Behaviour-Oriented Concurrency

Hanji Shen

Jan 5, 2025

https://github.com/Foreverhighness/boc-talk

Links

Behaviour-Oriented Concurrency Paper

Basic C# implementation

Core C++ implementation

Presentation video

Supplementary

KAIST CS431: Concurrent Programming

Concurrency

Parallelism

- Thread
- Task
- Coroutines
- Async
- …

Coordination

- Promises
- Locks
- Condition variables
- Transactions
- …

Concurrency

Parallelism

- Thread
- Task
- Coroutines
- Async
- …

Coordination

- Promises
- Locks
- Condition variables
- Transactions
- …

$$\text{spawn}(A \xrightarrow{\text{transfer 100}} B\ ) \qquad \text{spawn}(B \xrightarrow{\text{transfer 100}} C)$$

$$\text{spawn}(C \xrightarrow{\text{transfer 100}} D\ ) \qquad \text{spawn}(D \xrightarrow{\text{transfer 100}} A)$$

Goal

- Isolation
- Parallelism
- Deadlock Freedom
- Ordering

$$\text{spawn}(A \xrightarrow{\text{transfer } 100} B\ ) \qquad \text{spawn}(B \xrightarrow{\text{transfer } 100} C)$$

$$\text{spawn}(C \xrightarrow{\text{transfer } 100} D\ ) \qquad \text{spawn}(D \xrightarrow{\text{transfer } 100} A)$$

Goal

- Isolation $\rightarrow$ exclusive access (Mutex)
- Parallelism
- Deadlock Freedom
- Ordering

$$\text{spawn}(\&\text{mut } A \xrightarrow{\text{transfer } 100} \&\text{mut } B\ ) \qquad \text{spawn}(\&\text{mut } B \xrightarrow{\text{transfer } 100} \&\text{mut } C)$$

$$\text{spawn}(\&\text{mut } C \xrightarrow{\text{transfer } 100} \&\text{mut } D\ ) \qquad \text{spawn}(\&\text{mut } D \xrightarrow{\text{transfer } 100} \&\text{mut } A)$$

Goal

- Isolation → exclusive access (Mutex)
- Parallelism
- Deadlock Freedom
- Ordering

$\text{spawn(\&mut A} \xrightarrow{\text{transfer 100}} \text{\&mut B)}$ $\text{spawn(\&mut B} \xrightarrow{\text{transfer 100}} \text{\&mut C)}$

Goal

- Isolation → exclusive access (Mutex)
- Parallelism
- Deadlock Freedom
- Ordering

$$\text{spawn(\&mut A} \xrightarrow{\text{transfer 100}} \text{\&mut B)}$$

$$\text{spawn(\&mut C} \xrightarrow{\text{transfer 100}} \text{\&mut D)}$$

Goal

- Isolation → exclusive access (Mutex)
- Parallelism
- Deadlock Freedom → Deadlock avoidance (Sort)
- Ordering

$\text{spawn}(\&\text{mut } A \xrightarrow{\text{transfer } 100} \&\text{mut } B\ )$ $\text{spawn}(\&\text{mut } B \xrightarrow{\text{transfer } 100} \&\text{mut } C)$

$\text{spawn}(\&\text{mut } C \xrightarrow{\text{transfer } 100} \&\text{mut } D\ )$ $\text{spawn}(\&\text{mut } D \xrightarrow{\text{transfer } 100} \&\text{mut } A)$

Goal

- Isolation → exclusive access (Mutex)
- Parallelism
- Deadlock Freedom → Deadlock avoidance (Sort)
- Ordering → DAG (Dependency Graph)

$$\text{spawn}(\&\text{mut } A \xrightarrow{\text{transfer } 100} \&\text{mut } B\ ) \qquad \text{spawn}(\&\text{mut } B \xrightarrow{\text{transfer } 100} \&\text{mut } C)$$

$$\text{spawn}(\&\text{mut } C \xrightarrow{\text{transfer } 100} \&\text{mut } D\ ) \qquad \text{spawn}(\&\text{mut } D \xrightarrow{\text{transfer } 100} \&\text{mut } A)$$

Goal

- Isolation → exclusive access (Mutex)
- Parallelism
- Deadlock Freedom → Deadlock avoidance (Sort)
- Ordering → DAG (Dependency Graph)

$$\text{spawn}(\&\text{mut A} \xrightarrow{\text{transfer 100}} \&\text{mut B}) \qquad \text{spawn}(\&\text{mut B} \xrightarrow{\text{transfer 100}} \&\text{mut C})$$

$$\text{spawn}(\&\text{mut C} \xrightarrow{\text{transfer 100}} \&\text{mut D}) \qquad \text{spawn}(\&\text{mut D} \xrightarrow{\text{transfer 100}} \&\text{mut A})$$

BoC in nutshell

- Cown: protects a piece of separated data → Mutex
- Behaviour: unit of concurrent execution → Thread
- When: spawns a behaviour with a set of required cowns → Spawn

$$\text{when}(\text{Cown}<\text{A}>,\ \text{Cown}<\text{B}>;\ \&\text{mut A} \xrightarrow{\text{transfer 100}} \&\text{mut B})$$

$$\text{when}(\text{Cown}<\text{B}>,\ \text{Cown}<\text{C}>;\ \&\text{mut B} \xrightarrow{\text{transfer 100}} \&\text{mut C})$$

$$\text{when}(\text{Cown}<\text{C}>,\ \text{Cown}<\text{D}>;\ \&\text{mut C} \xrightarrow{\text{transfer 100}} \&\text{mut D})$$

$$\text{when}(\text{Cown}<\text{D}>,\ \text{Cown}<\text{A}>;\ \&\text{mut D} \xrightarrow{\text{transfer 100}} \&\text{mut A})$$

BoC in nutshell

- Cown: protects a piece of separated data → Mutex
- Behaviour: unit of concurrent execution → Thread
- When: spawns a behaviour with a set of required cowns → Spawn

$$\text{when}(\text{Cown}<A>,\ \text{Cown}<B>;\ \&\text{mut}\ A \xrightarrow{\text{transfer }100} \&\text{mut}\ B)$$

$$\text{when}(\text{Cown}<B>,\ \text{Cown}<C>;\ \&\text{mut}\ B \xrightarrow{\text{transfer }100} \&\text{mut}\ C)$$

$$\text{when}(\text{Cown}<C>,\ \text{Cown}<D>;\ \&\text{mut}\ C \xrightarrow{\text{transfer }100} \&\text{mut}\ D)$$

$$\text{when}(\text{Cown}<D>,\ \text{Cown}<A>;\ \&\text{mut}\ D \xrightarrow{\text{transfer }100} \&\text{mut}\ A)$$

BoC in nutshell

- Cown: protects a piece of separated data → Mutex
- Behaviour: unit of concurrent execution → Thread
- When: spawns a behaviour with a set of required cowns → Spawn

$$\text{when}(\text{Cown}<\text{A}>,\ \text{Cown}<\text{B}>;\ \&\text{mut A} \xrightarrow{\text{transfer } 100} \&\text{mut B})$$

$$\text{when}(\text{Cown}<\text{B}>,\ \text{Cown}<\text{C}>;\ \&\text{mut B} \xrightarrow{\text{transfer } 100} \&\text{mut C})$$

$$\text{when}(\text{Cown}<\text{C}>,\ \text{Cown}<\text{D}>;\ \&\text{mut C} \xrightarrow{\text{transfer } 100} \&\text{mut D})$$

$$\text{when}(\text{Cown}<\text{D}>,\ \text{Cown}<\text{A}>;\ \&\text{mut D} \xrightarrow{\text{transfer } 100} \&\text{mut A})$$

BoC in nutshell

- Cown: protects a piece of separated data → Mutex
- Behaviour: unit of concurrent execution → Thread
- When: spawns a behaviour with a set of required cowns → Spawn

$$\text{when}(\text{Cown}<A>,\ \text{Cown}<B>;\ \&\text{mut } A \xrightarrow{\text{transfer } 100} \&\text{mut } B)$$

$$\text{when}(\text{Cown}<B>,\ \text{Cown}<C>;\ \&\text{mut } B \xrightarrow{\text{transfer } 100} \&\text{mut } C)$$

$$\text{when}(\text{Cown}<C>,\ \text{Cown}<D>;\ \&\text{mut } C \xrightarrow{\text{transfer } 100} \&\text{mut } D)$$

$$\text{when}(\text{Cown}<D>,\ \text{Cown}<A>;\ \&\text{mut } D \xrightarrow{\text{transfer } 100} \&\text{mut } A)$$

BoC in nutshell

- Cown: protects a piece of separated data → Mutex
- Behaviour: unit of concurrent execution → Thread
- When: spawns a behaviour with a set of required cowns → Spawn

$$\text{when}(\text{Cown}<\text{A}>,\ \text{Cown}<\text{B}>;\ \&\text{mut A} \xrightarrow{\text{transfer 100}} \&\text{mut B})$$

$$\text{when}(\text{Cown}<\text{B}>,\ \text{Cown}<\text{C}>;\ \&\text{mut B} \xrightarrow{\text{transfer 100}} \&\text{mut C})$$

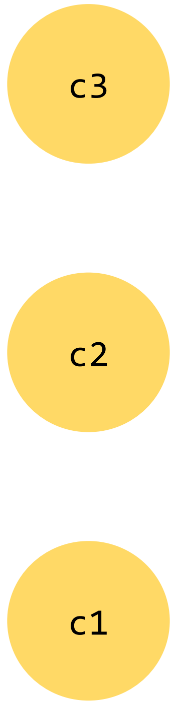
$$\text{when}(\text{Cown}<\text{C}>,\ \text{Cown}<\text{D}>;\ \&\text{mut C} \xrightarrow{\text{transfer 100}} \&\text{mut D})$$

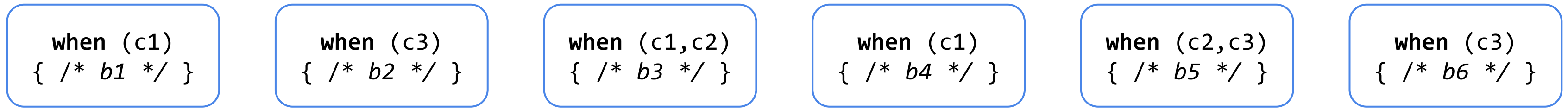
$$\text{when}(\text{Cown}<\text{D}>,\ \text{Cown}<\text{A}>;\ \&\text{mut D} \xrightarrow{\text{transfer 100}} \&\text{mut A})$$

Abstraction

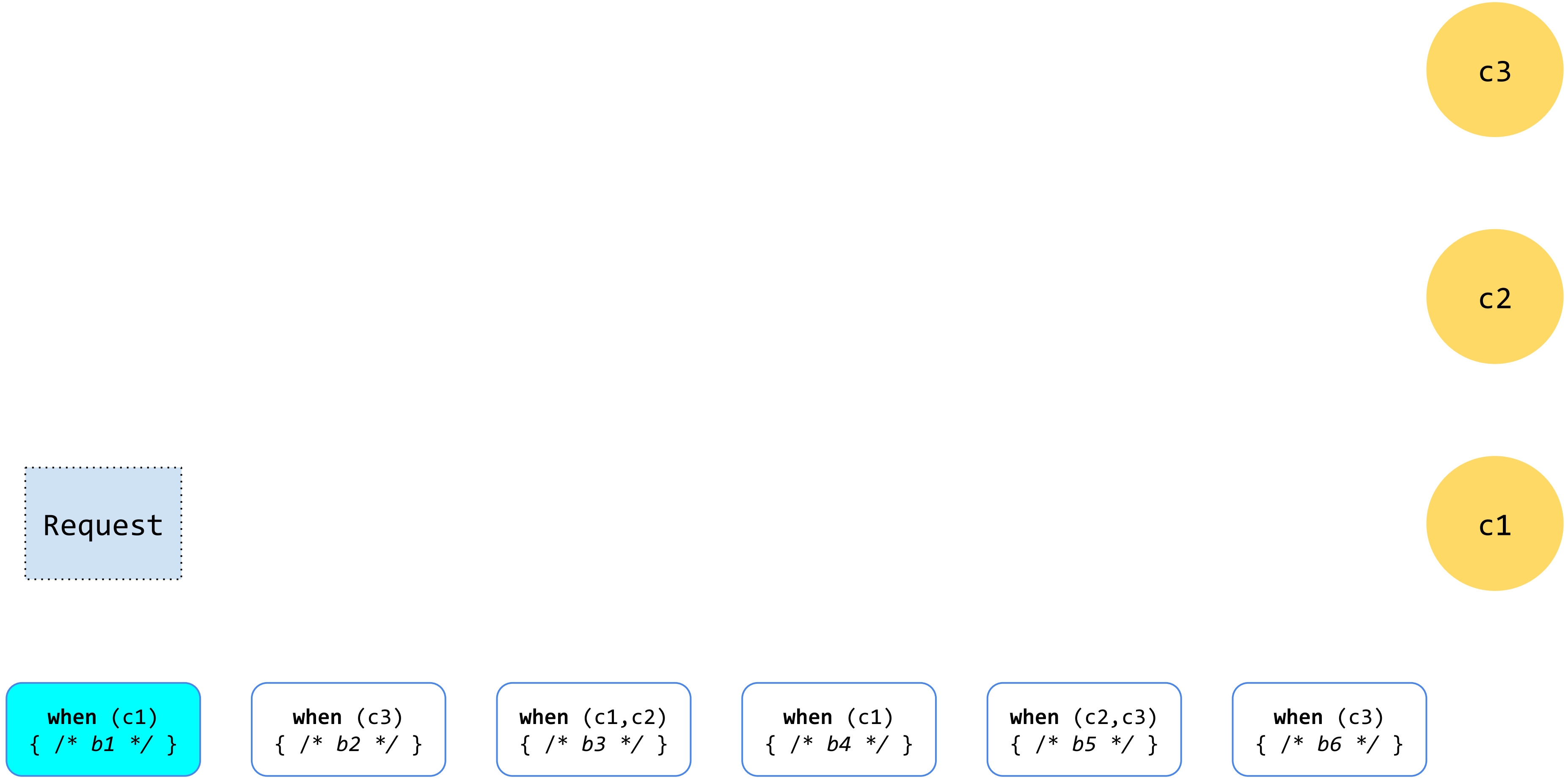
Example:

```
when (c1)     { /* b1 */ }
when (c3)     { /* b2 */ }
when (c1, c2) { /* b3 */ }
when (c1)     { /* b4 */ }
when (c2, c3) { /* b5 */ }
when (c3)     { /* b6 */ }
when (c1)     { /* b1 */ }
when (c3)     { /* b2 */ }
when (c1, c2) { /* b3 */ }
when (c1)     { /* b4 */ }
when (c2, c3) { /* b5 */ }
when (c3)     { /* b6 */ }
```





```
when (c1)        when (c3)        when (c1,c2)     when (c1)        when (c2,c3)     when (c3)
{ /* b1 */ }     { /* b2 */ }     { /* b3 */ }     { /* b4 */ }     { /* b5 */ }     { /* b6 */ }
```



```
when (c1)
{ /* b1 */ }
```

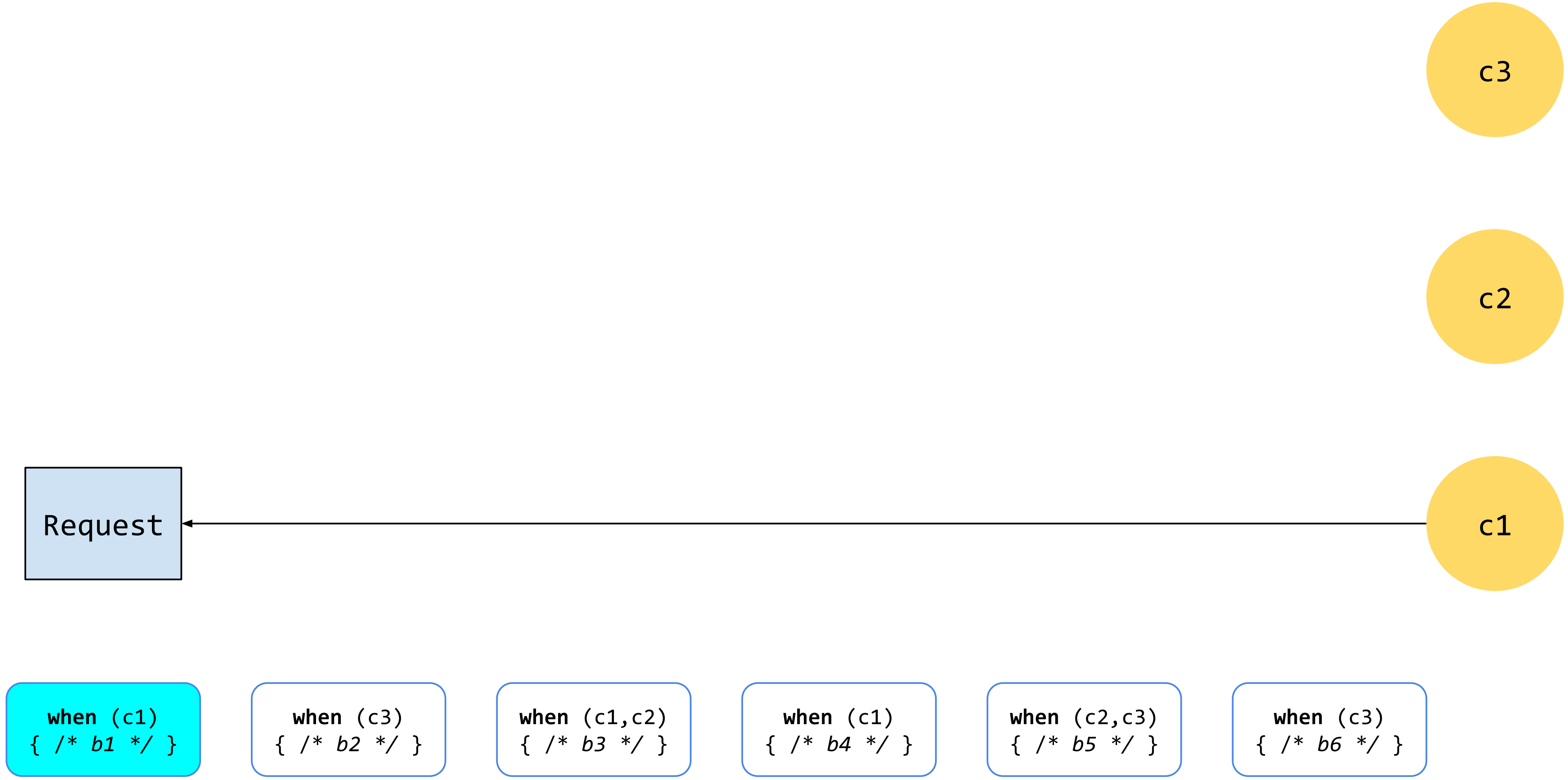
```
when (c3)
{ /* b2 */ }
```

```
when (c1,c2)
{ /* b3 */ }
```

```
when (c1)
{ /* b4 */ }
```

```
when (c2,c3)
{ /* b5 */ }
```

```
when (c3)
{ /* b6 */ }
```



```
when (c1)
{ /* b1 */ }
```

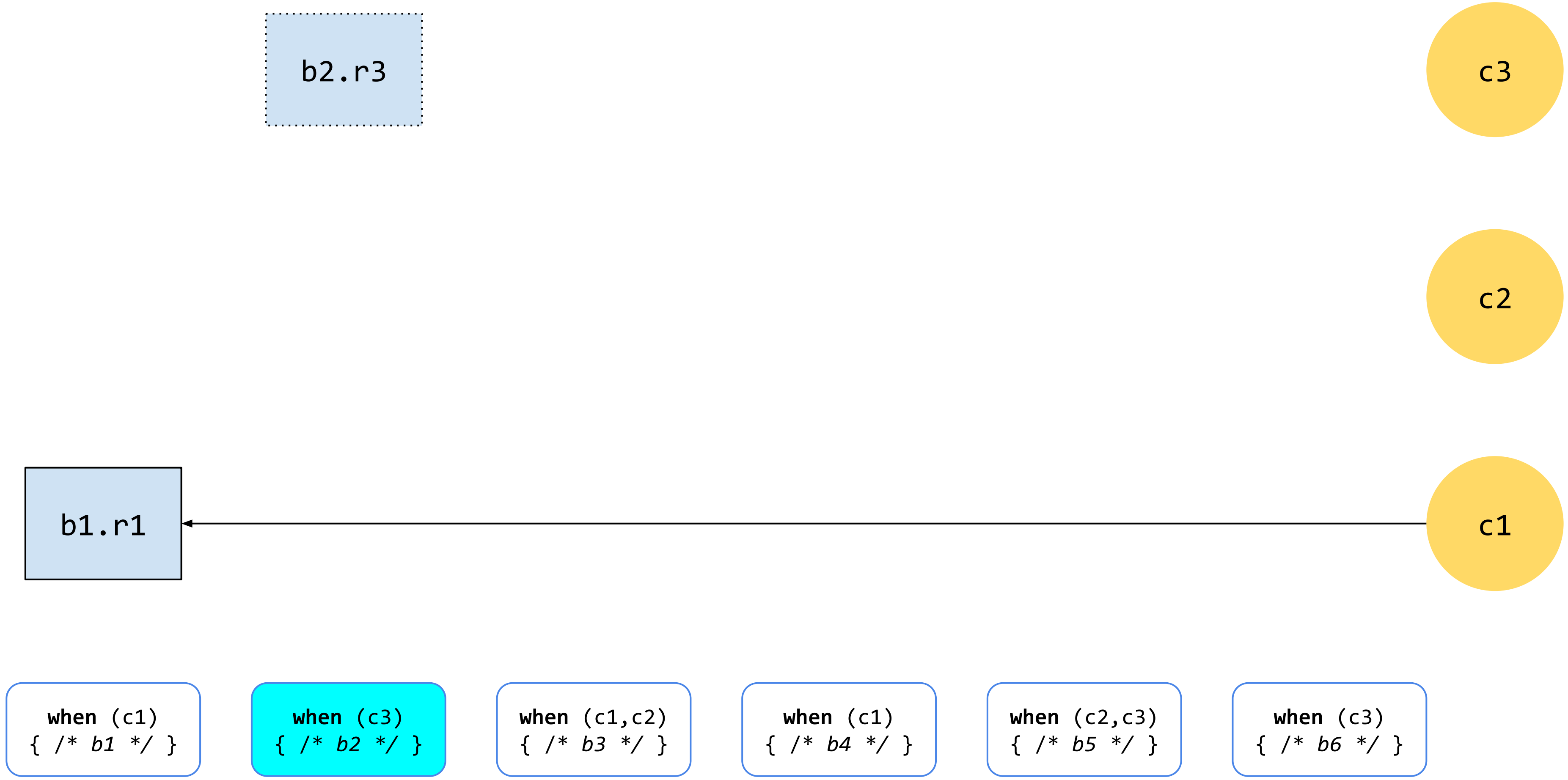
```
when (c3)
{ /* b2 */ }
```

```
when (c1,c2)
{ /* b3 */ }
```

```
when (c1)
{ /* b4 */ }
```

```
when (c2,c3)
{ /* b5 */ }
```

```
when (c3)
{ /* b6 */ }
```



```
when (c1)
{ /* b1 */ }
```

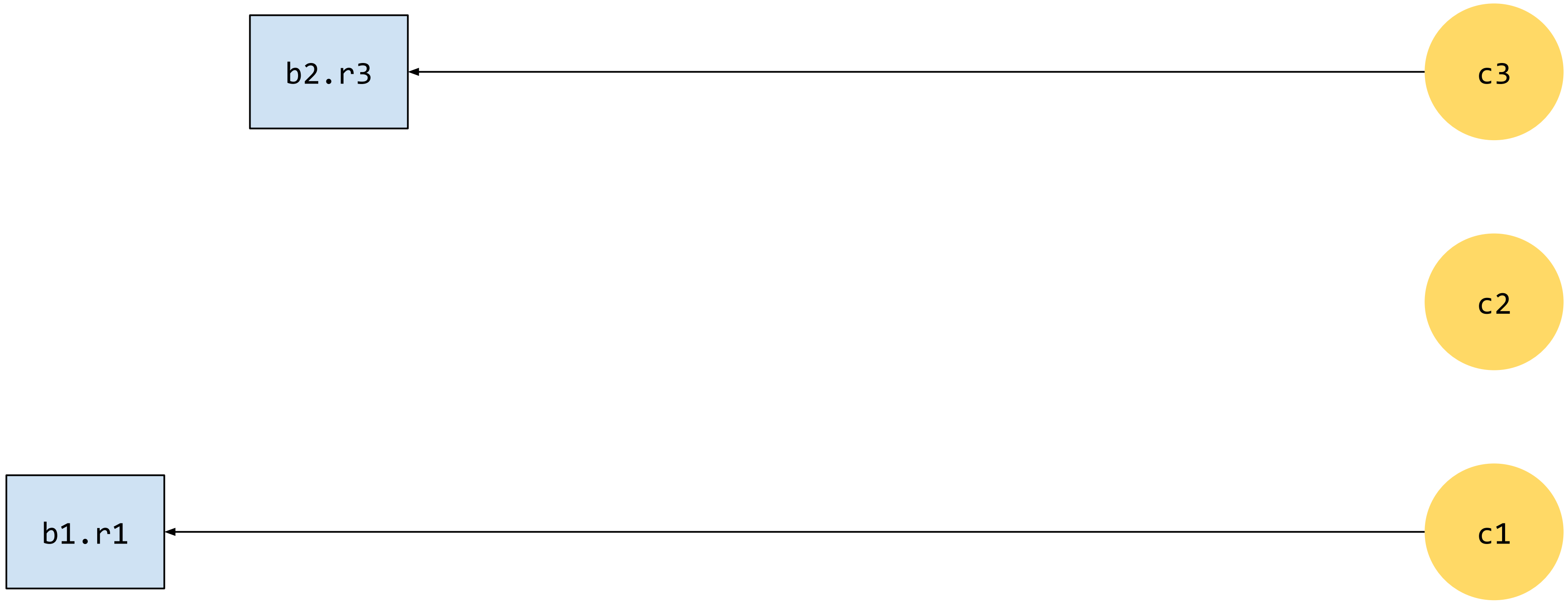
```
when (c3)
{ /* b2 */ }
```

```
when (c1,c2)
{ /* b3 */ }
```

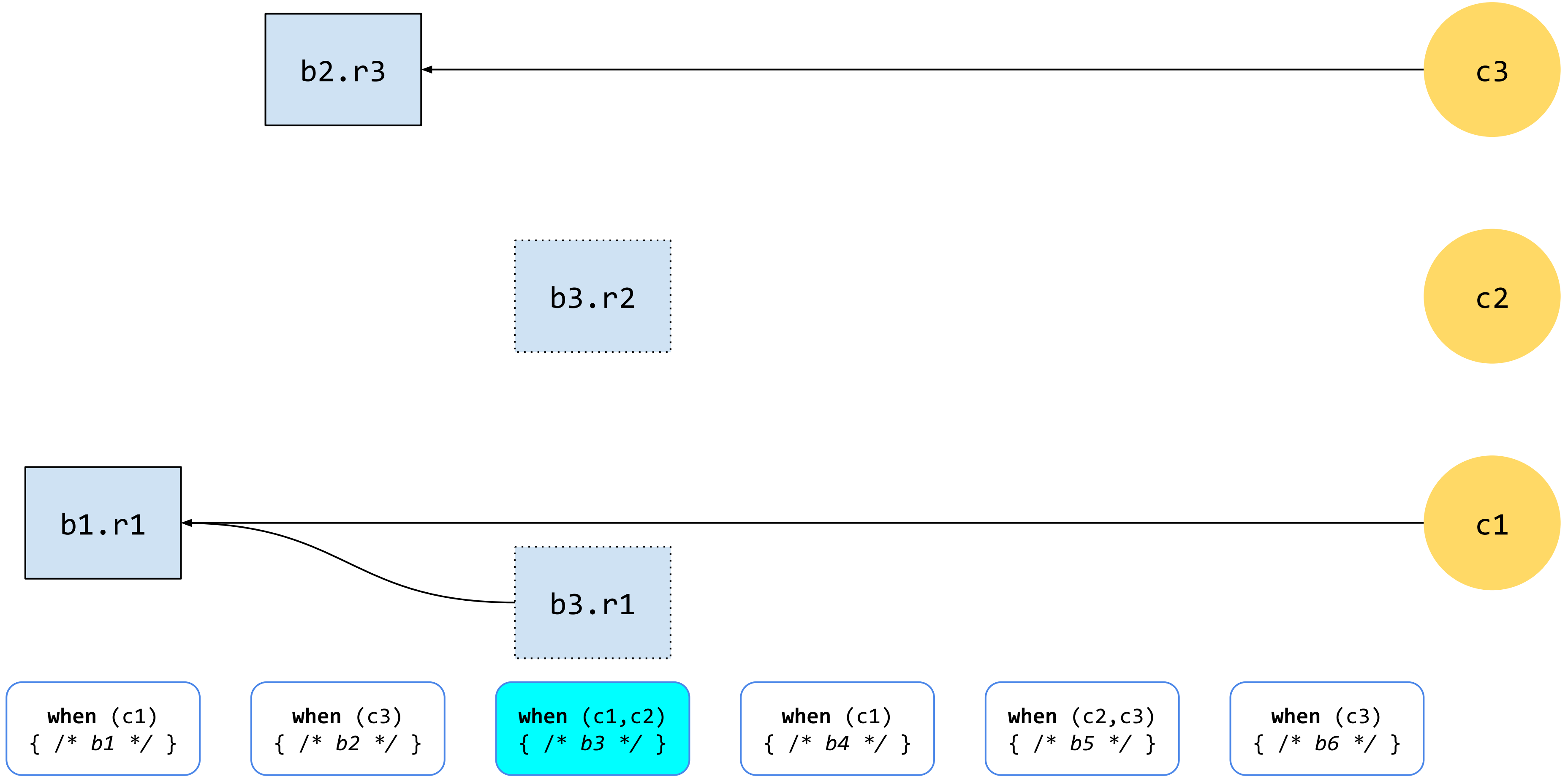
```
when (c1)
{ /* b4 */ }
```

```
when (c2,c3)
{ /* b5 */ }
```

```
when (c3)
{ /* b6 */ }
```



```
when (c1)      when (c3)      when (c1,c2)   when (c1)      when (c2,c3)   when (c3)
{ /* b1 */ }   { /* b2 */ }   { /* b3 */ }   { /* b4 */ }   { /* b5 */ }   { /* b6 */ }
```



```
when (c1)
{ /* b1 */ }
```

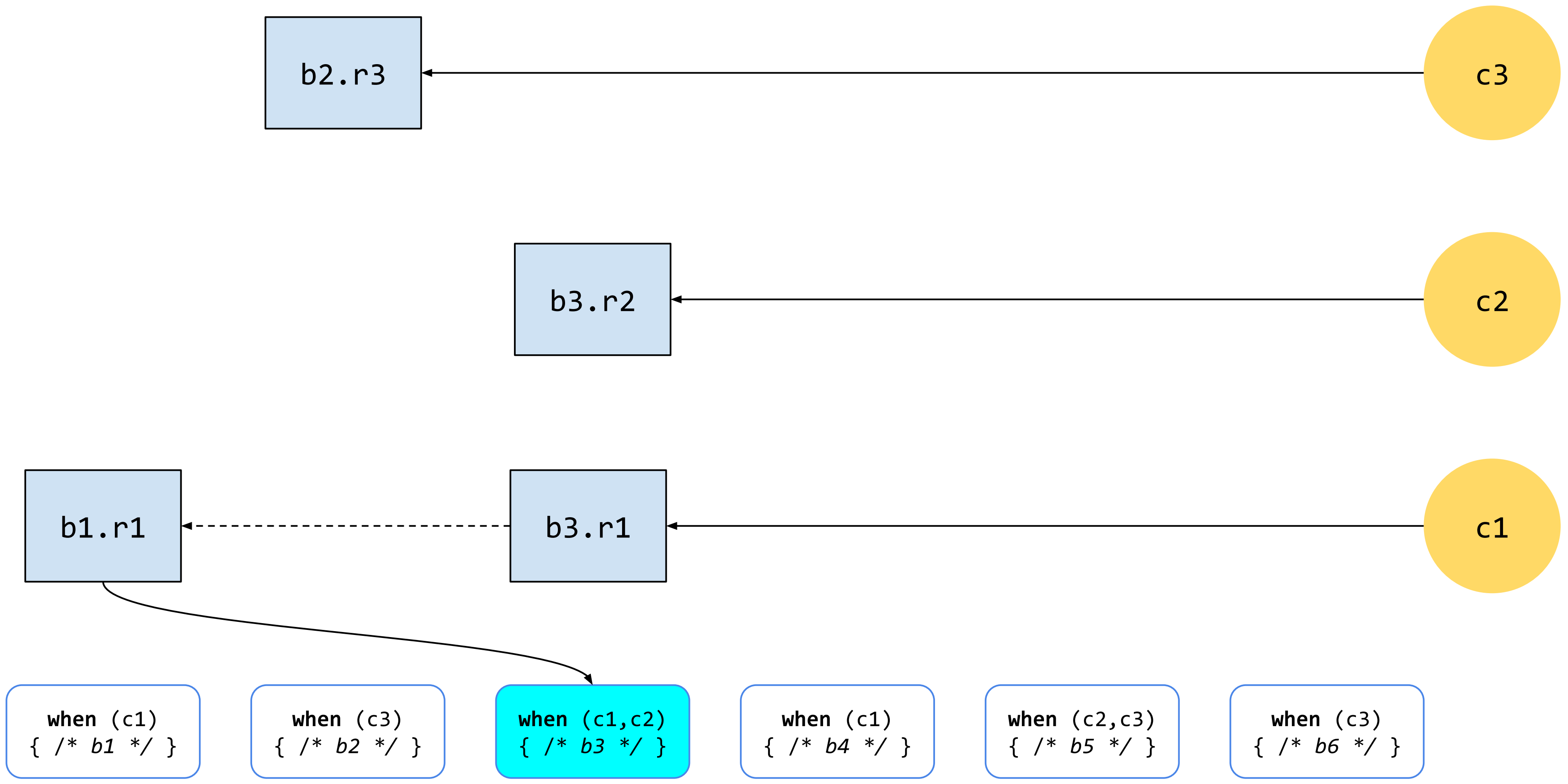
```
when (c3)
{ /* b2 */ }
```

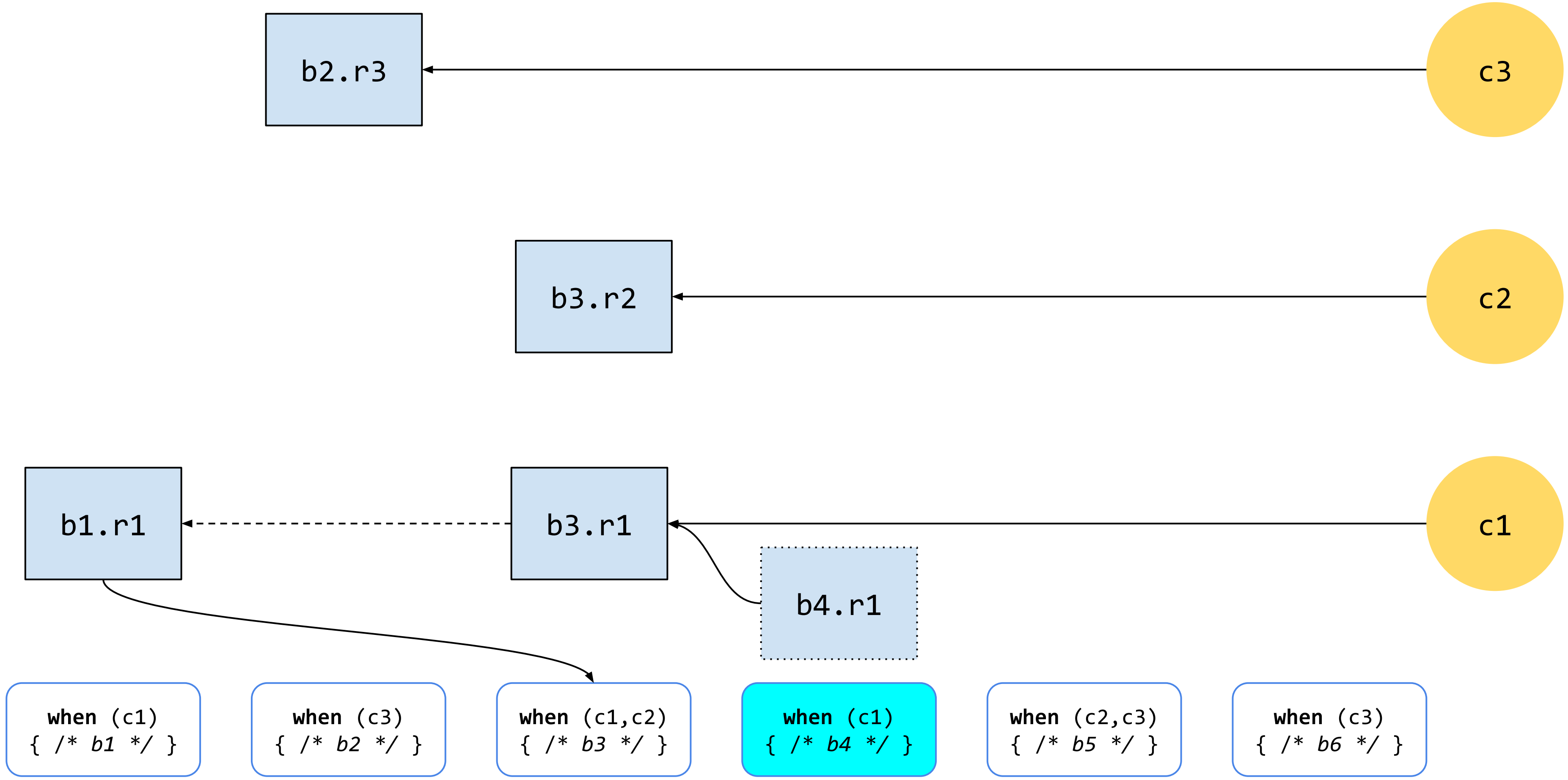
```
when (c1,c2)
{ /* b3 */ }
```

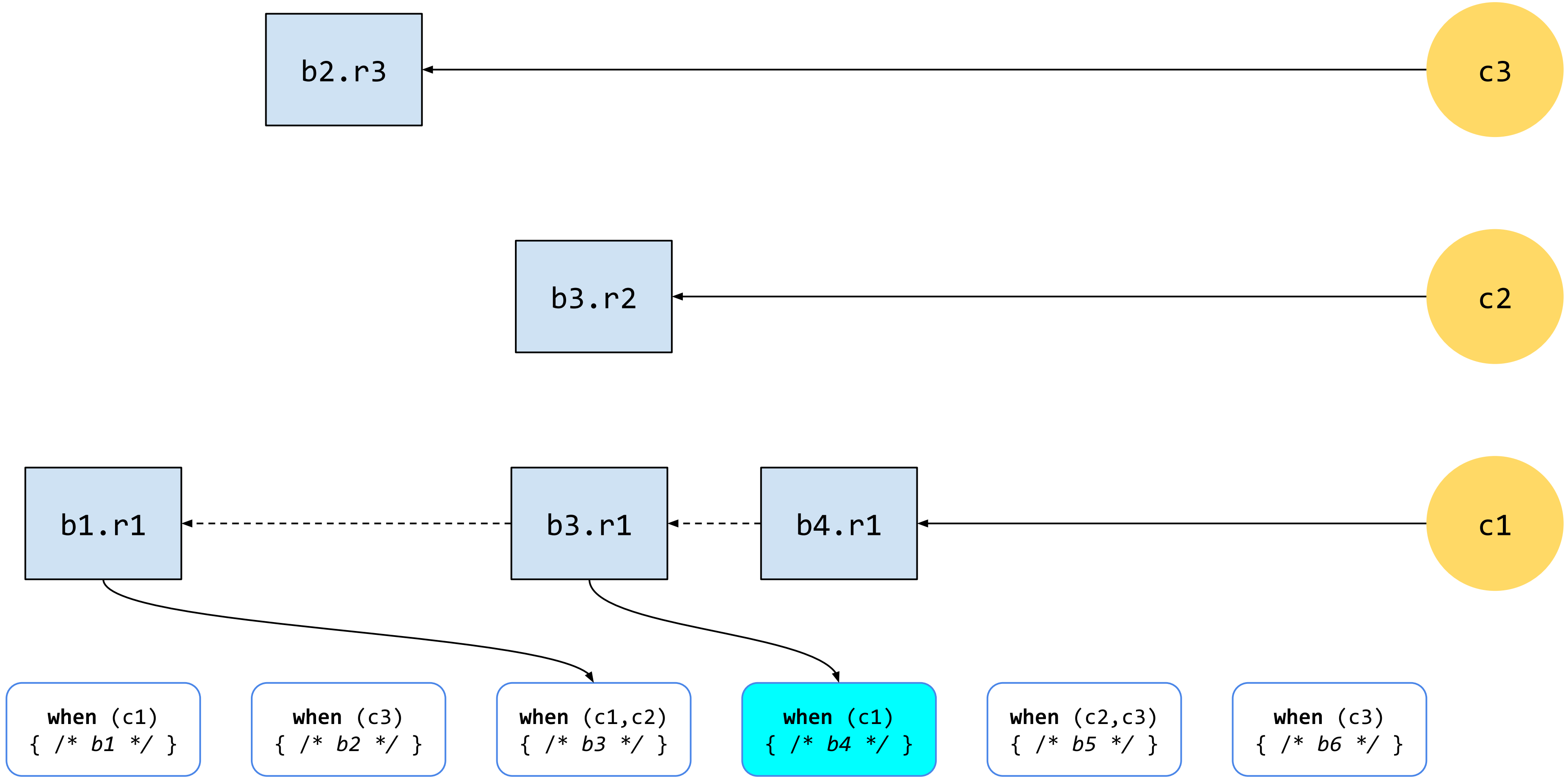
```
when (c1)
{ /* b4 */ }
```

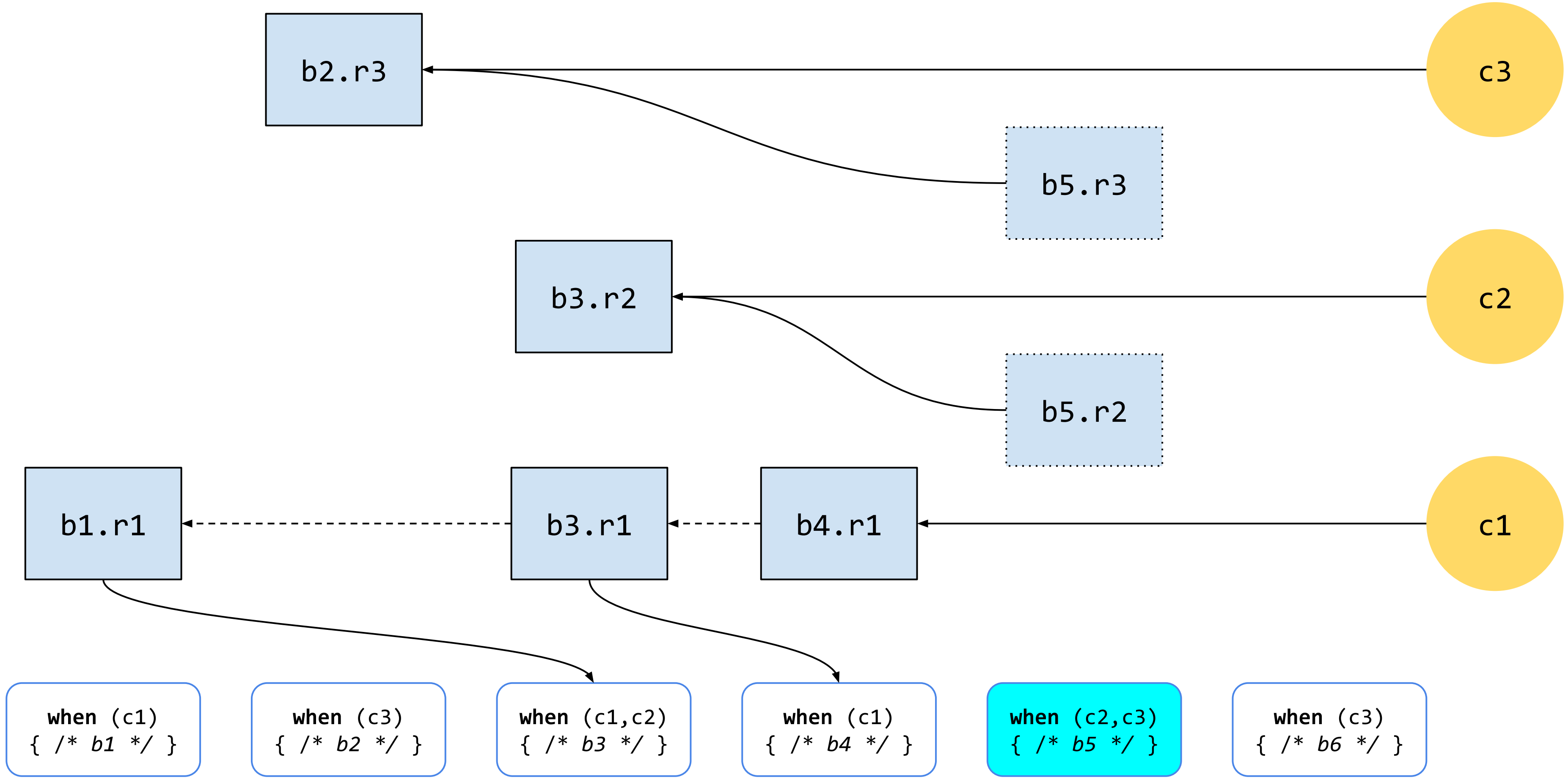
```
when (c2,c3)
{ /* b5 */ }
```

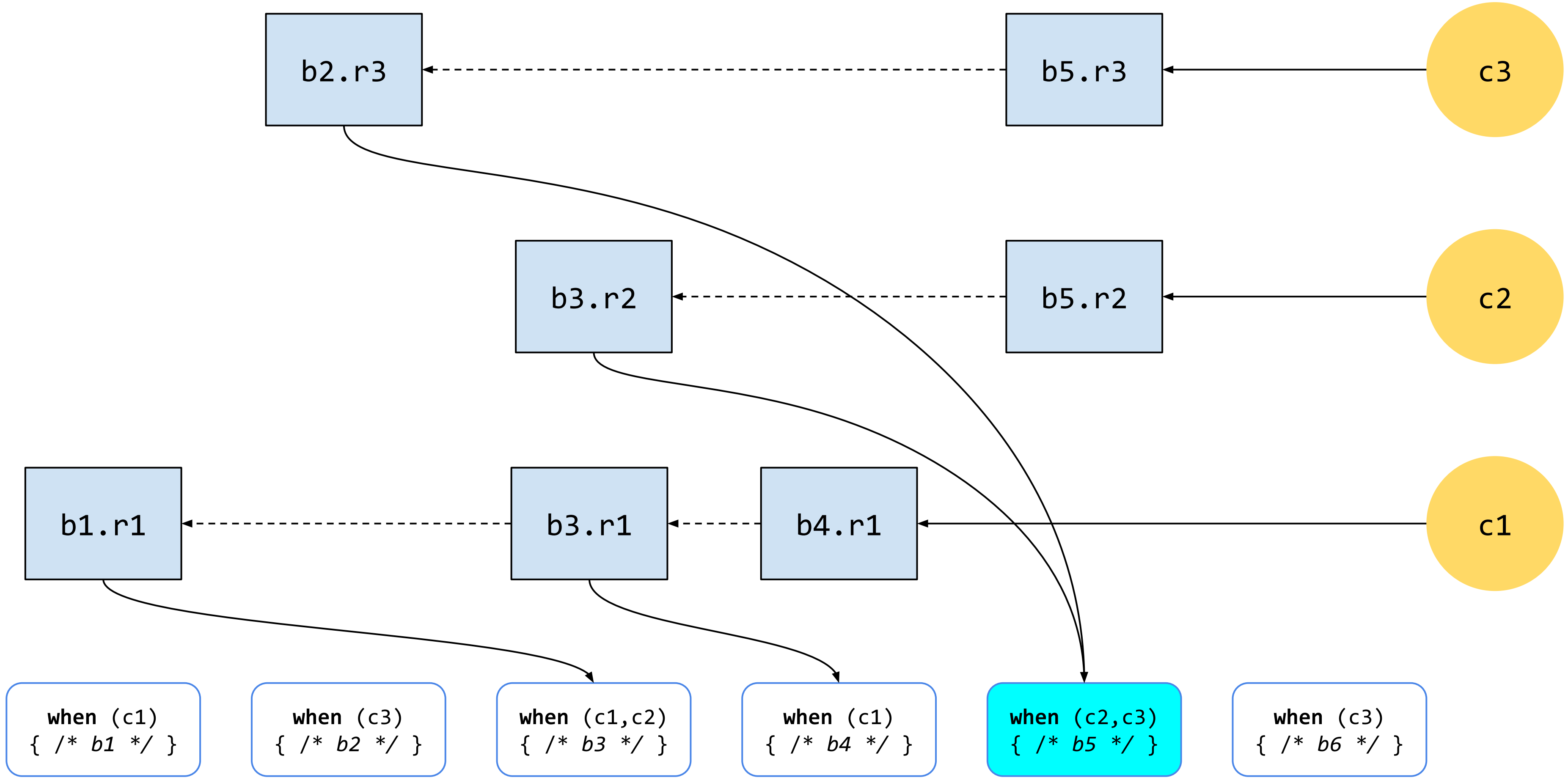
```
when (c3)
{ /* b6 */ }
```

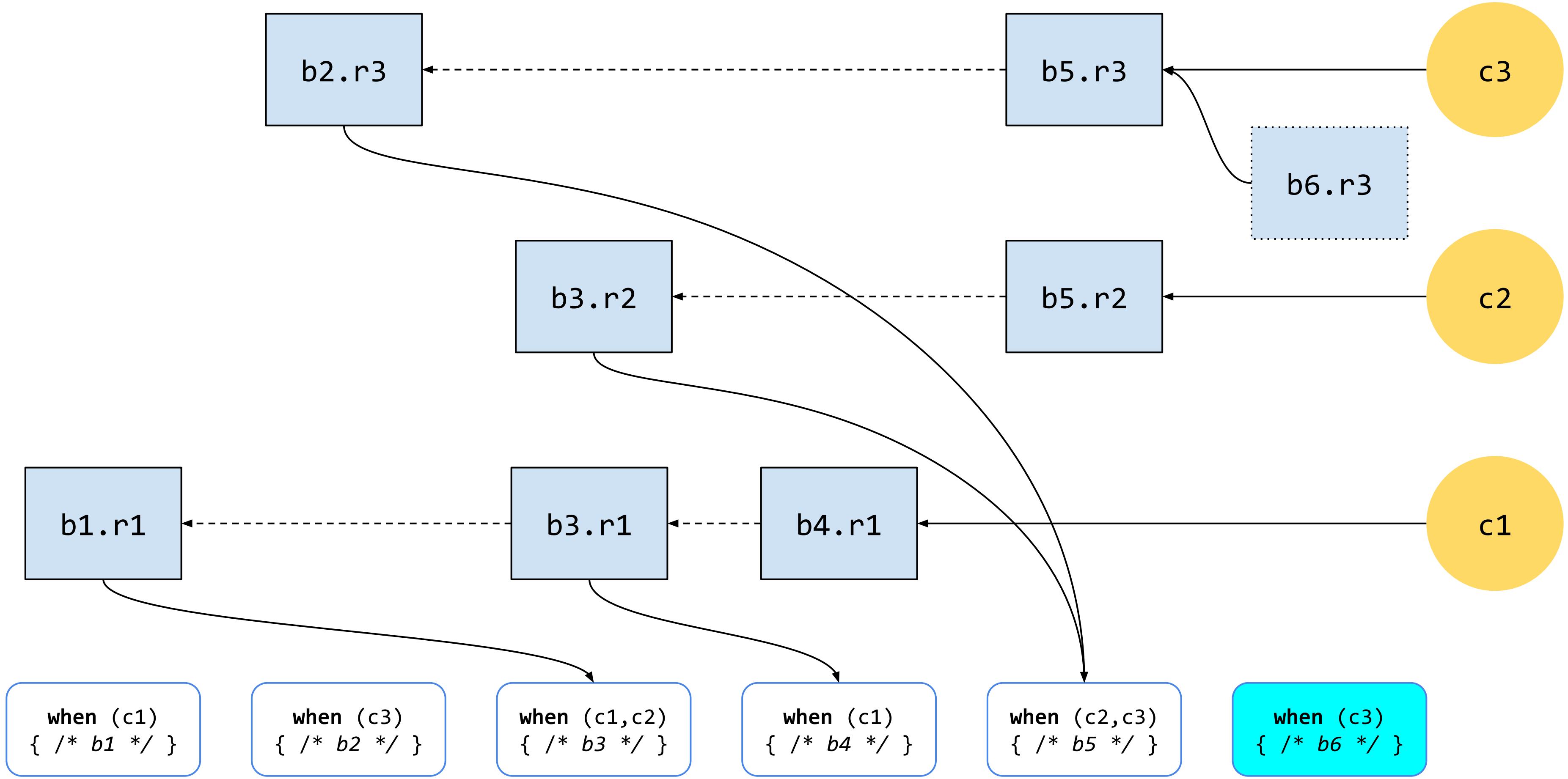

b2.r3
c3
b3.r2
c2
b1.r1
b3.r1
c1
when (c1) { /* b1 */ }
when (c3) { /* b2 */ }
when (c1,c2) { /* b3 */ }
when (c1) { /* b4 */ }
when (c2,c3) { /* b5 */ }
when (c3) { /* b6 */ }


b2.r3
c3
b3.r2
c2
b1.r1
b3.r1
c1
b4.r1
when (c1) { /* b1 */ }
when (c3) { /* b2 */ }
when (c1,c2) { /* b3 */ }
when (c1) { /* b4 */ }
when (c2,c3) { /* b5 */ }
when (c3) { /* b6 */ }


b2.r3
c3
b3.r2
c2
b1.r1
b3.r1
b4.r1
c1
when (c1) { /* b1 */ }
when (c3) { /* b2 */ }
when (c1,c2) { /* b3 */ }
when (c1) { /* b4 */ }
when (c2,c3) { /* b5 */ }
when (c3) { /* b6 */ }


b2.r3
c3
b5.r3
b3.r2
c2
b5.r2
b1.r1
b3.r1
b4.r1
c1
when (c1) { /* b1 */ }
when (c3) { /* b2 */ }
when (c1,c2) { /* b3 */ }
when (c1) { /* b4 */ }
when (c2,c3) { /* b5 */ }
when (c3) { /* b6 */ }


b2.r3
b5.r3
c3
b3.r2
b5.r2
c2
b1.r1
b3.r1
b4.r1
c1
when (c1) { /* b1 */ }
when (c3) { /* b2 */ }
when (c1,c2) { /* b3 */ }
when (c1) { /* b4 */ }
when (c2,c3) { /* b5 */ }
when (c3) { /* b6 */ }



```
when (c1)
{ /* b1 */ }
```

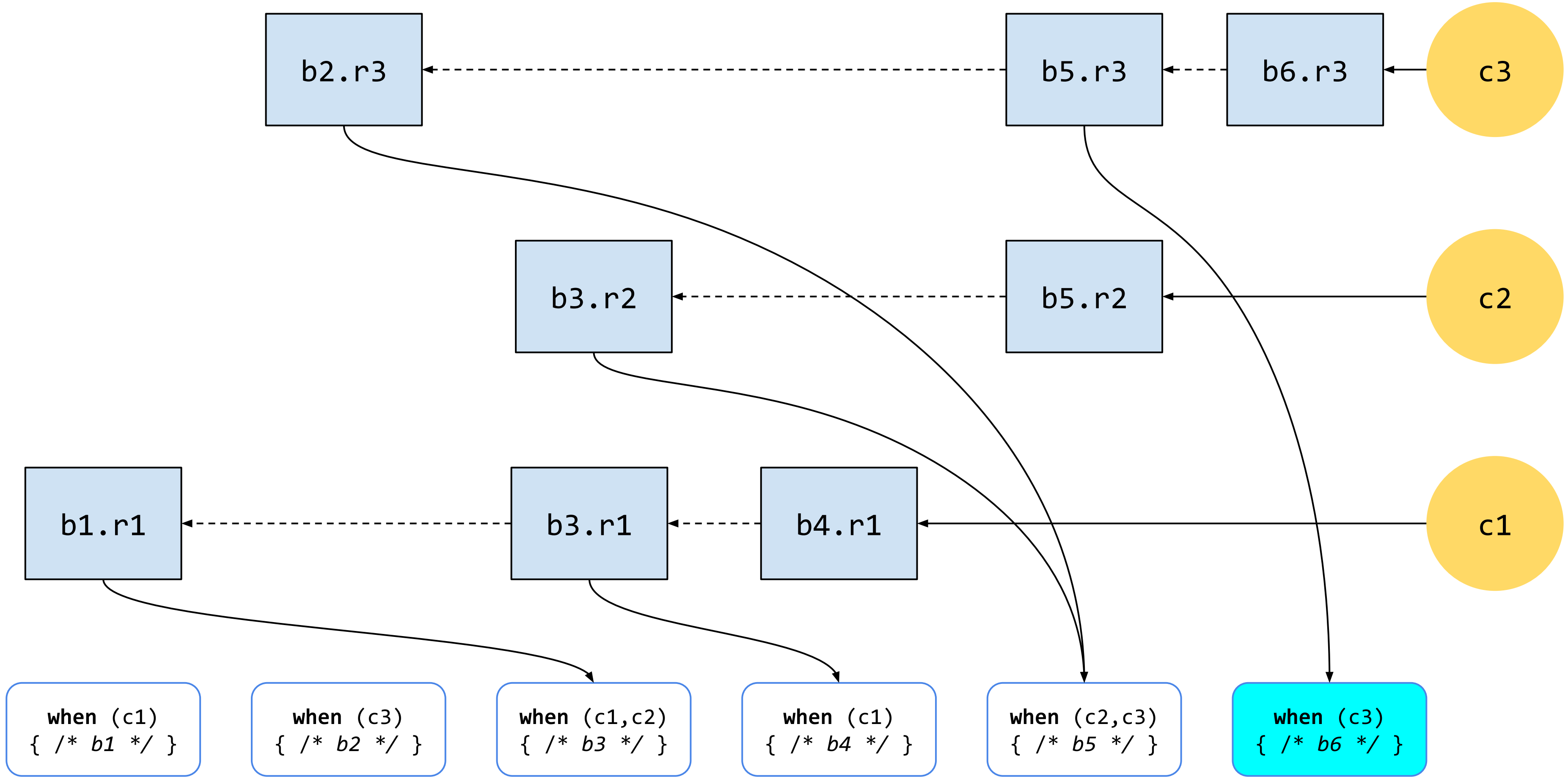
```
when (c3)
{ /* b2 */ }
```

```
when (c1,c2)
{ /* b3 */ }
```

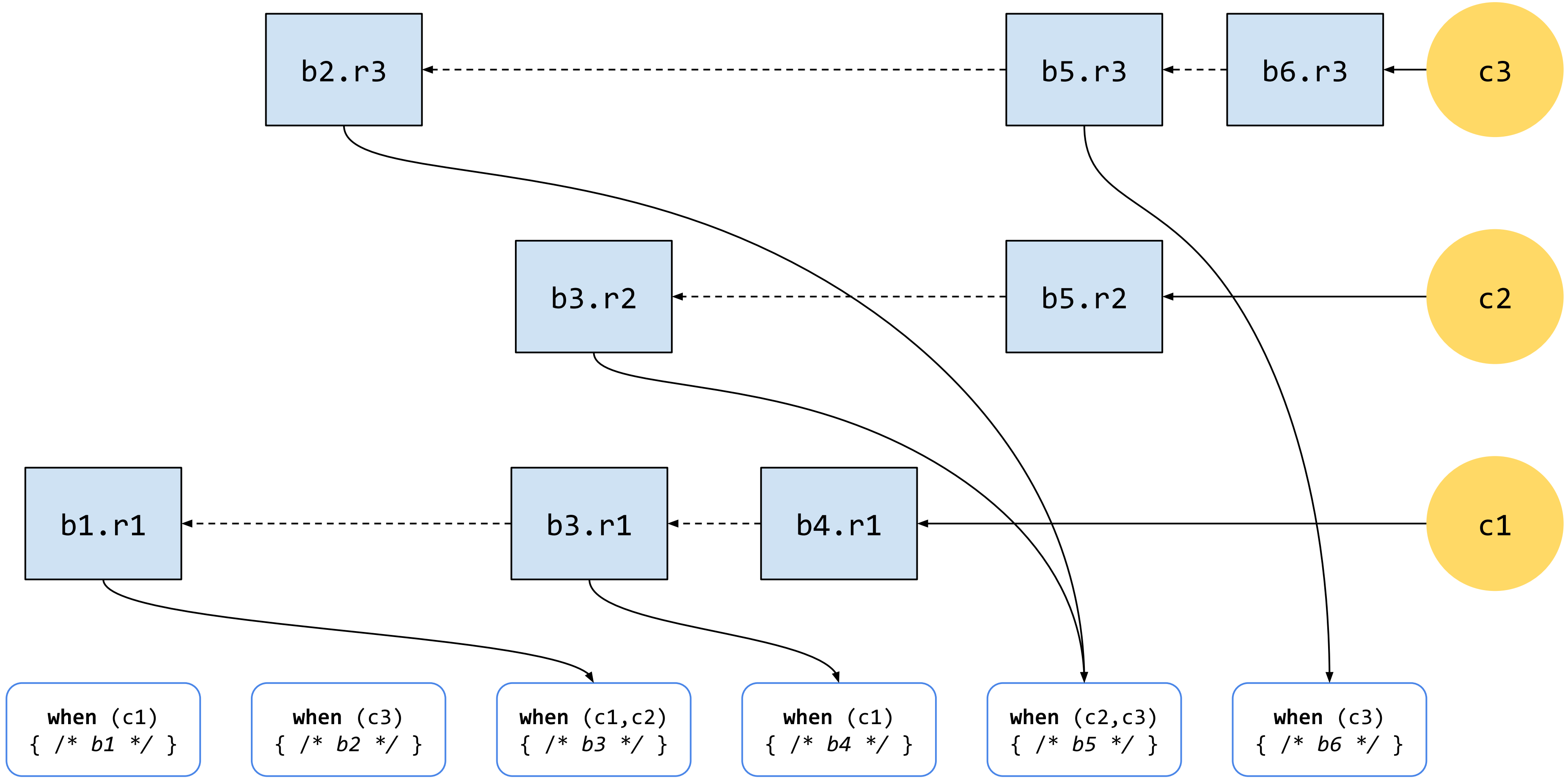
```
when (c1)
{ /* b4 */ }
```

```
when (c2,c3)
{ /* b5 */ }
```

```
when (c3)
{ /* b6 */ }
```



```
when (c1)
{ /* b1 */ }

when (c3)
{ /* b2 */ }

when (c1,c2)
{ /* b3 */ }

when (c1)
{ /* b4 */ }

when (c2,c3)
{ /* b5 */ }

when (c3)
{ /* b6 */ }
```



```
when (c1)
{ /* b1 */ }
```

```
when (c3)
{ /* b2 */ }
```

```
when (c1,c2)
{ /* b3 */ }
```

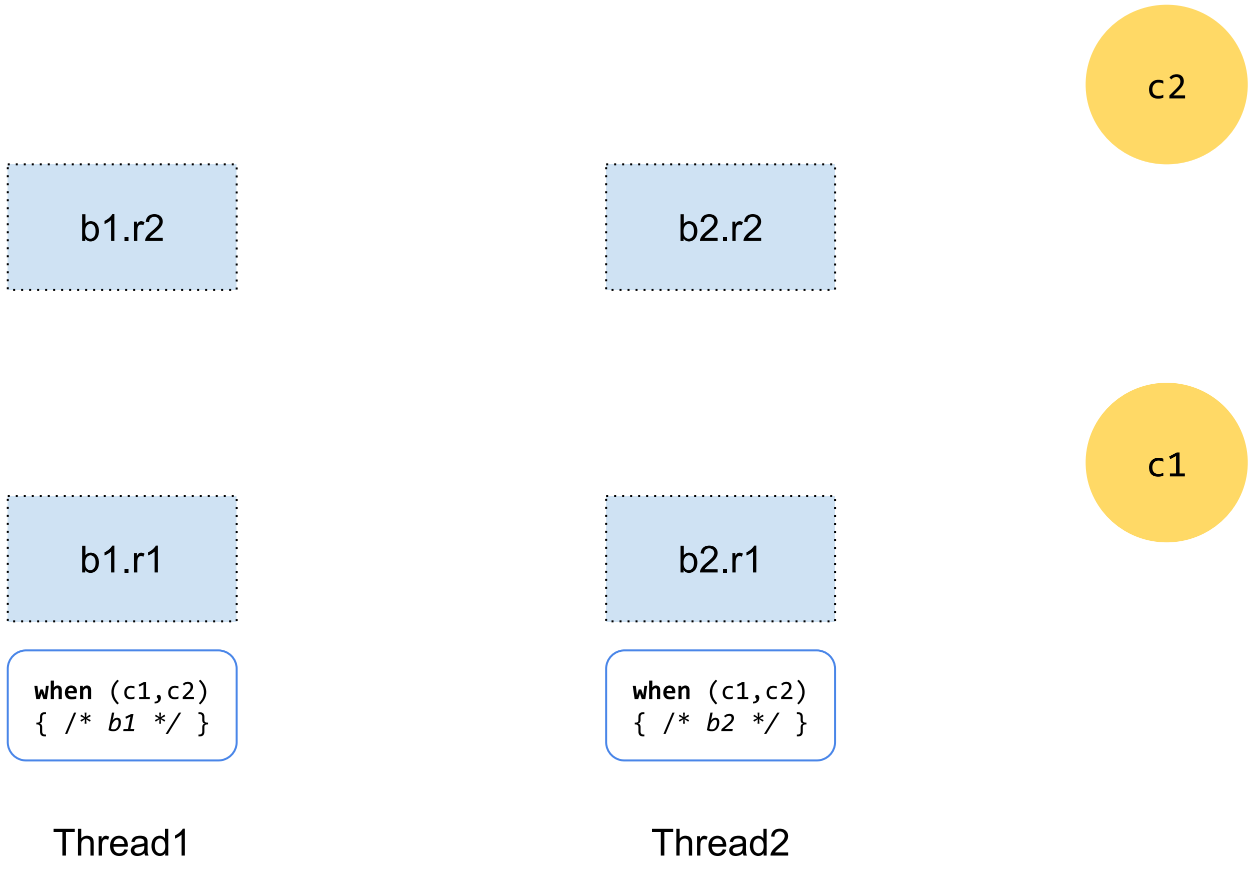
```
when (c1)
{ /* b4 */ }
```

```
when (c2,c3)
{ /* b5 */ }
```

```
when (c3)
{ /* b6 */ }
```

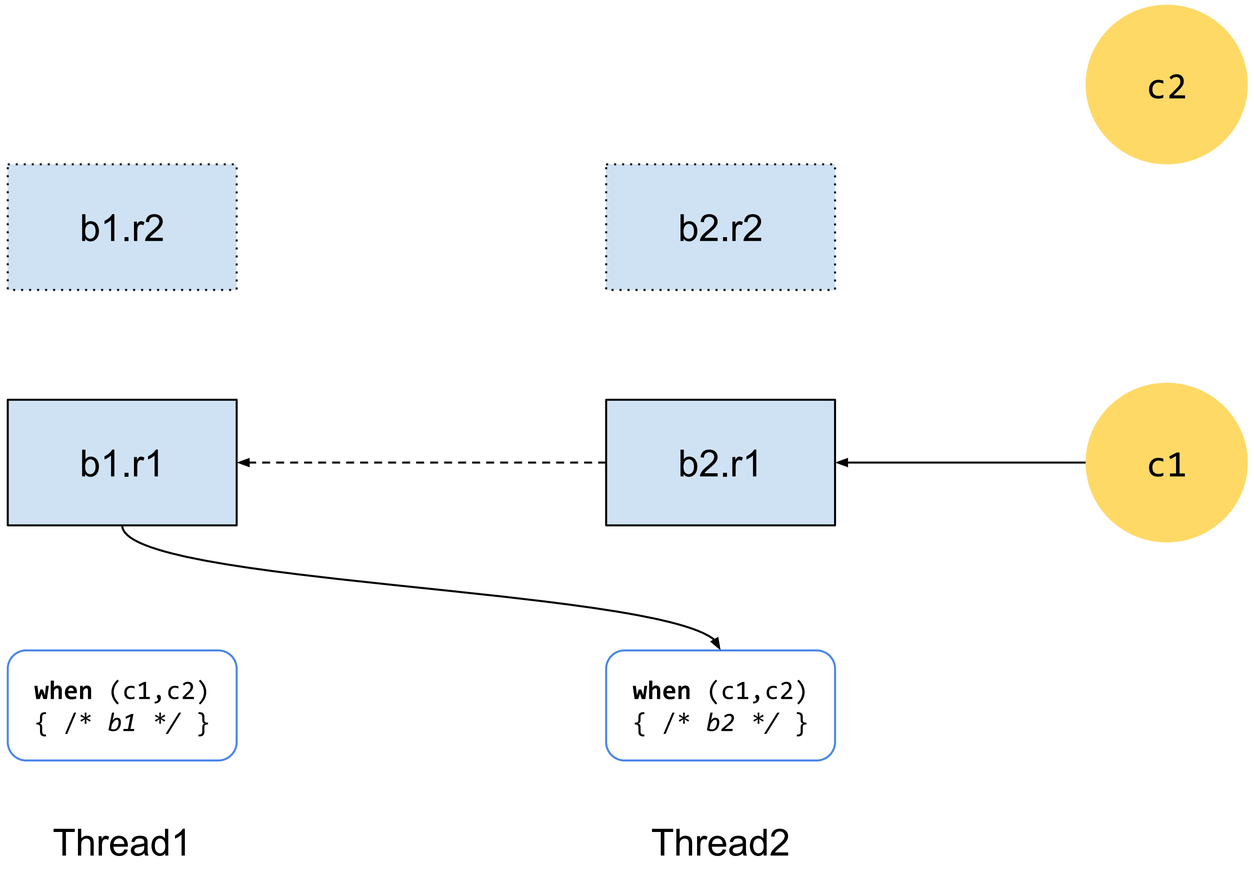
Implementation with lock

- Additional count
- Scheduled flag



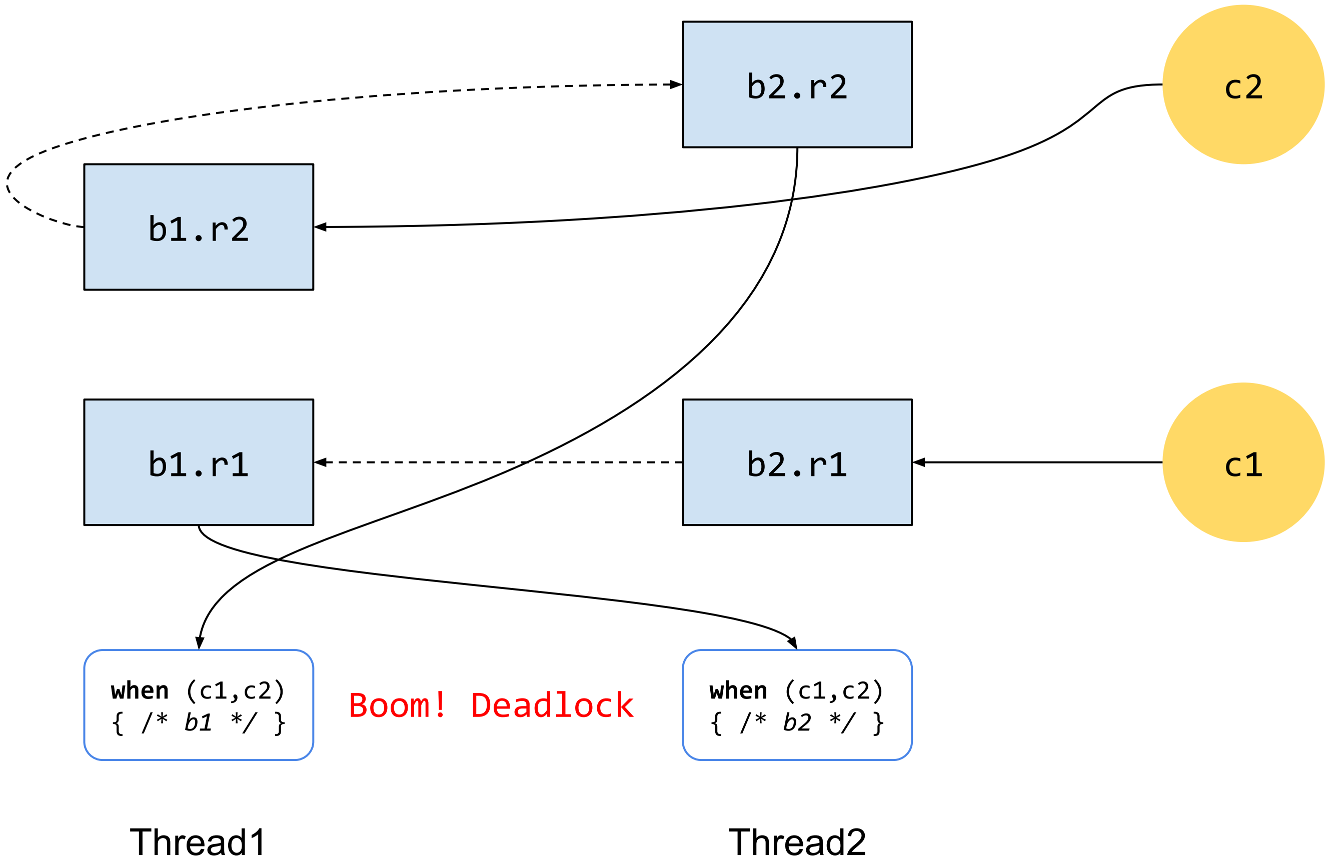
```
when (c1,c2)
{ /* b1 */ }
```

```
when (c1,c2)
{ /* b2 */ }
```



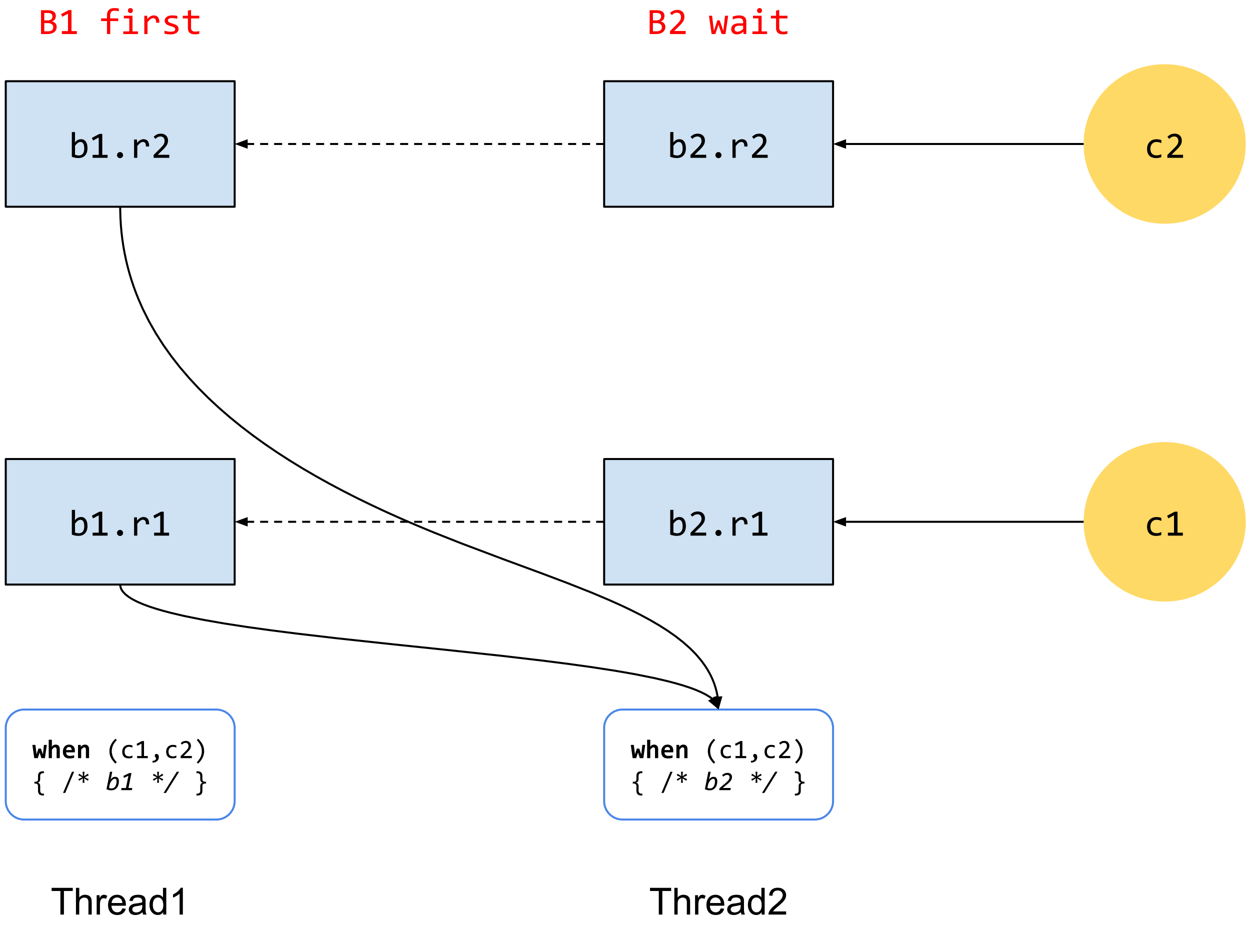
```
when (c1,c2)
{ /* b1 */ }
```

```
when (c1,c2)
{ /* b2 */ }
```



```
when (c1,c2)
{ /* b1 */ }
```

```
when (c1,c2)
{ /* b2 */ }
```


B1 first
B2 wait
b1.r2
b2.r2
c2
b1.r1
b2.r1
c1
when (c1,c2) { /* b1 */ }
when (c1,c2) { /* b2 */ }
Thread1
Thread2

Implementation without lock

- Behaviour, Request, and Cown all on heap
- Pin semantics

Related topic

- Actor
- Transaction
- Distribute Programming

Thanks for watching!