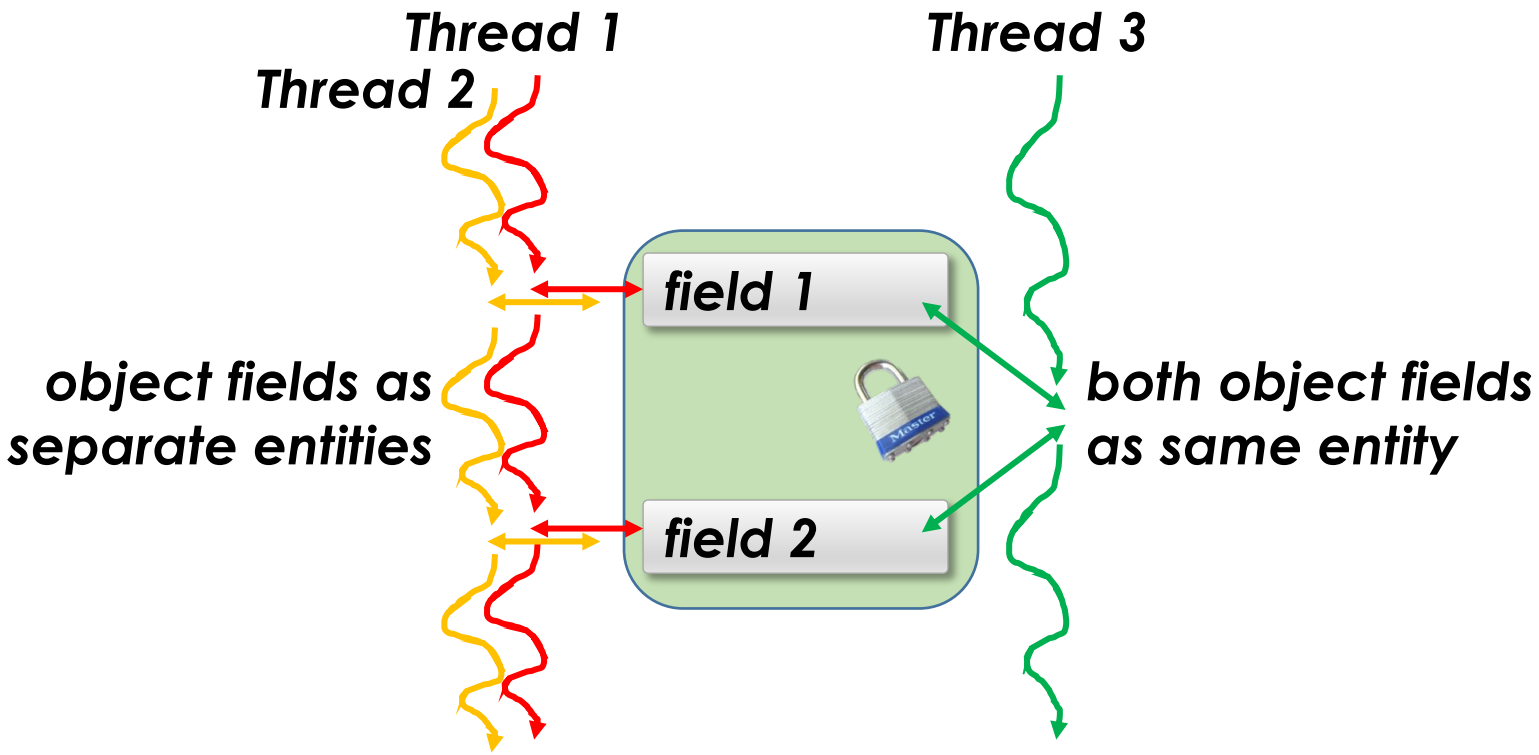
Detection of High-level Data Races and Stale-value Errors [Artho03, Dias12]

Concurrency Anomalies



High-Level Data Race

- Wrongly defined atomic blocks

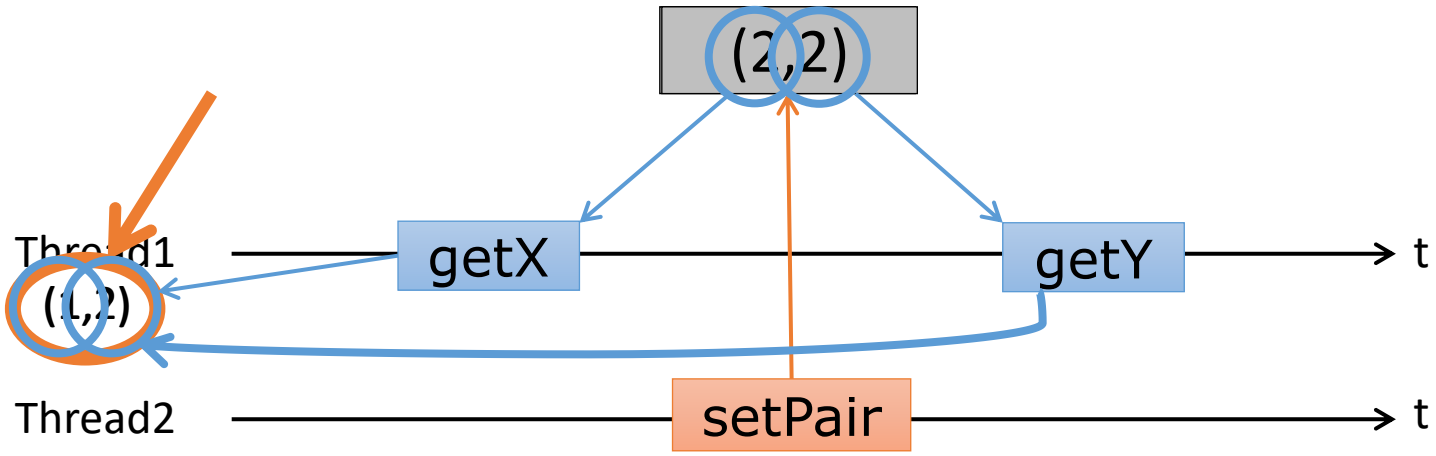


High-Level Data Races

Shared variables: x, y

```
// Thread 1
public boolean equals(){
  int loc_x = getX(); // synchr
  int loc_y = getY(); // synchr
  return loc_x == loc_y;
}
```

```
// Thread 2
public synchronized
int setPair(int v1, int v2){
    x = v1;
    y = v2;
}
```



View of an Atomic Block [Artho03]

- A view of an atomic block B — V(B) — if the set of variables accessed inside the atomic code block B

View of an Atomic Block

- A view of an atomic block B — V(B) — if the set of variables accessed inside the atomic code block B
- The *read view* of B — $V_R(B) \subseteq V(B)$ — is the set of variables read inside the atomic code block B
- The *write view* of B — $V_W(B) \subseteq V(B)$ — is the set of variables written inside the atomic code block B

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
public synchronized
  void incX(){
    int local = x;
    setX(local + 1)
  }

public void setX(int aux){
  x = aux;
}
```

V(incX) = ?

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

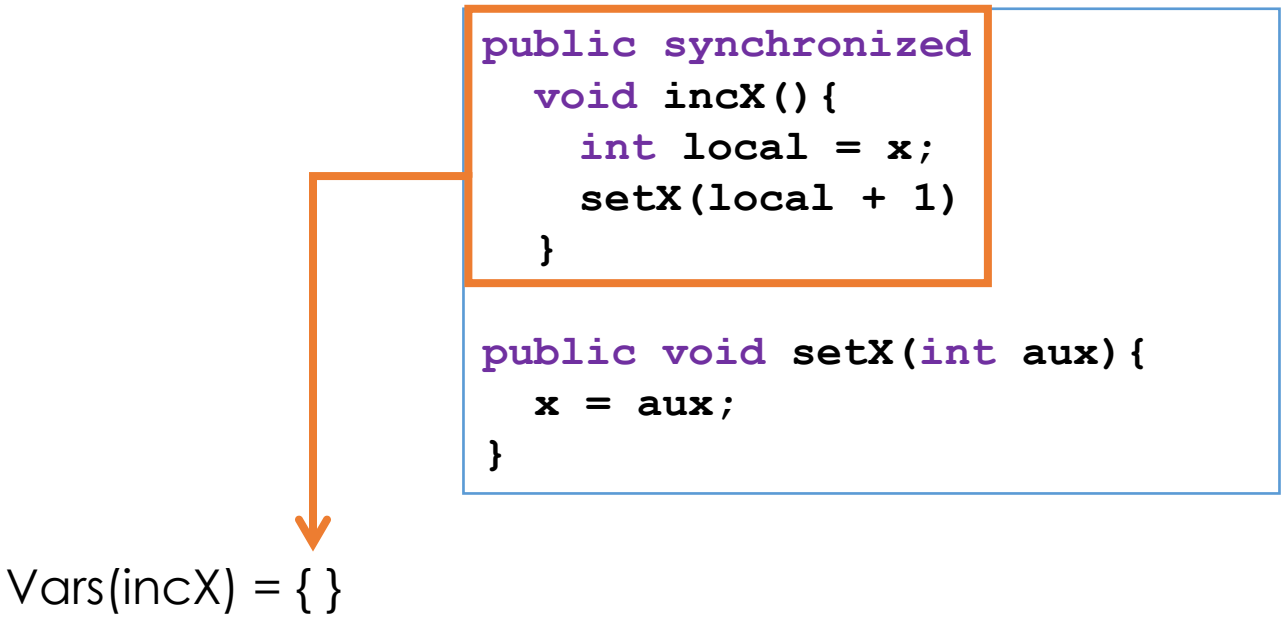
```
public synchronized
  void incX(){
    int local = x;
    setX(local + 1)
  }

public void setX(int aux){
  x = aux;
}
```

Which (shared) variables are accessed and how?

Views Analysis [Artho03]

- View: set of shared variables accessed atomically



Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)}

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ Vars(setX)

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ Vars(setX)

Vars(incX) = { }

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ Vars(setX)

Vars(incX) = { } ∪ {write(X)}

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ Vars(setX)

Vars(incX) = {write(X)}

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ Vars(setX)

Vars(incX) = {write(X)}

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = { } ∪ {read(X)} ∪ {write(X)}

Views Analysis [Artho03]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = {read(X), write(X)} V(incX) = { X }

Views Analysis [Dias12]

- View: set of shared variables accessed atomically

```
@Atomic
public void incX(){
    int local = x;
    setX(local + 1)
}

public void setX(int aux){
        x = aux;
}
```

Vars(incX) = {read(X), write(X)}

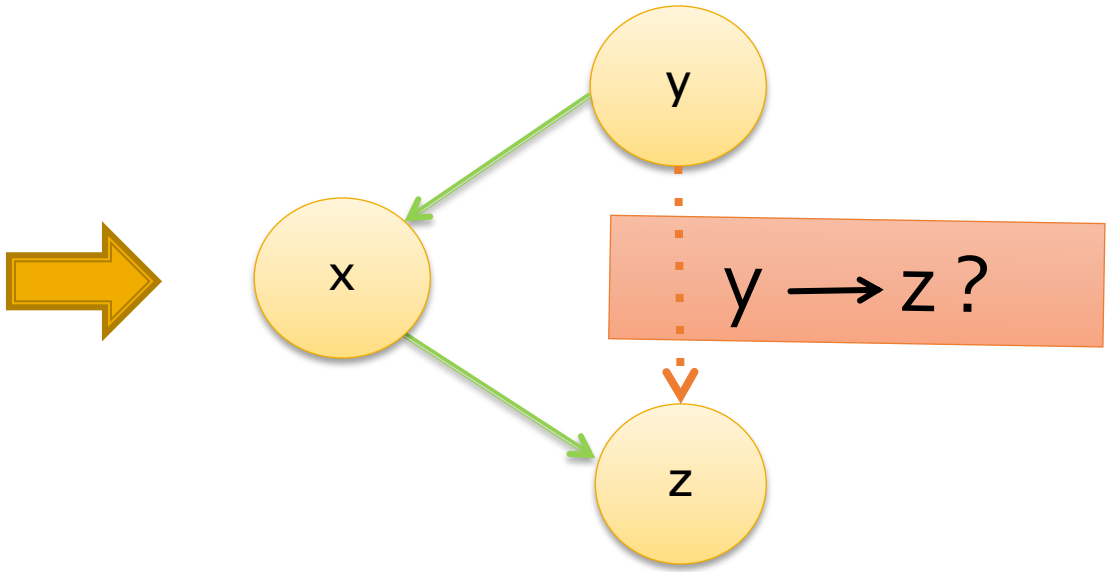
V(incX) = { X }

V_R(incX) = { X }

V_W(incX) = { X }

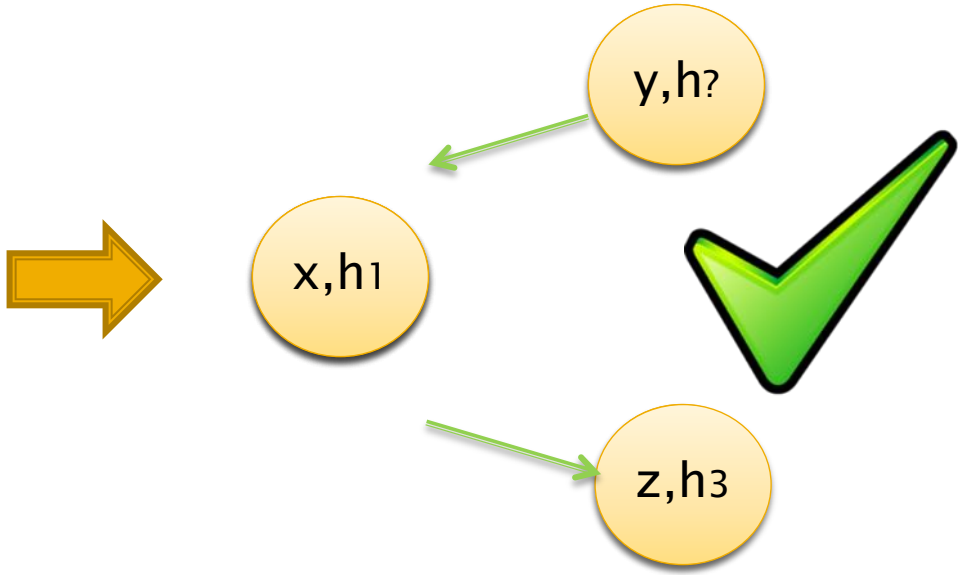
Data Dependency Analysis

```
h1: x = y;
h2: x = 2;
h3: z = x;
```



Data Dependency Analysis

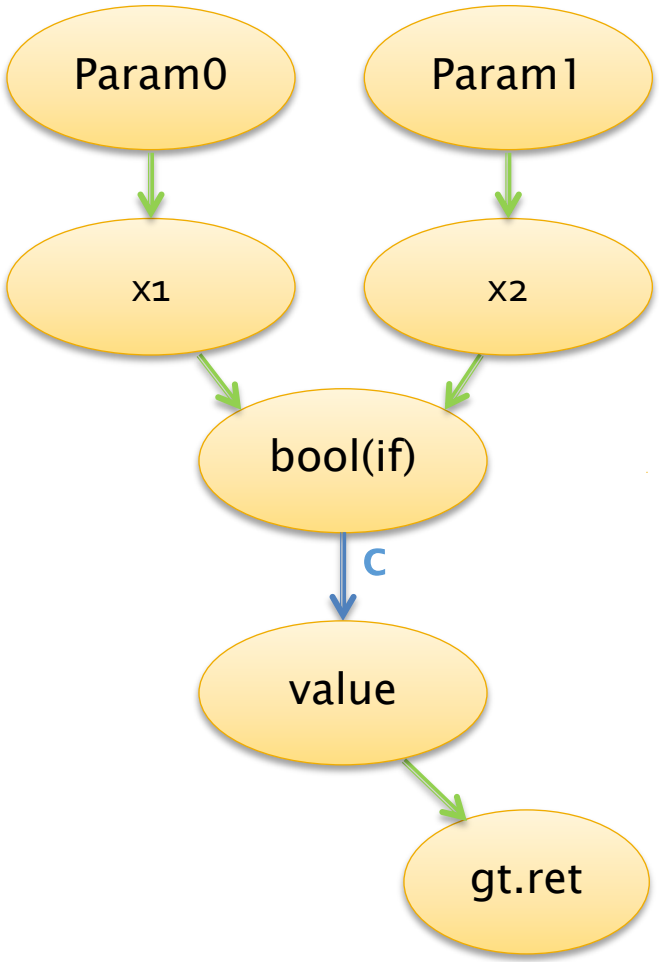
```
h1: x = y;
h2: x = 2;
h3: z = x;
```



Control Dependency Analysis

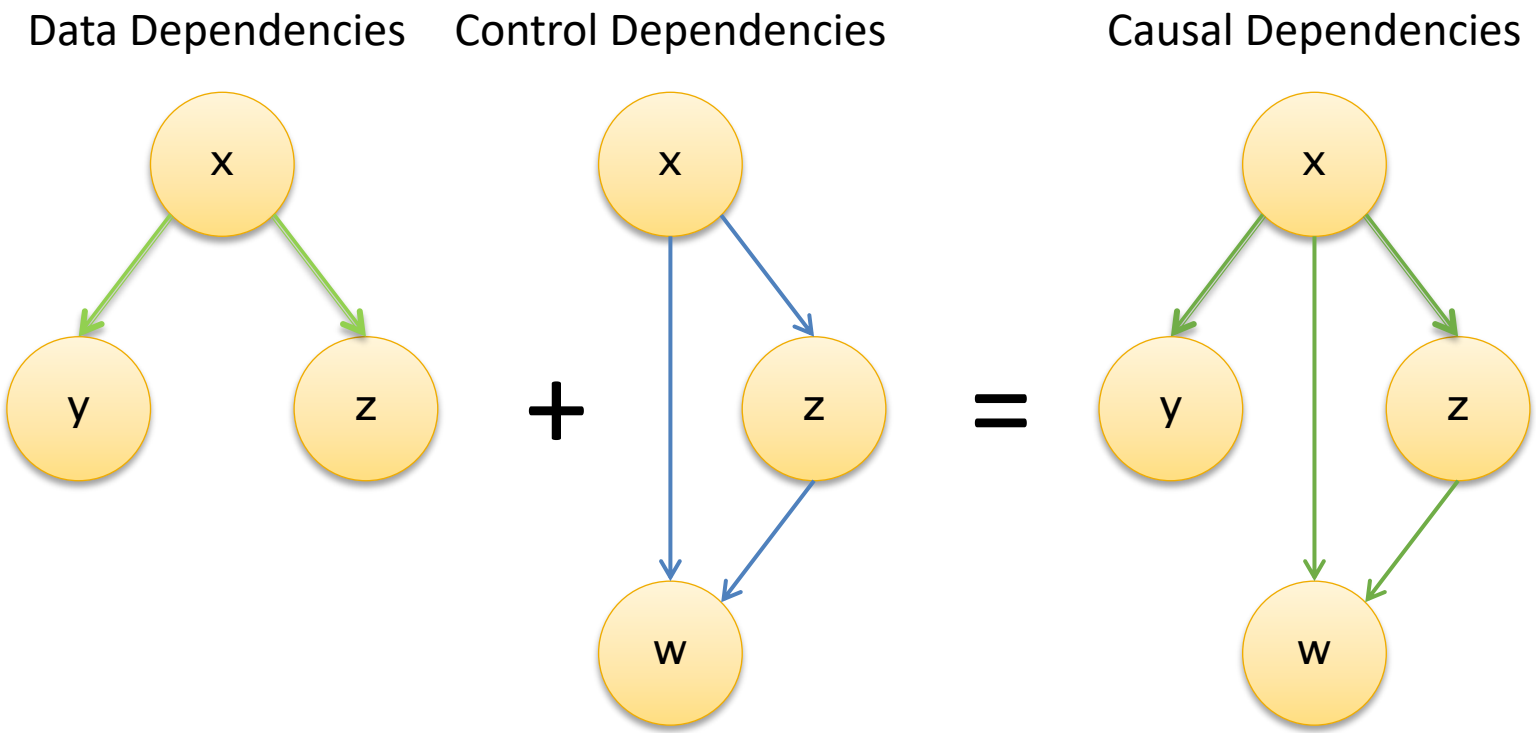
- Data Dependencies are not enough!

```
    boolean gt(int x1, int x2){
        boolean value;
h1:     if(x1>x2){
h2:             value = true;
        }else{
h3:             value = false;
        }
h4:     return value;
    }
```



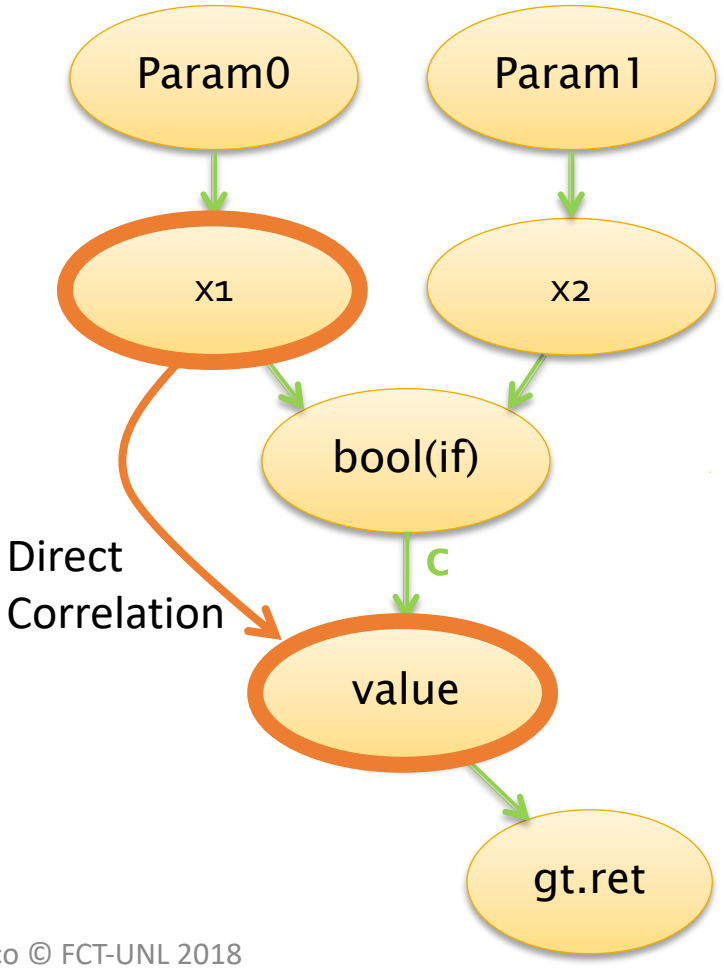
Causal Dependencies Graph

- Merge Data and Control Flow Dependencies in the Causal Dependencies Graph



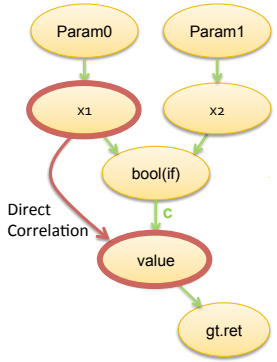
Direct Correlation

```
   boolean gt(int x1, int x2){
       boolean value;
h1:    if(x1>x2){
h2:            value = true;
       }else{
h3:            value = false;
       }
h4:    return value;
   }
```



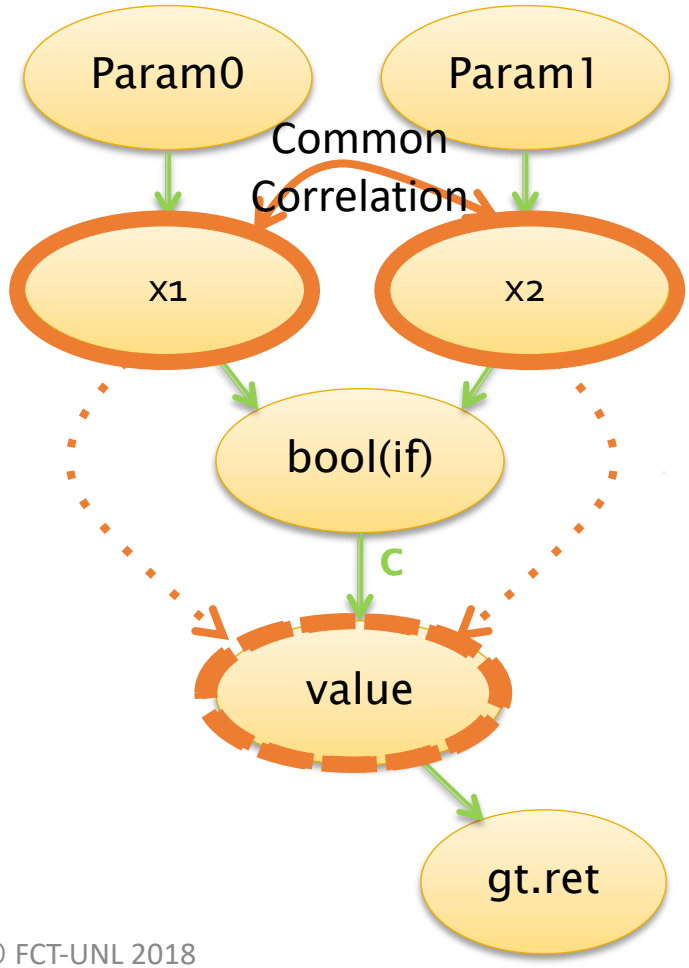
Variable's Correlation [Dias12]

- Direct Correlation (x, y):
  There is a direct correlation between a read variable 'x' and a written variable 'y' if there is a path from 'x' to 'y', in the dependency graph D.



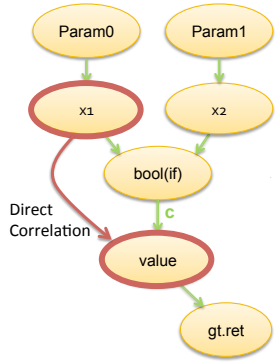
Common Correlation

```
   boolean gt(int x1, int x2){
       boolean value;
h1:    if(x1>x2){
h2:            value = true;
       }else{
h3:            value = false;
       }
h4:    return value;
   }
```

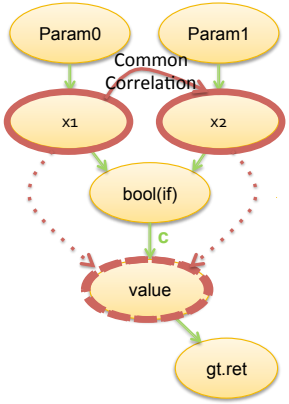


Variable's Correlation [Dias12]

- Direct Correlation (x, y):
  There is a direct correlation between a read variable 'x' and a written variable 'y' if there is a path from 'x' to 'y', in the dependency graph D.



- Common Correlation (x, y):
  There is a common correlation between read variables 'x' and 'y' if there is a written variable 'z', where 'z ≠ x' and 'z ≠ y', for which there is a path from 'x' to 'z' and another path from 'y' to 'z', in the dependency graph D.



High-Level Data Race

Thread 1

```
public synchronized
  int setPair(int v1, int v2){
    x = v1;
    y = v2;
  }
```

Thread 2

```
public boolean equals(){
  int loc_x = getX(); // synchr
  int loc_y = getY(); // synchr
  return loc_x == loc_y;
}
```

High-Level Data Race

Thread 1

```
public synchronized
  int setPair(int v1, int v2){
    x = v1;
    y = v2;
  }
```

Thread 2

```
public boolean equals(){
  int loc_x = getX(); // synchr
  int loc_y = getY(); // synchr
  return loc_x == loc_y;
}
```

Thread 1	Thread 2
V_w (setPair) = {x, y}	V_w (getX) = { }
V_r (setPair) = { }	V_r (getX) = {x}
	V_w (getY) = { }
	V_r (getY) = {y}

High-Level Data Race [Dias12]

Thread 1

```
public synchronized
  int setPair(int v1, int v2){
    x = v1;
    y = v2;
  }
```

Thread 2

```
public boolean equals(){
  int loc_x = getX(); // synchr
  int loc_y = getY(); // synchr
  return loc_x == loc_y;
}
```

Maximal View$_W$ of T1

Thread 1	Thread 2	
V_w (setPair) = {x, y}		
V_r (setPair) = { }	V_r (getX) = {x}	View$_R$ of T2
	V_w (getY) = { }	
	V_r (getY) = {y}	View$_R$ of T2

Data Race!

$\{x, y\} \cap \{x\} = \{x\}$

$\{x, y\} \cap \{y\} = \{y\}$

$(\{x\} \not\subseteq \{y\} \wedge \{y\} \not\subseteq \{x\})$ $\wedge$ Common Correlation ({x}, {y})

HLDR Quiz

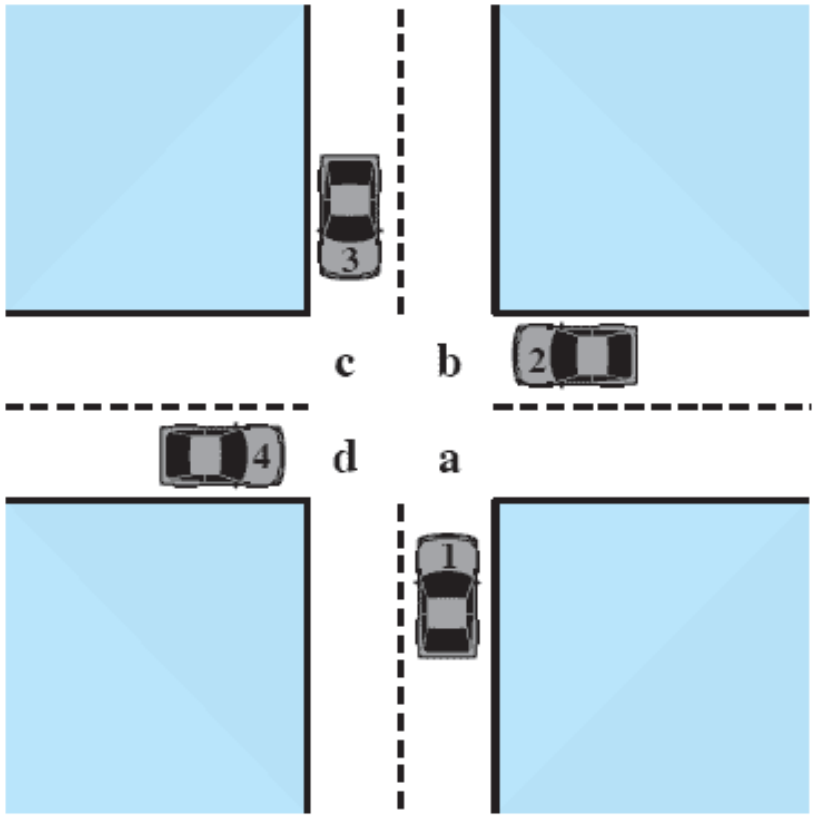
- T1 runs **V1 = {A, B, C}** and **V2 = {A, B, C, D}**
- T2 runs **V3 = {A, B, E}** and **V4 = {B, C, F}**
- **Is there a HLDR?**

- V2 is maximal in T1 (ignore V1)
- V2 ∩ V3 = {A, B} V2 ∩ V4 = {B, C}
- {A, B} ⊆ {B, C} or {B, C} ⊆ {A, B}? No!
- Common-Correlation(A, C)?
  - Yes! High Level Data Race!
  - No! **No** High Level Data Race

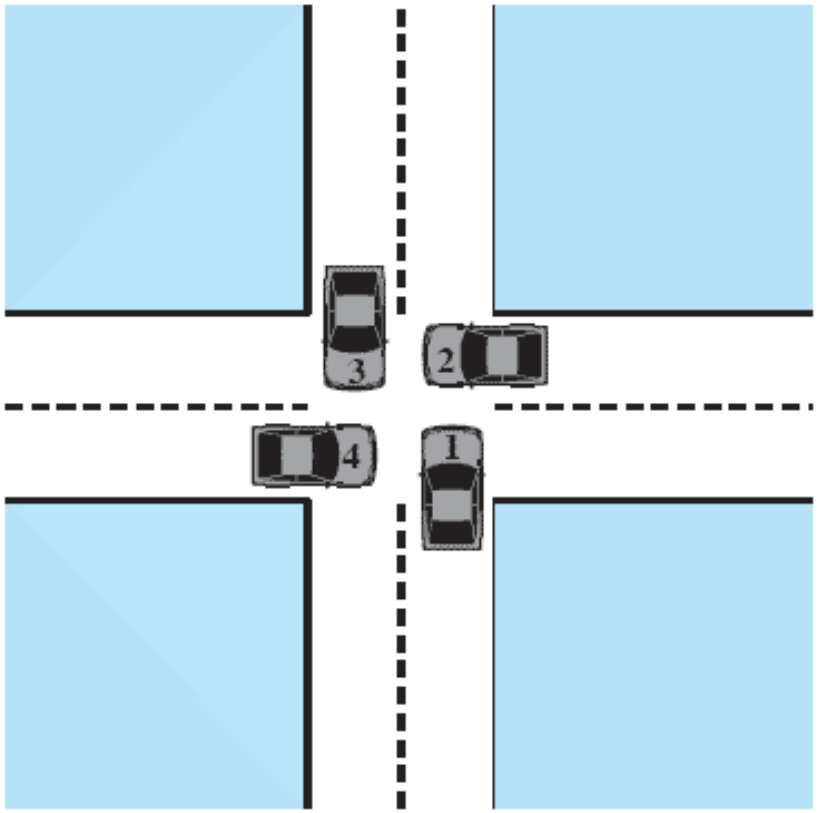
Deadlocks

Deadlock

Permanent blocking of a set of processes that either compete for system resources or communicate with each other.



(a) Deadlock possible



(b) Deadlock

System Model

- Finite number of resources
- Resources are organized into classes
  - Each class only contain identical resource instances
- Processes compete for accessing resources
- If a process request an instance of a resource class, *any* instance of that class must satisfy the process

Protocol to Use a Resource

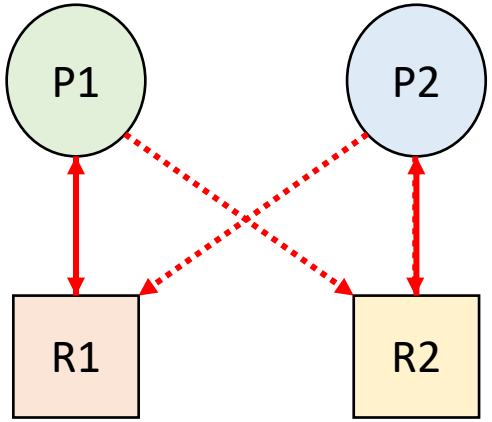
- **Request** — The process either gets an instance of the resource immediately; or waits until one is available (and gets it)
- **Use** — The process can operate on its resource instance
- **Release** — The process releases its resource instance

- Examples: malloc() & free() — open() & close()

Deadlock

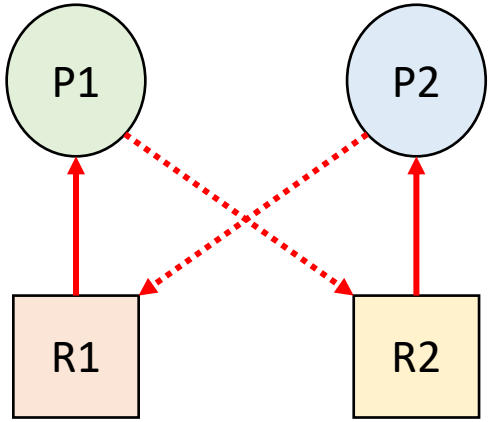
A set of two or more processes are deadlocked if:

1. They are blocked (i.e., in the waiting state)
2. Each is holding a resource
3. Each is waiting to acquire a resource held by another process in the set



Deadlock

- Deadlock depends on the dynamics of the execution
- Is difficult to identify and test for deadlocks, which may occur only under certain circumstances



Conditions Necessary for Deadlock

- **mutual exclusion**: only one process can use a resource at a time
- **hold and wait**: a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes
- **no preemption**: a resource can only be released voluntarily by the process holding it
- **circular wait**: a cycle of process requests exists (i.e., P_0 --> P_1 --> P_2 --> … --> P_{n-1} --> P_0).

Example

Thread 1

```
void *do_work_one(void *param) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param) {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

Example

Will deadlock happen?

Thread 1

```
void *do_work_one(void *param) {
  pthread_mutex_lock(&m1);
  pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param) {
  pthread_mutex_lock(&m2);
  pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

Example

Deadlock if executed in order:
- *1, 3, 2, 4; or*
- *1, 3, 4, 2; or*
- *3, 1, 2, 4; or*
- *3, 1, 4, 2*

Thread 1

```
void *do_work_one(void *param) {
1 pthread_mutex_lock(&m1);
2 pthread_mutex_lock(&m2);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m2);
  pthread_mutex_unlock(&m1);
  pthread_exit(0);
}
```

Thread 2

```
void *do_work_two(void *param) {
3 pthread_mutex_lock(&m2);
4 pthread_mutex_lock(&m1);
  /**
   * Do some work
   */
  pthread_mutex_unlock(&m1);
  pthread_mutex_unlock(&m2);
  pthread_exit(0);
}
```

These orderings are ok:
- *1, 2, 3, 4; and*
- *3, 4, 1, 2*

Resource Allocation Graph

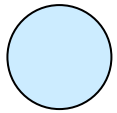
- A set of vertices **V** and a set of edges **E**

- **V** is partitioned into two types:
  - $P = \{P_1, P_2, \ldots, P_n\}$, the set of all the processes in the system
  - $R = \{R_1, R_2, \ldots, R_m\}$, the set of all resource types in the system

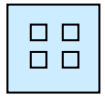
- **E** is partitioned into two types:
  - Request edge – directed edge $P_i \rightarrow R_j$
  - Assignment edge – directed edge $R_j \rightarrow P_i$

Resource Allocation Graph

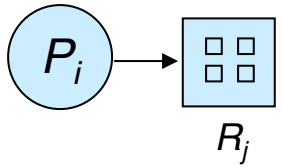
- Process



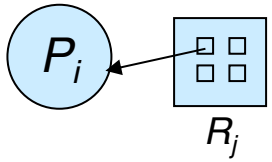
- Resource Type with 4 instances



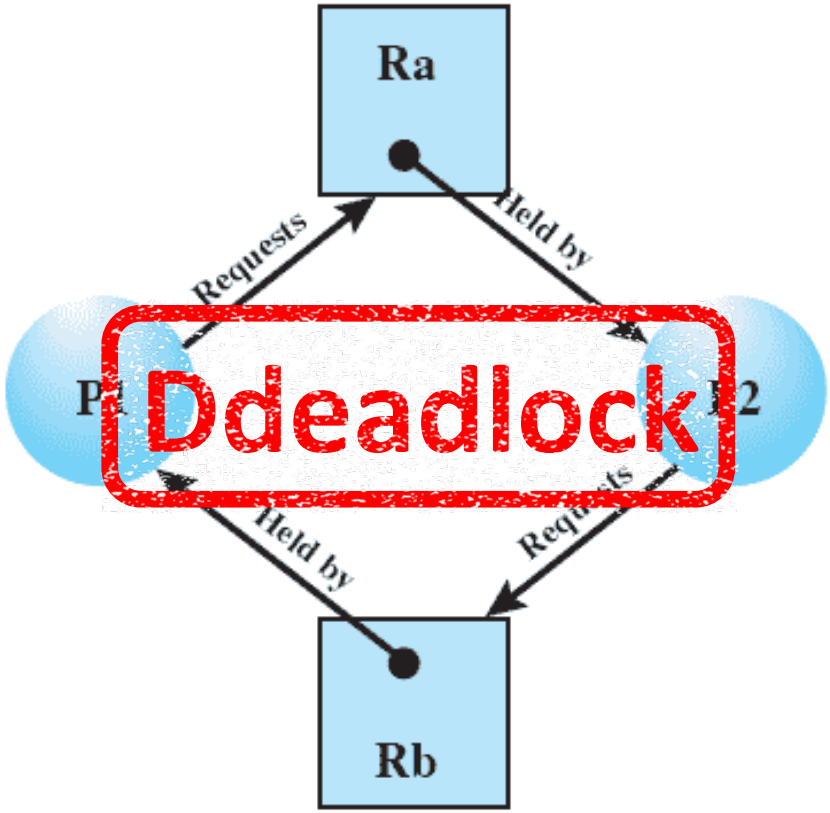
- P_i requests instance of R_j



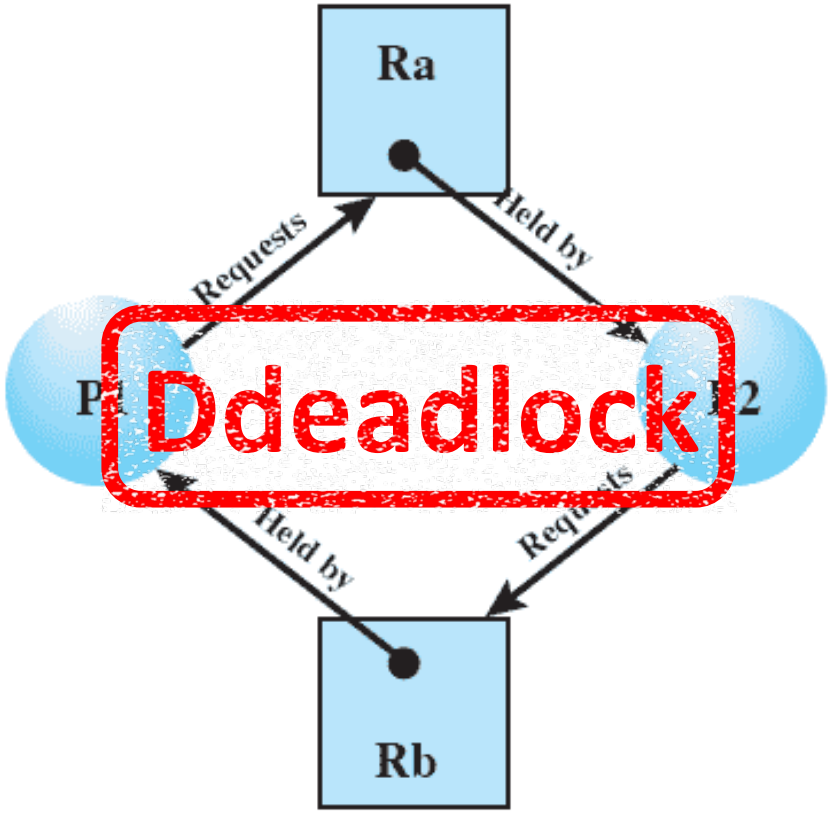
- P_i is holding an instance of R_j

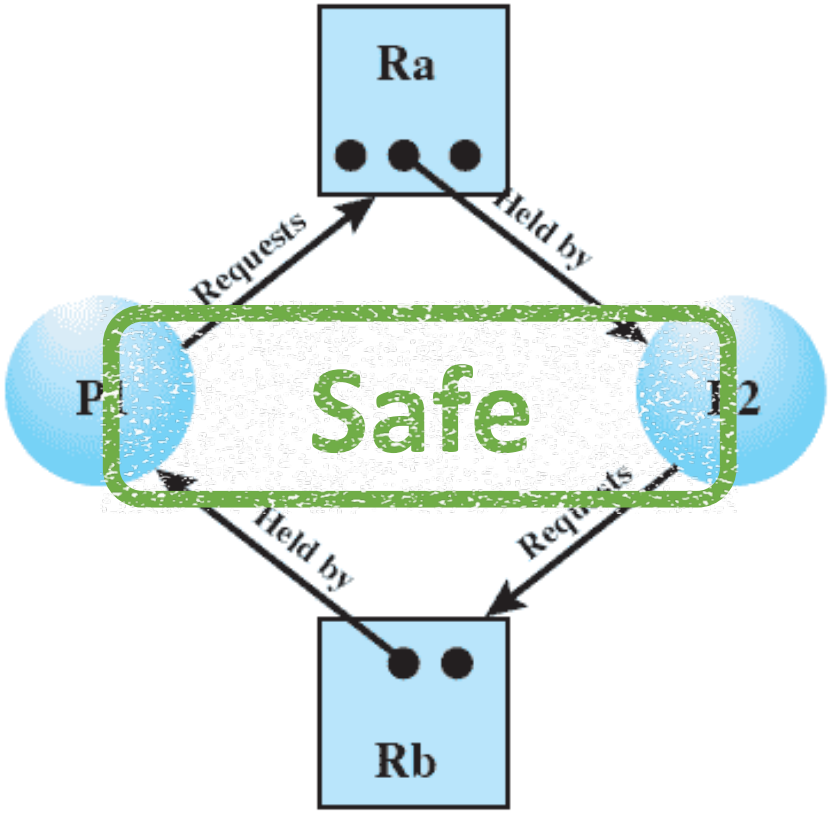


Example of Resource Allocation Graphs

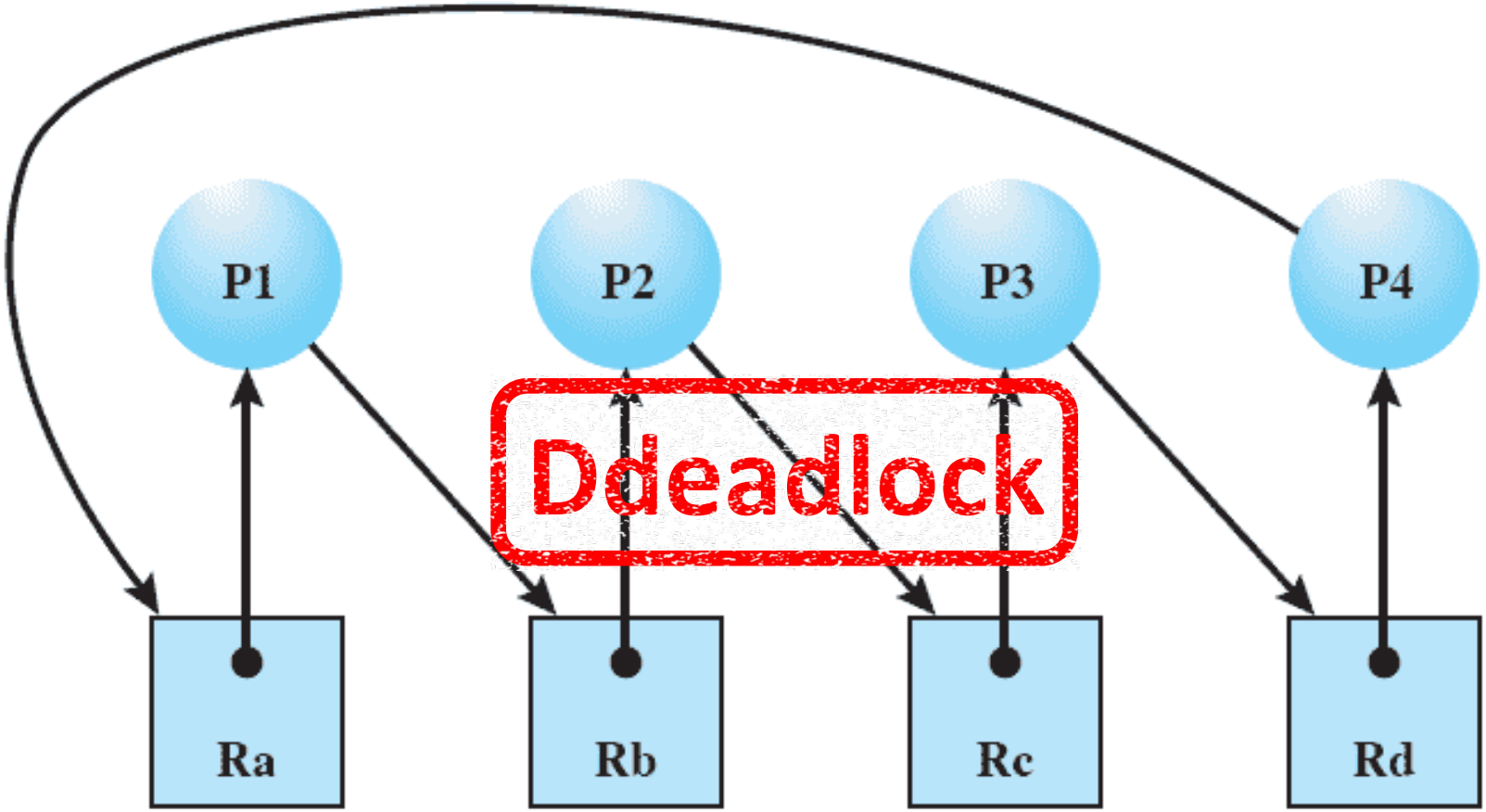


Example of Resource Allocation Graphs

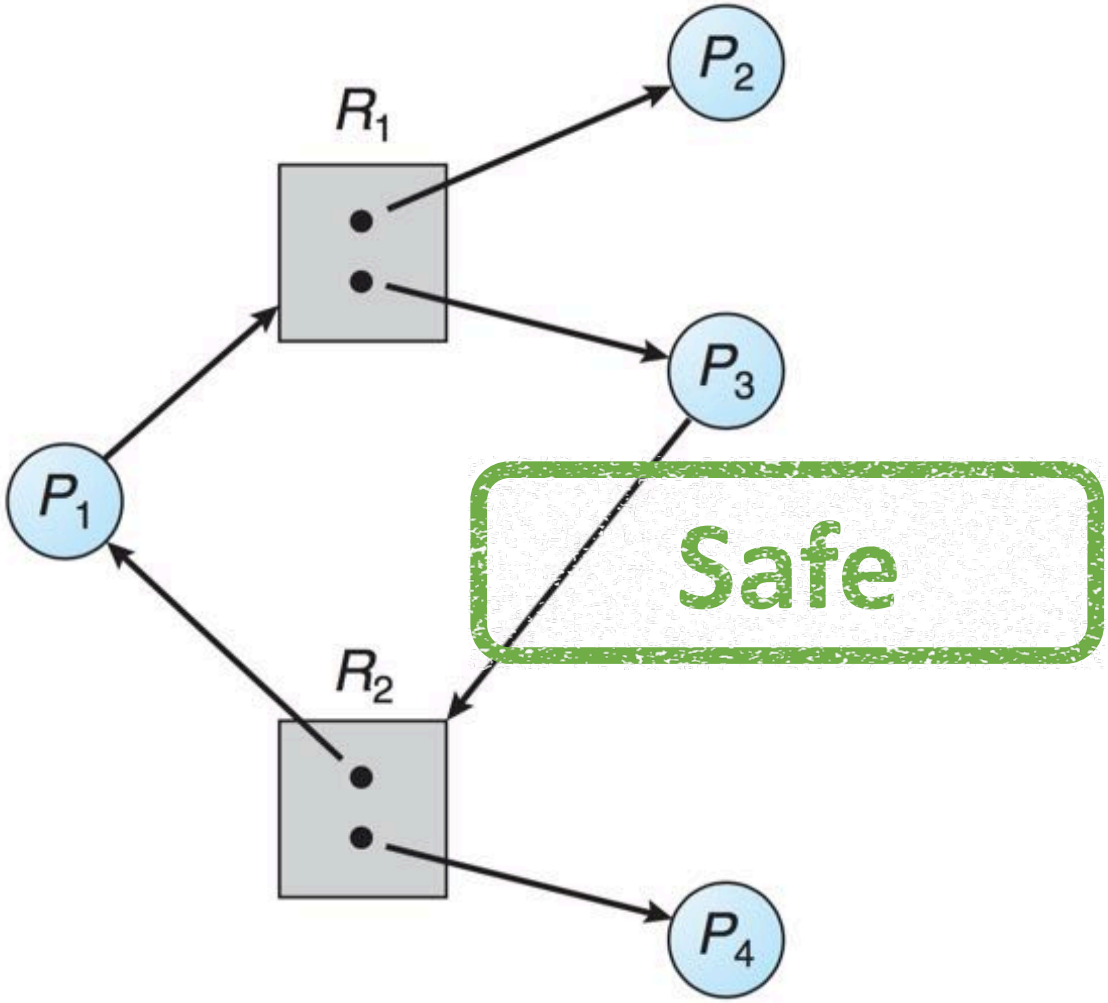




Example of Resource Allocation Graphs



Example of Resource Allocation Graphs



Basic Facts

- If graph contains no cycles ⇒ no deadlock

- If graph contains a cycle ⇒
  - if only one instance per resource type, then deadlock
  - if several instances per resource type, possibility of deadlock

How to Deal with Deadlocks?

- Deadlock prevention
- Deadlock avoidance

The system never enters a deadlock state

How to Deal with Deadlocks?

- Deadlock prevention
- Deadlock avoidance

The system never enters a deadlock state

- Deadlock detection and recovery
- Ignore the issue! ;)

The system may enter a deadlock state

Deadlocks

Deadlock prevention

Deadlock Prevention

- Provides a set of methods to ensure that at least one of the necessary conditions cannot hold

- These methods prevent deadlocks by constraining how requests for resources can be made

Conditions Necessary for Deadlock

- **mutual exclusion**: only one process can use a resource at a time
- **hold and wait**: a process holding at least one resource is waiting to acquire additional resources which are currently held by other processes
- **no preemption**: a resource can only be released voluntarily by the process holding it
- **circular wait**: a cycle of process requests exists (i.e., P_0 --> P_1 --> P_2 --> … --> P_{n-1} --> P_0).

Deadlock Prevention

- Restrict the way requests can be made…
- **Mutual Exclusion**
  - not required for sharable resources (e.g., read-only files); must hold for non-sharable resources
- **Hold and Wait**
  - must guarantee that whenever a process requests a resource, it does not hold any other resources
  - require process to request and be allocated all its resources before it begins execution, or allow process to request resources only when the process has none allocated to it
  - low resource utilization; starvation possible

Deadlock Prevention

- Restrict the way requests can be made…
- **No Preemption**
  - if a process that is holding some resources requests another resource that cannot be immediately allocated to it, then all resources currently being held are released
  - preempted resources are added to the list of resources for which the process is waiting
  - process will be restarted only when it can regain its old resources, as well as the new ones that it is requesting
- **Circular Wait**
  - impose a total ordering of all resource types, and require that each process requests resources in an increasing order of enumeration

Deadlocks

Deadlock avoidance

Deadlock Avoidance

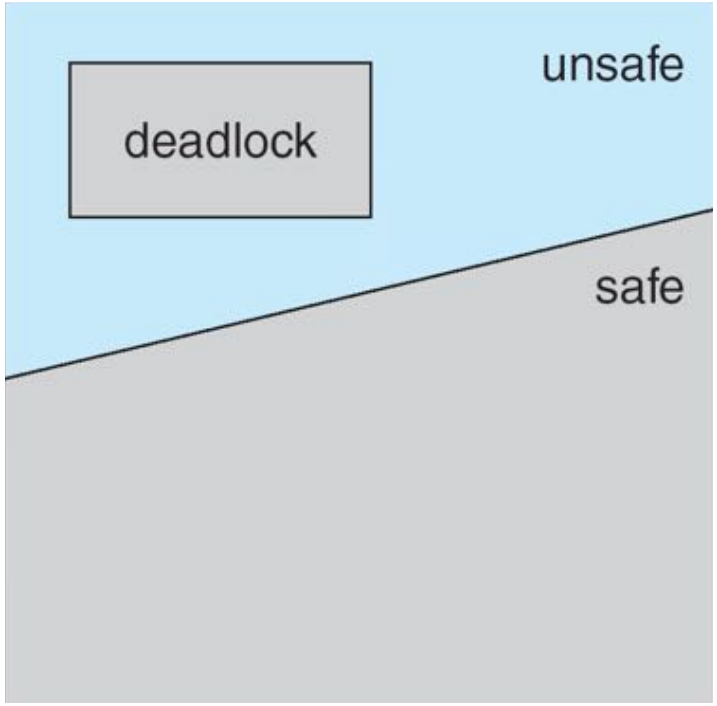
- Requires that the system has some additional *a priori* information available
  - Requires that each process declare the maximum number of resources of each type that it may need
  - The deadlock-avoidance algorithm dynamically examines the resource-allocation state to ensure that there can never be a circular-wait condition
  - Resource-allocation state is defined by the number of available and allocated resources, and the maximum demands of the processes

Safe State

- When a process requests an available resource, system must decide if immediate allocation leaves the system in a safe state
- **System is in safe state** if there exists a sequence $<P_1, P_2, \ldots, P_n>$ of ALL the processes in the systems such that for each P_i, the resources that P_i can still request can be satisfied by currently available resources + resources held by all the P_j, with $j < i$
- That is:
  - If P_i resource needs are not immediately available, then P_i can wait until all P_j have finished
  - When P_j is finished, P_i can obtain needed resources, execute, return allocated resources, and terminate
  - When P_i terminates, P_{i+1} can obtain its needed resources, and so on

Deadlock Avoidance

- If a system is in safe state ⇒ no deadlocks
- If a system is in unsafe state ⇒ possibility of deadlock
- Avoidance ⇒ ensure that a system will never enter an unsafe state



Avoidance Algorithms

- Single instance of a resource type

  Use a resource-allocation graph

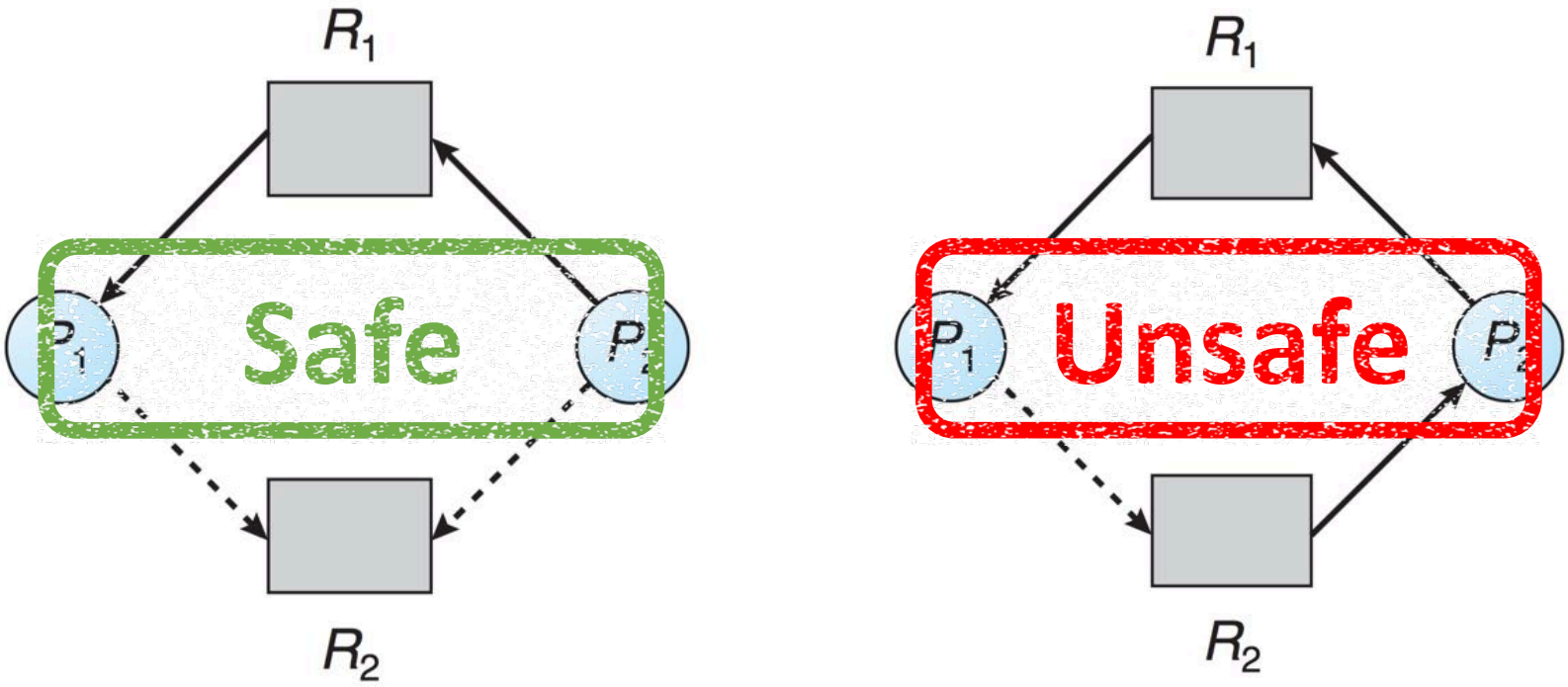
- Multiple instances of a resource type

  Use the banker's algorithm

Resource-Allocation Graph Scheme

- Claim edge $P_i \dashrightarrow R_j$ indicated that process P_j may request resource R_j; represented by a dashed line
- Claim edge converts to request edge when a process requests a resource
- Request edge converted to an assignment edge when the resource is allocated to the process
- When a resource is released by a process, assignment edge reconverts to a claim edge
- Resources must be claimed a priori in the system

Resource-Allocation Graph



Banker's Algorithm

- Resources may have multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Deadlocks

Deadlock detection

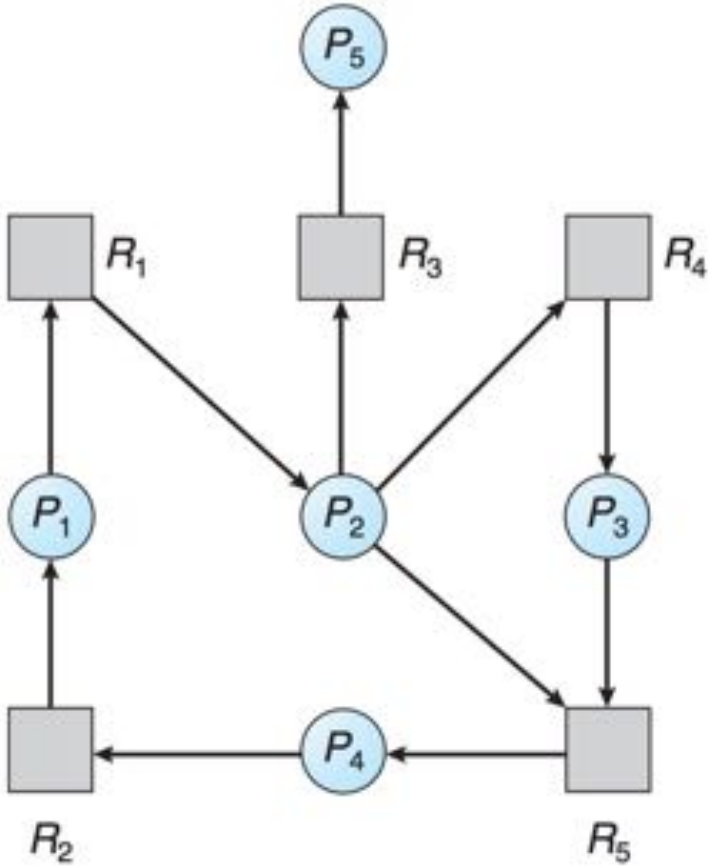
Deadlock Detection

- If neither avoidance or prevention is implemented, deadlocks can (and will) occur.
- Coping with this requires:
  - **Detection**: finding out if deadlock has occurred
    - Keep track of resource allocation (who has what)
    - Keep track of pending requests (who is waiting for what)
  - **Recovery**: resolve the deadlock

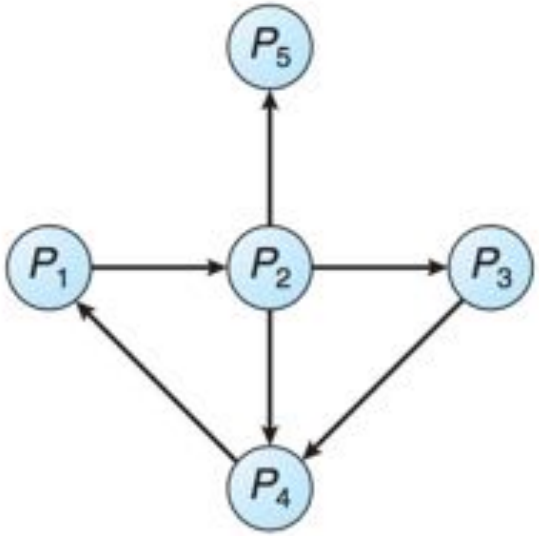
Single Instance of Each Resource Type

- Maintain wait-for graph
  - Nodes are processes
  - $P_i \rightarrow P_j$ if P_i is waiting for P_j
- Periodically invoke an algorithm that searches for a cycle in the graph
  - If there is a cycle, there exists a deadlock
- An algorithm to detect a cycle in a graph requires an order of n^2 operations, where n is the number of vertices in the graph

Resource-Allocation Graph and Wait-for Graph



Resource-Allocation Graph



Corresponding wait-for graph

Several Instances of a Resource Type

- Yes! It is possible!
- Algorithm inspired in the Banker's algorithm

Strategies Once Deadlock Detected

- Abort all deadlocked processes
- Resource preemption
- Roll back each deadlocked process to some previously defined checkpoint, and restart all process
  - Original deadlock may occur

Recovery from Deadlock: Process Termination

- Abort all deadlocked processes
- Abort one process at a time until the deadlock cycle is eliminated
- In which order should we choose to abort?
  - Priority of the process?
  - How long process has computed
  - How much longer to completion?
  - Resources the process has used?
  - Resources process needs to complete?
  - How many processes will need to be terminated?
  - Is process interactive or batch?

Recovery from Deadlock: Resource Preemption

- **Selecting a victim** – minimize cost
- **Rollback** – return to some safe state, restart process for that state
- **Starvation** – same process may always be picked as victim, include number of rollback in cost factor

Roll Back

- Roll back all the processes
  - Possibly to a situation where no locks are being held
- Pray for the deadlock to not happen again

Banker's Algorithm

- Resources may have multiple instances
- Each process must a priori claim maximum use
- When a process requests a resource it may have to wait
- When a process gets all its resources it must return them in a finite amount of time

Banker's Algorithm

https://www.youtube.com/watch?v=w0LwGqffUkg

Initial

A	B	C
10	5	7

Available

A	B	C
3	3	2

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
	7	2	5

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Need (= max – alloc)

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

P_0 — finish[0] = F

P_1 — finish[1] = T

P_2 — finish[2] = F

P_3 — finish[3] = T

P_4 — finish[4] = T

Available

5	3	2
7	4	3
7	4	5

P_0 — finish[0] = T

P_2 — finish[2] = T

Available

7	5	5
10	5	7

$<P_1, P_3, P_4, P_0, P_2>$

Banker's Algorithm

https://www.youtube.com/watch?v=w0LwGqffUkg

Initial

A	B	C
10	5	7

Available

A	B	C
3	3	2

Allocation

	A	B	C
P_0	0	1	0
P_1	2	0	0
P_2	3	0	2
P_3	2	1	1
P_4	0	0	2
	7	2	5

Max

	A	B	C
P_0	7	5	3
P_1	3	2	2
P_2	9	0	2
P_3	2	2	2
P_4	4	3	3

Need (= max – alloc)

	A	B	C
P_0	7	4	3
P_1	1	2	2
P_2	6	0	0
P_3	0	1	1
P_4	4	3	1

P_0 — finish[0] = F

P_1 — finish[1] = T

P_2 — finish[2] = F

P_3 — finish[3] = T

P_4 — finish[4] = T

Available

5	3	2
7	4	3
7	4	5

P_0 — finish[0] = T

P_2 — finish[2] = T

Available

7	5	5
10	5	7

$<P_1, P_3, P_4, P_0, P_2>$

Acknowledgments

- Some parts of this presentation was based in publicly available slides and PDFs
  - www.cs.cornell.edu/courses/cs4410/2011su/slides/lecture10.pdf
  - www.microsoft.com/en-us/research/people/madanm/
  - williamstallings.com/OperatingSystems/
  - codex.cs.yale.edu/avi/os-book/OS9/slide-dir/


That's all Folks!

PosDoc position @ Lisbon, PT

- **HiPsTr — High Performance Transactions**
  - Recently approved research project (April 2018)
  - Work contract (2½ years)

- The project addresses the dilemma of improving the efficiency of resilient systems while keeping the software systems correct and easy to develop and use.
  - **Keywords:** Fault Tolerance, Replication, Performance, Transactional Systems

Joao Lourenço <joao.lourenco@fct.unl.pt>

departamento de informática
FACULDADE DE CIÊNCIAS E TECNOLOGIA
UNIVERSIDADE NOVA DE LISBOA

NOVALINCS
LABORATORY FOR COMPUTER SCIENCE AND INFORMATICS