Verifying Concurrent Memory Reclamation Algorithms with Grace

+

From Hoare Logic to Separation Logic

Noam Rinetzky Tel Aviv University, Israel

Technion, 1/July/2013

Verifying Concurrent Memory Reclamation Algorithms with Grace

European Symposium on Programming (ESOP) 2013

Alexey Gotsman	IMDEA, Spain
Noam Rinetzky	Tel Aviv University, Israel
Hongseok Yang	University of Oxford, UK

Technion, 1/July/2013

research problem

- verify non-blocking data structures with explicit memory management

non-blocking concurrent data structures

- stack, queue, set
- highly concurrent
  - no global locks protecting the entire DS
  - fine grained concurrency

research problem

- verify non-blocking data structures with explicit memory management
  - memory safety
  - no memory leaks
  - no anomalies due to reallocation (ABA)
  - correctness (atomicity)

concurrency is challenging

- non-blocking concurrent data structures
  - stack, queue, set
- challenges
  - interference
  - pointers
  - dynamic allocation (malloc)

concurrency very is challenging

- non-blocking concurrent data structures
  - stack, queue, set
- challenges
  - interference
  - pointers
  - dynamic allocation (malloc)
  - **deallocation (free)**
    - **reallocation (free ; malloc)**

concurrent counter (with GC)

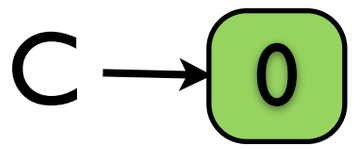
```
int *C;
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

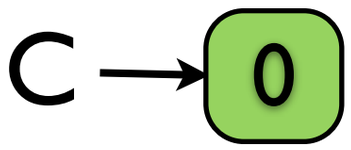



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

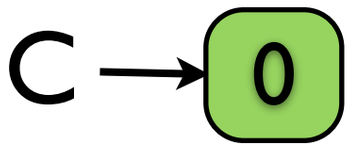



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```



v

s

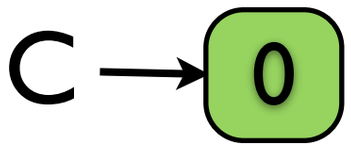
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```



v

s

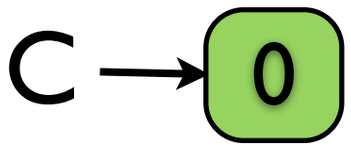
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

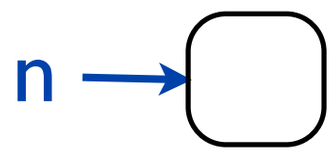
concurrent counter (with GC)

```
int *C;
```



v

s

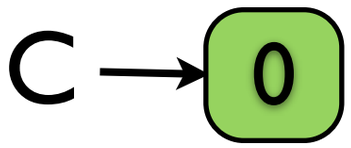



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

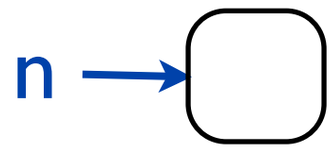
concurrent counter (with GC)

```
int *C;
```



v

s

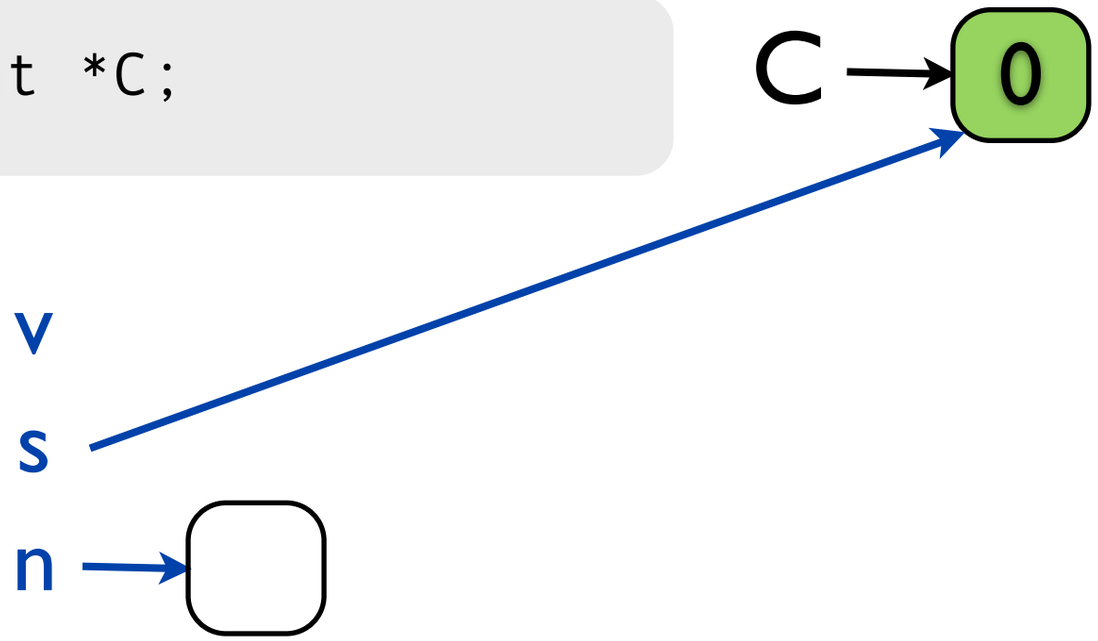



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

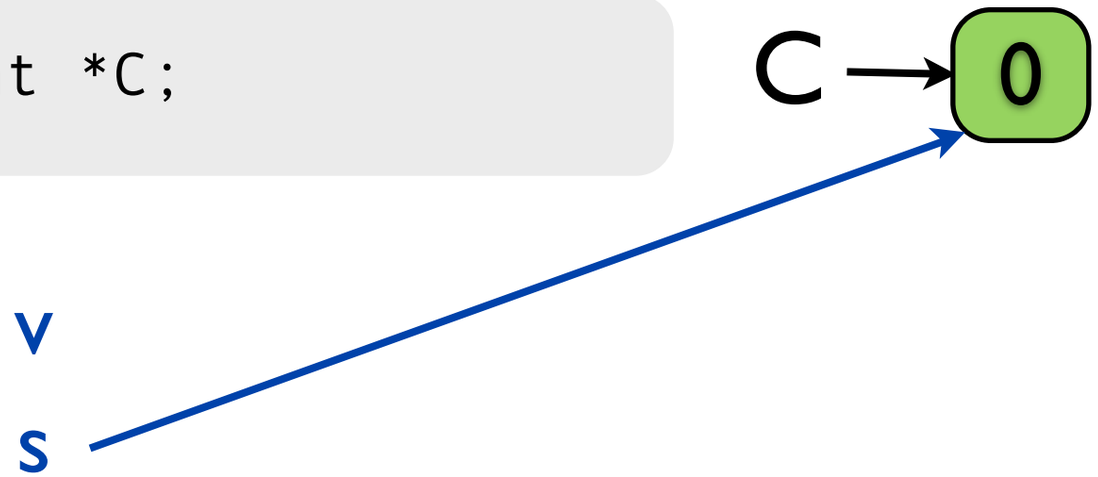



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

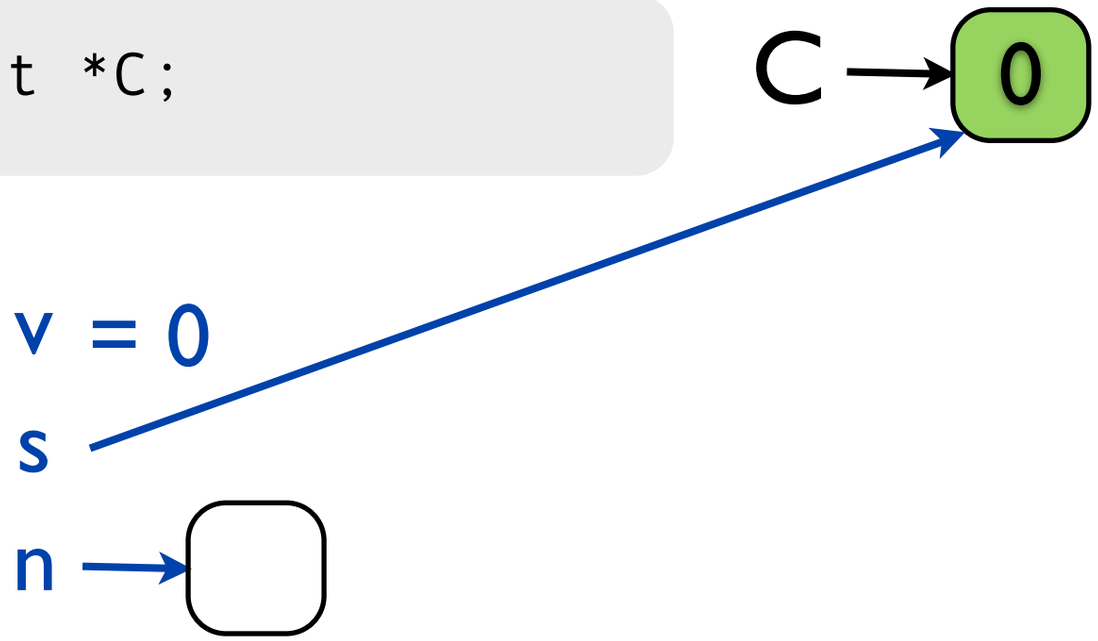



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

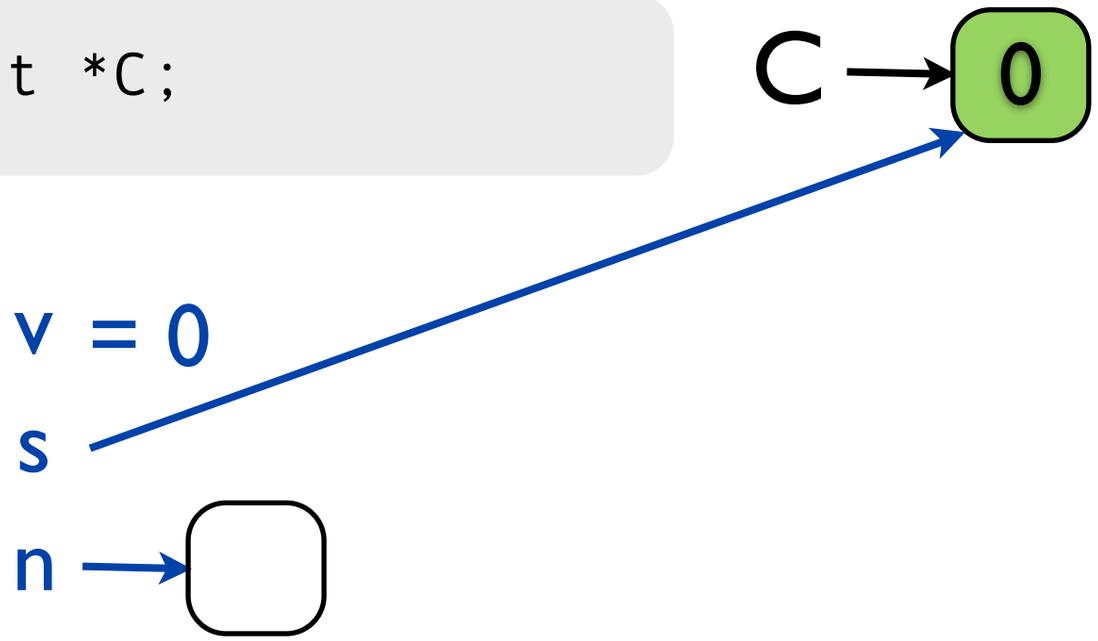



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

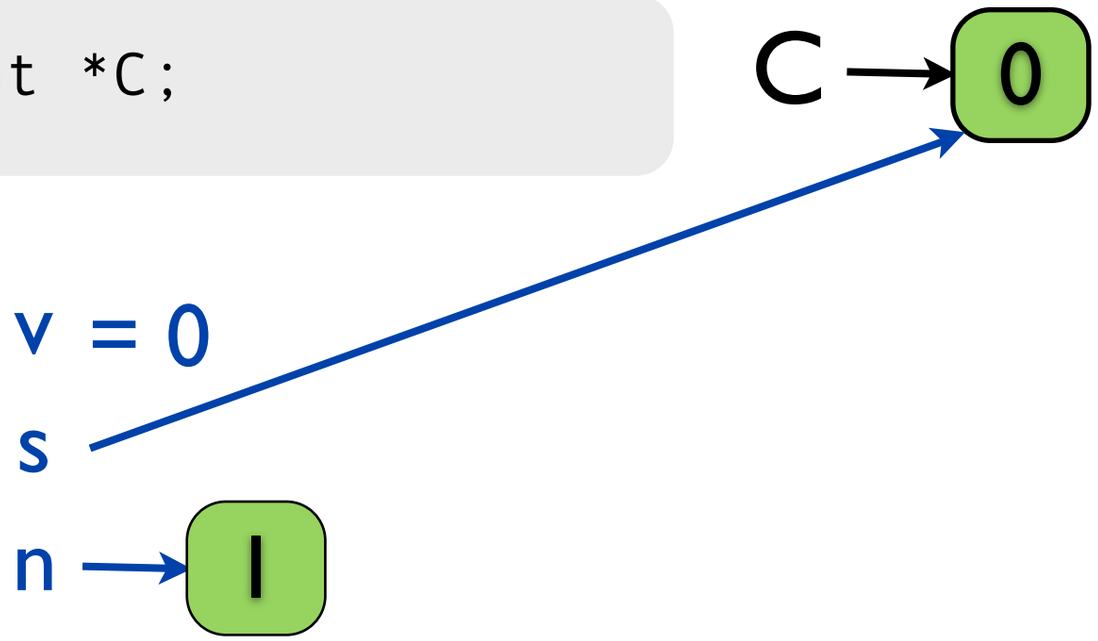



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

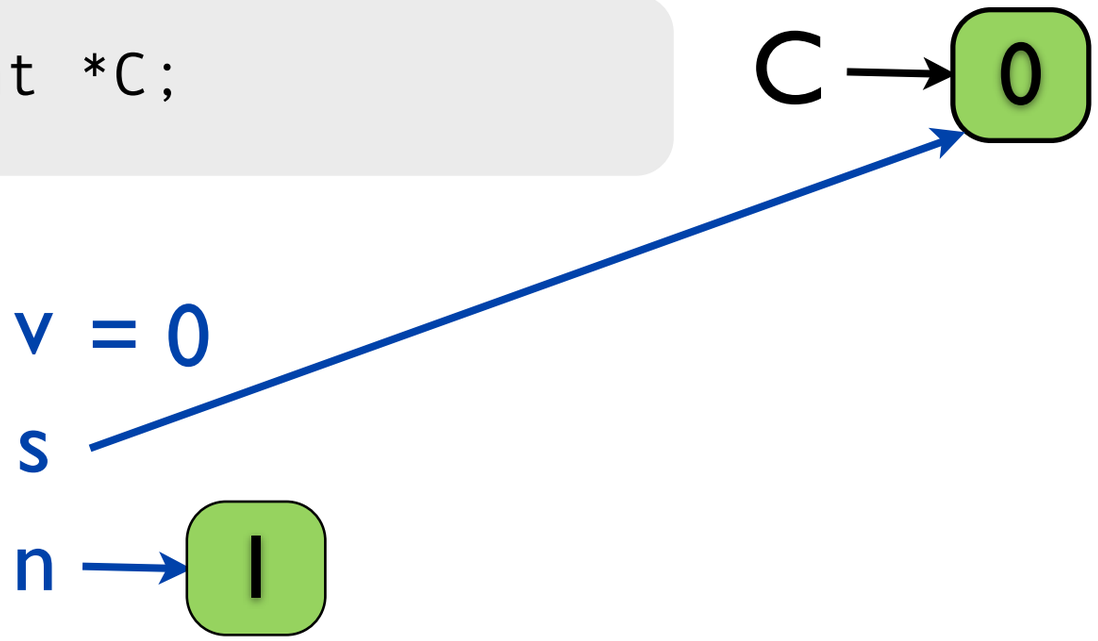



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

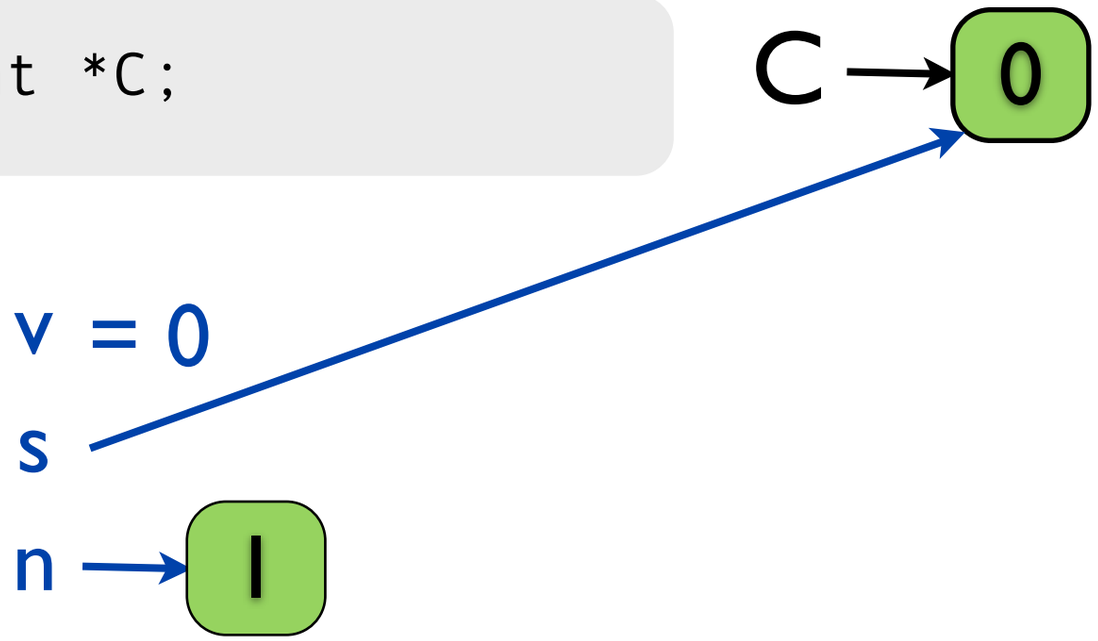



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```



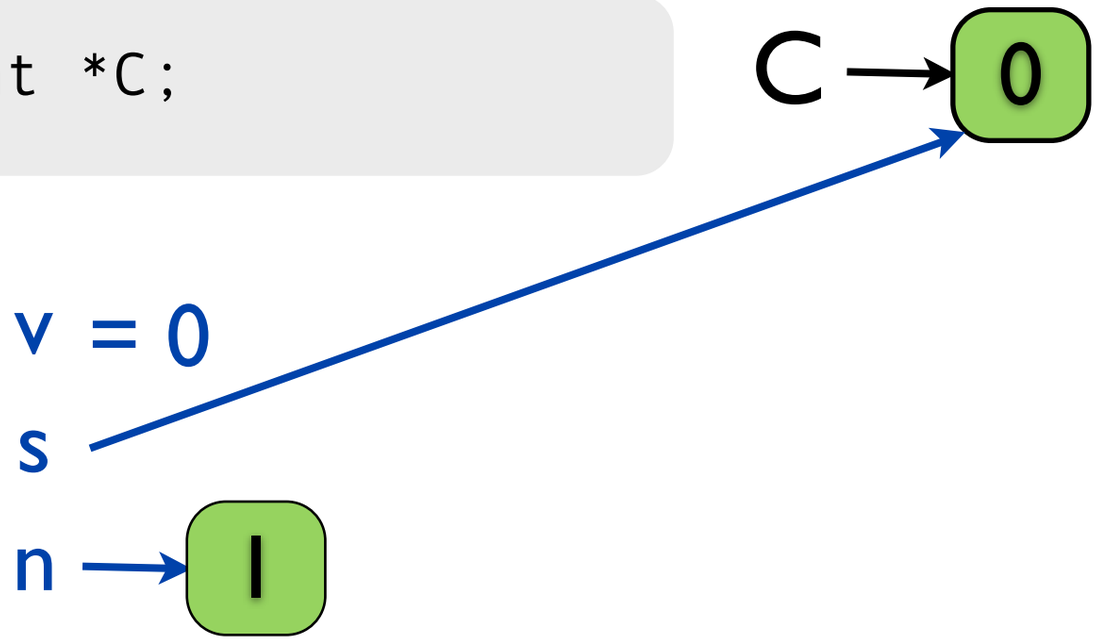

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

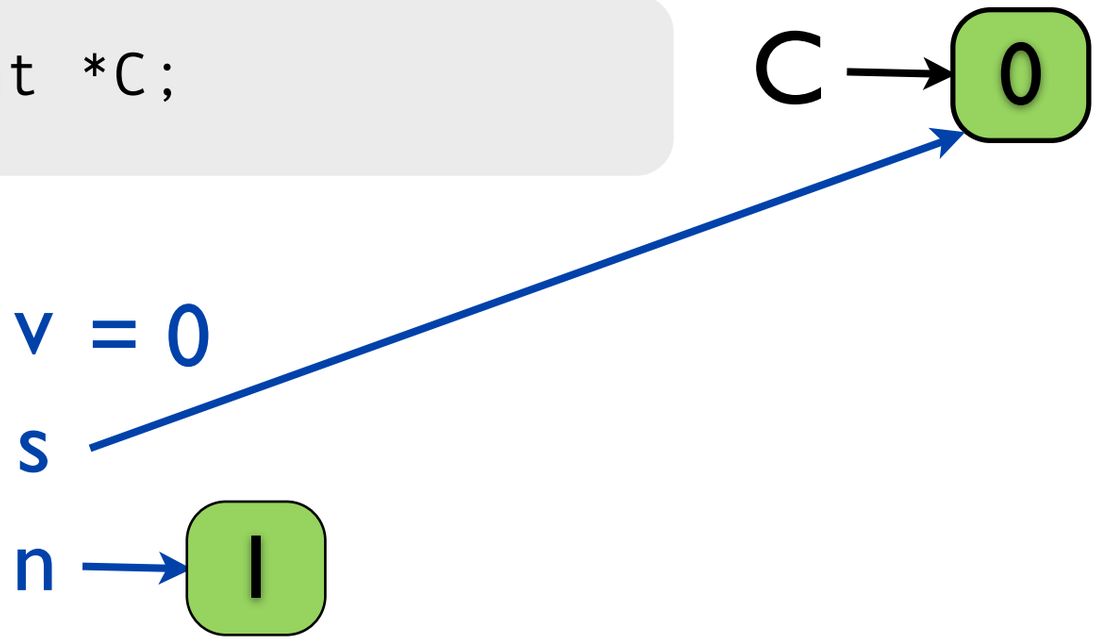
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

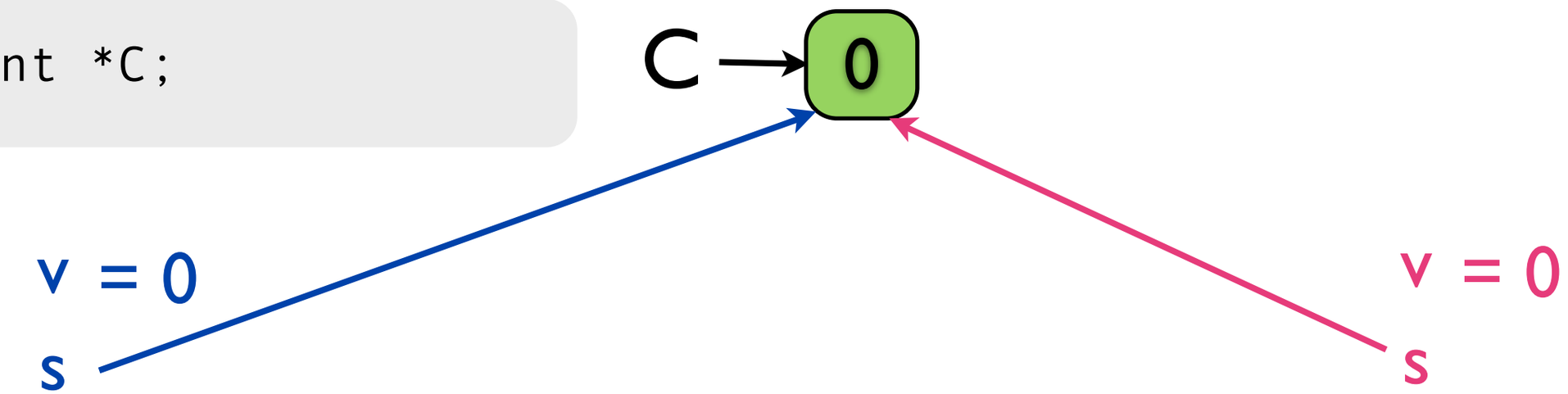
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

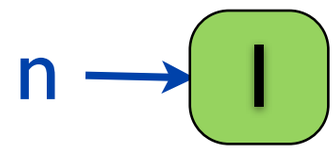
```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

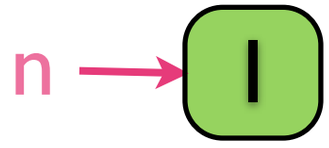





```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

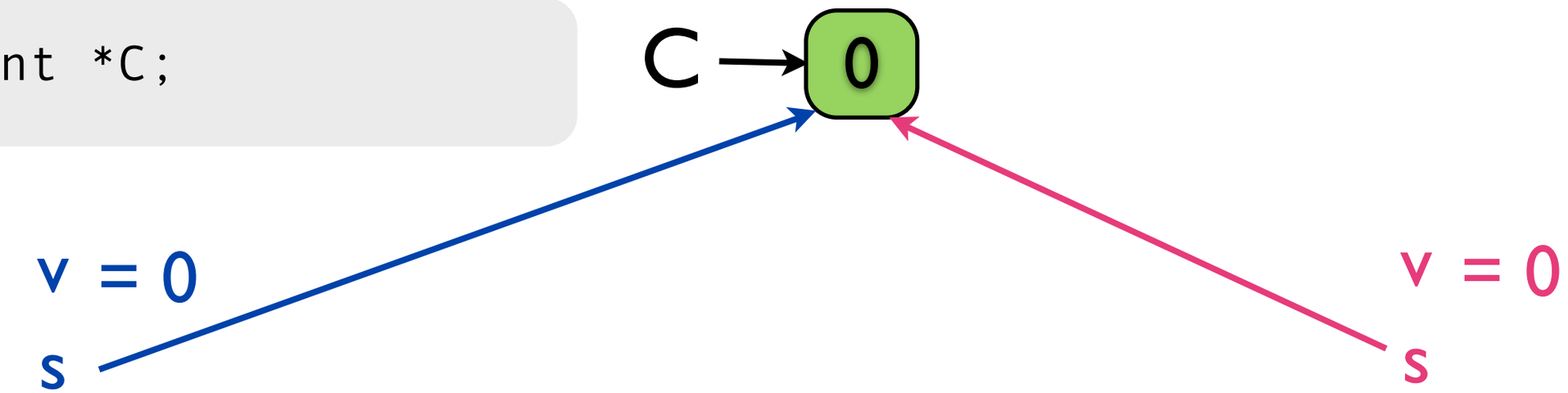


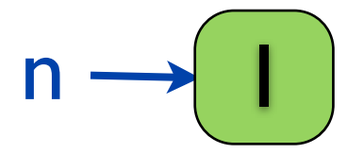

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```

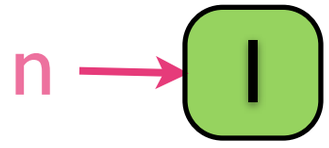





```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

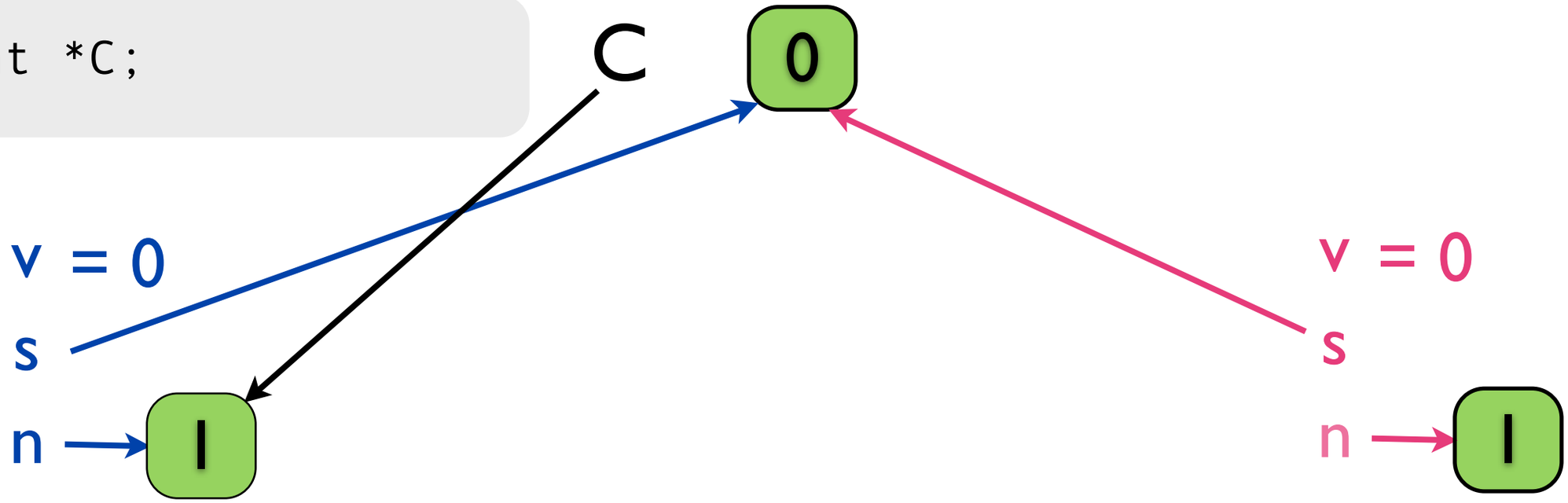



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

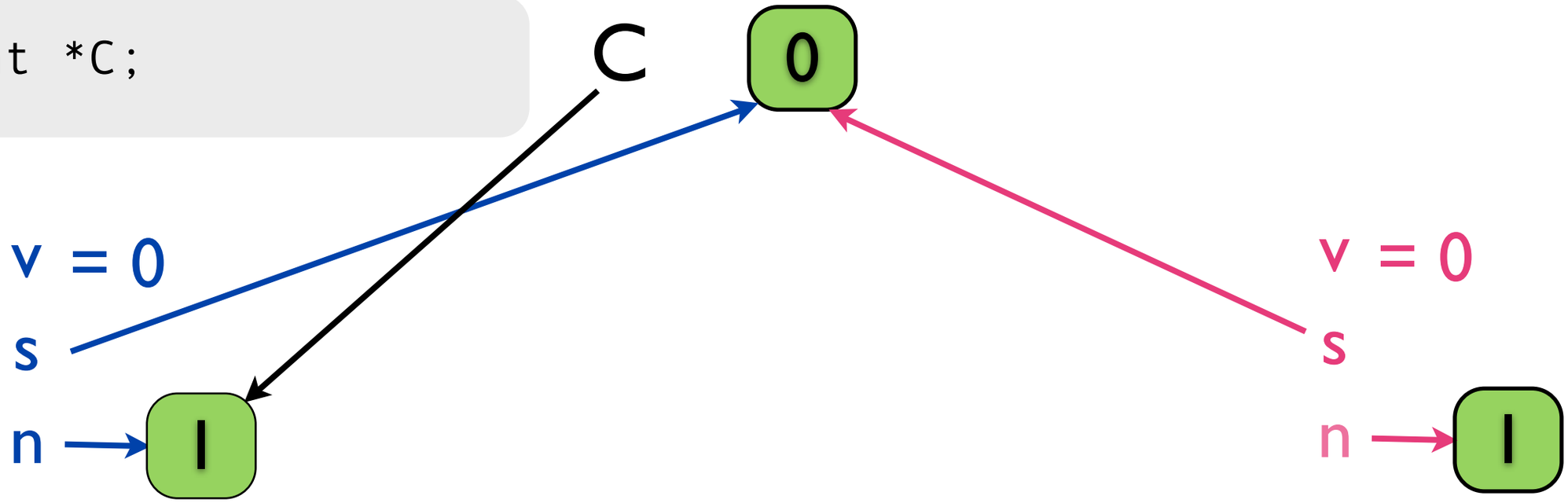
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

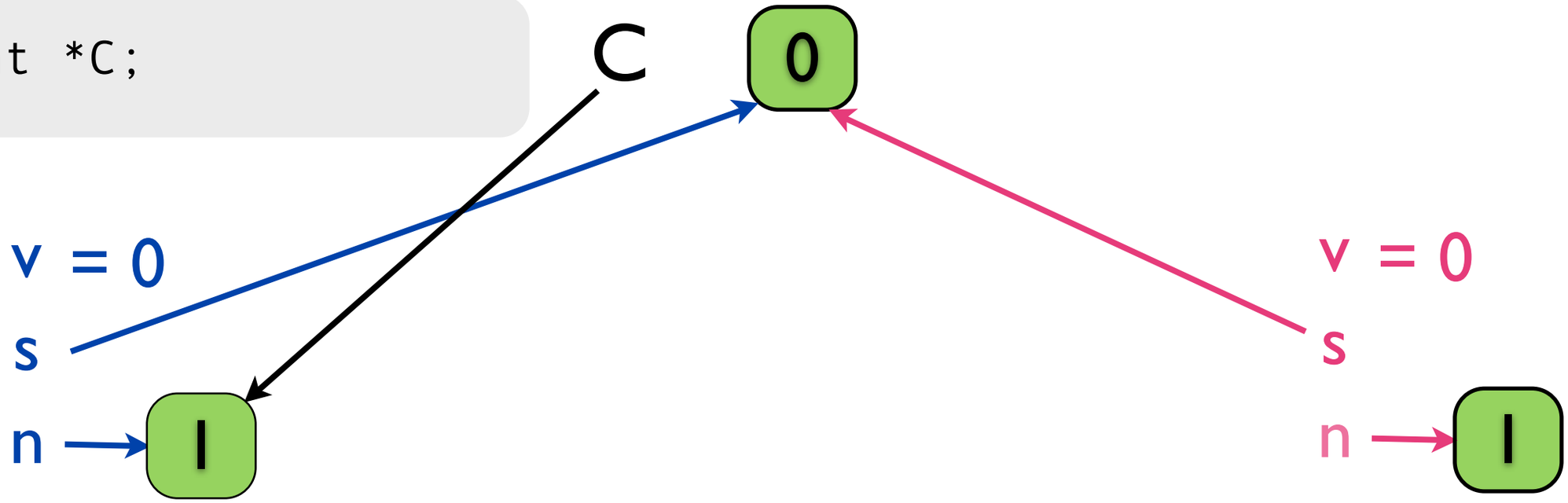
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

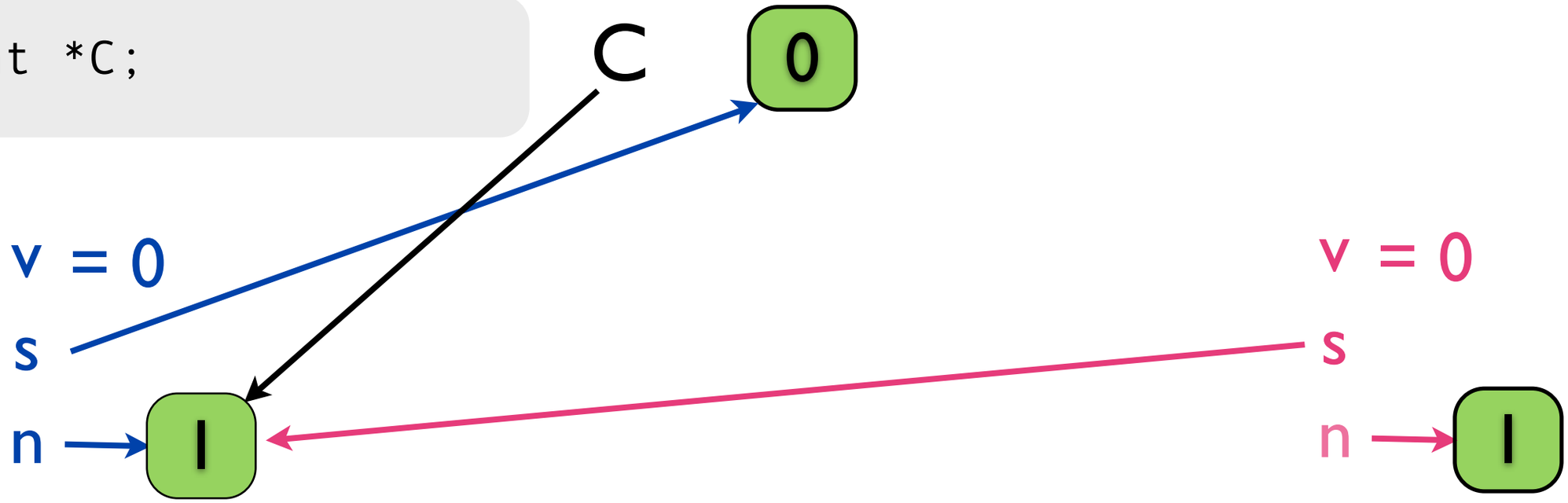
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

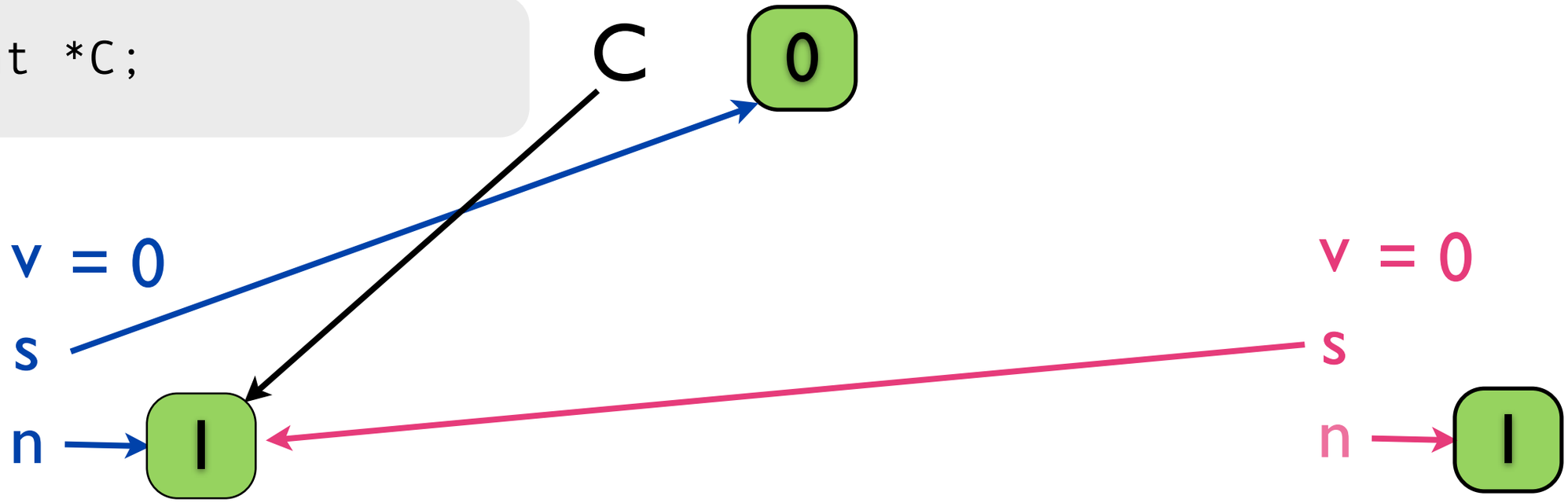
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

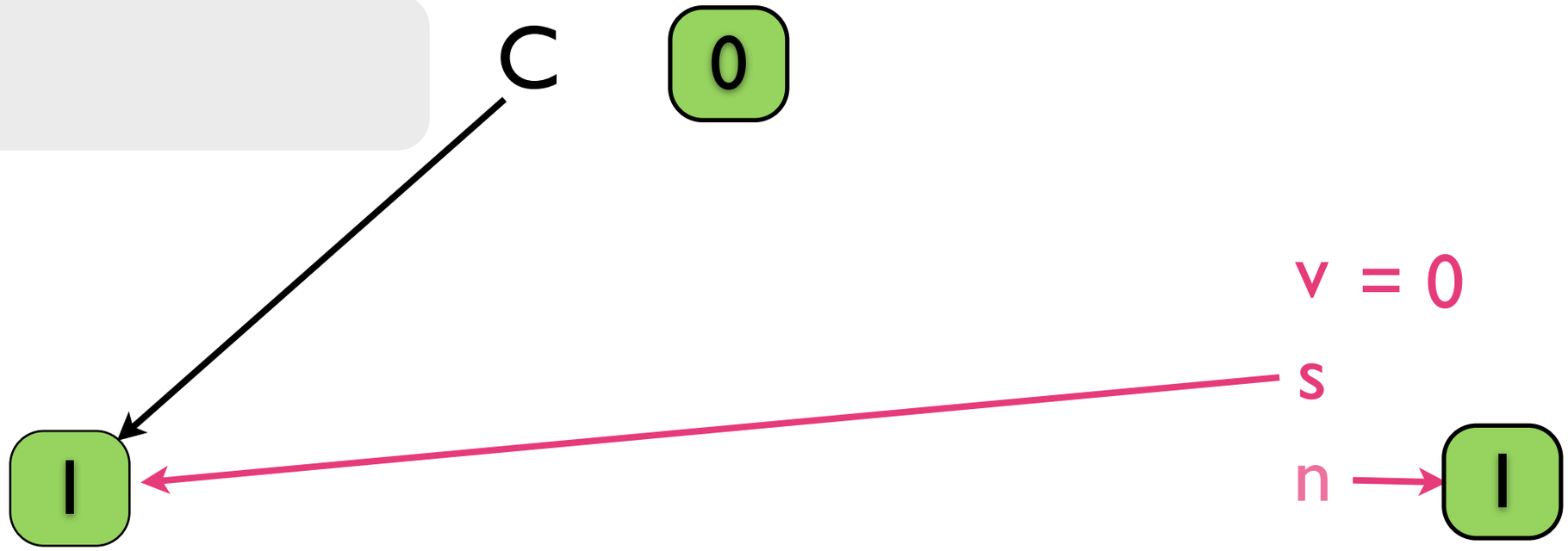
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

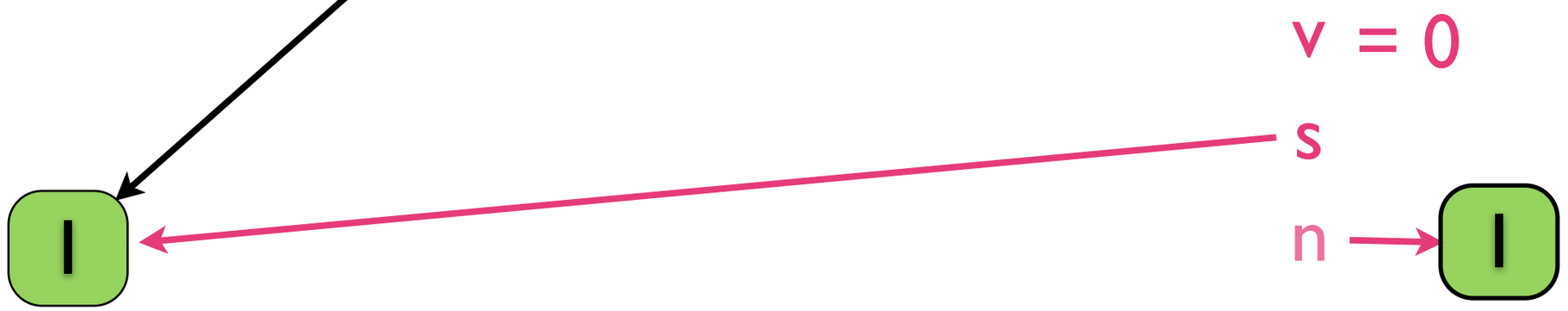
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

concurrent counter (with GC)

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

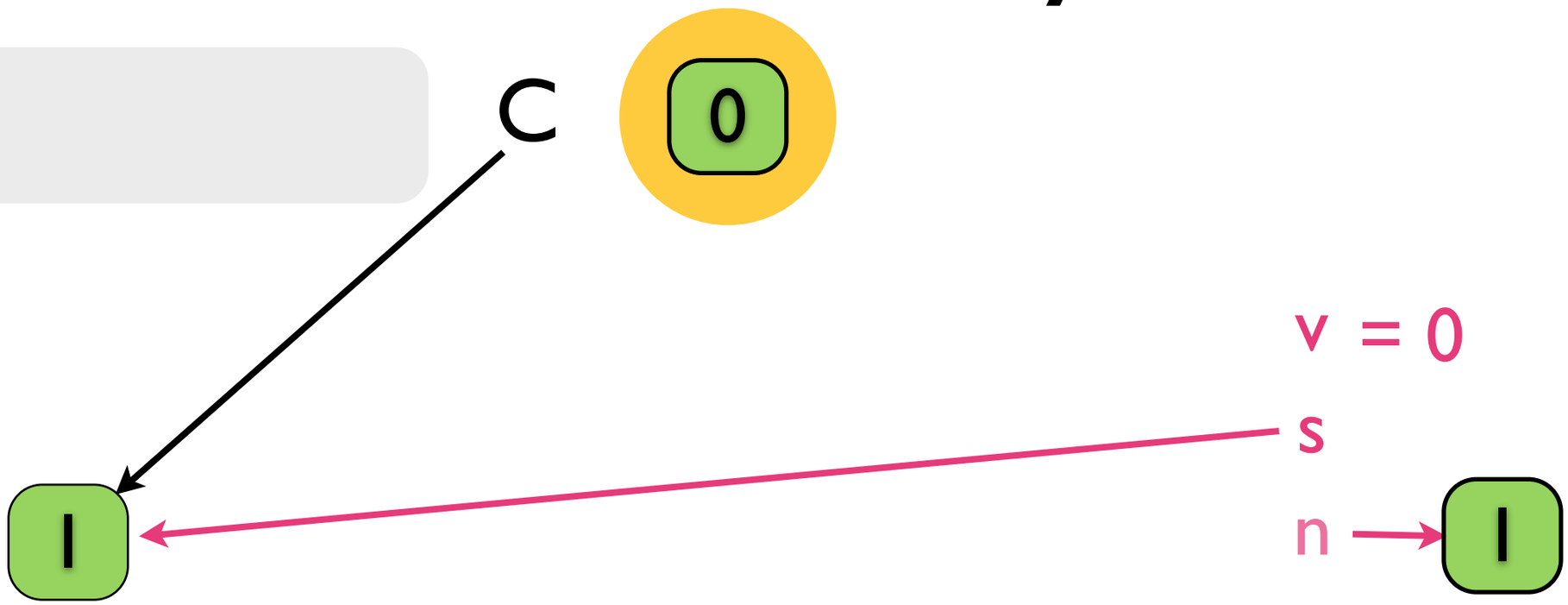
```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

no GC ⇒ memory leaks

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));

 return v;
}
```

naive fix

```
int *C;
```

C

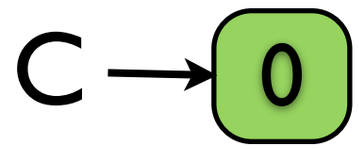

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

problem: memory safety

```
int *C;
```

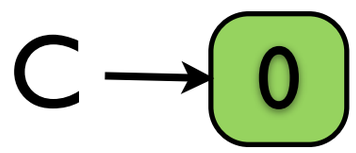



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

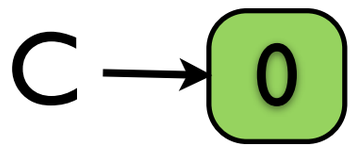



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```



v

s

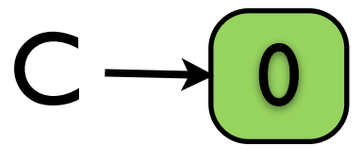
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```



v

s

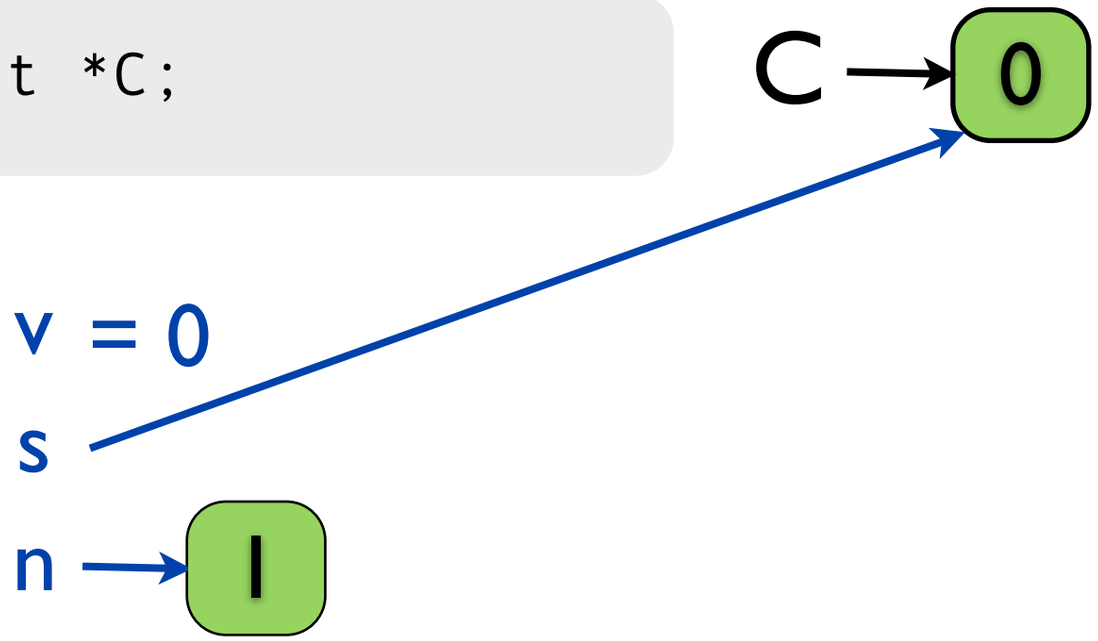
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

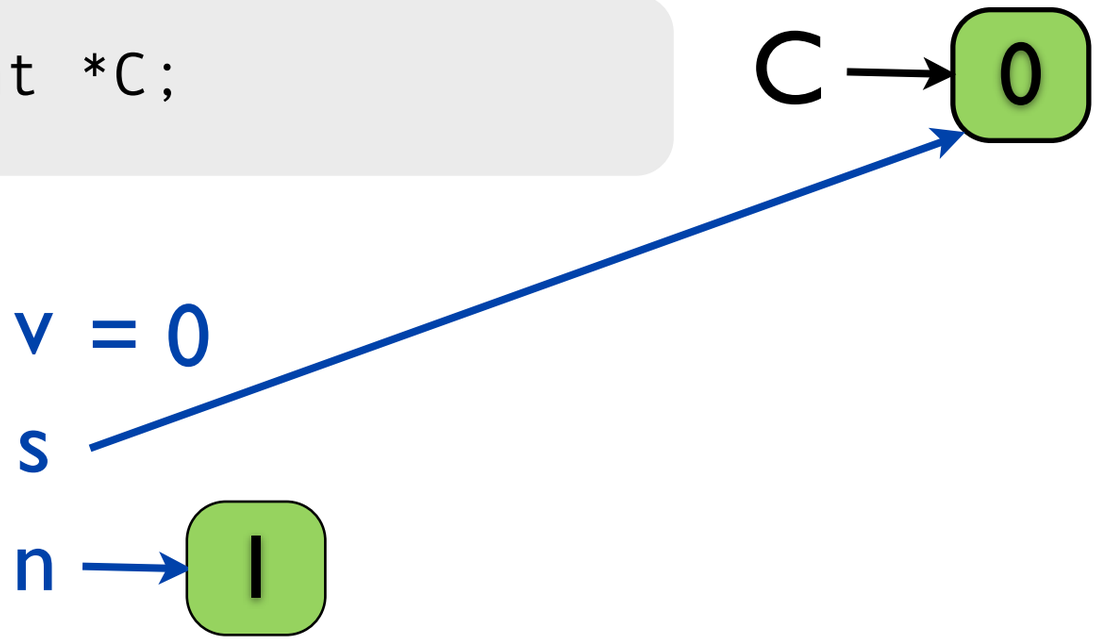



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

v = 0

s

n

1

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

v

s

n

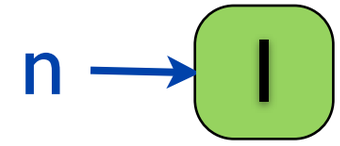
```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

v = 0

s




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

v

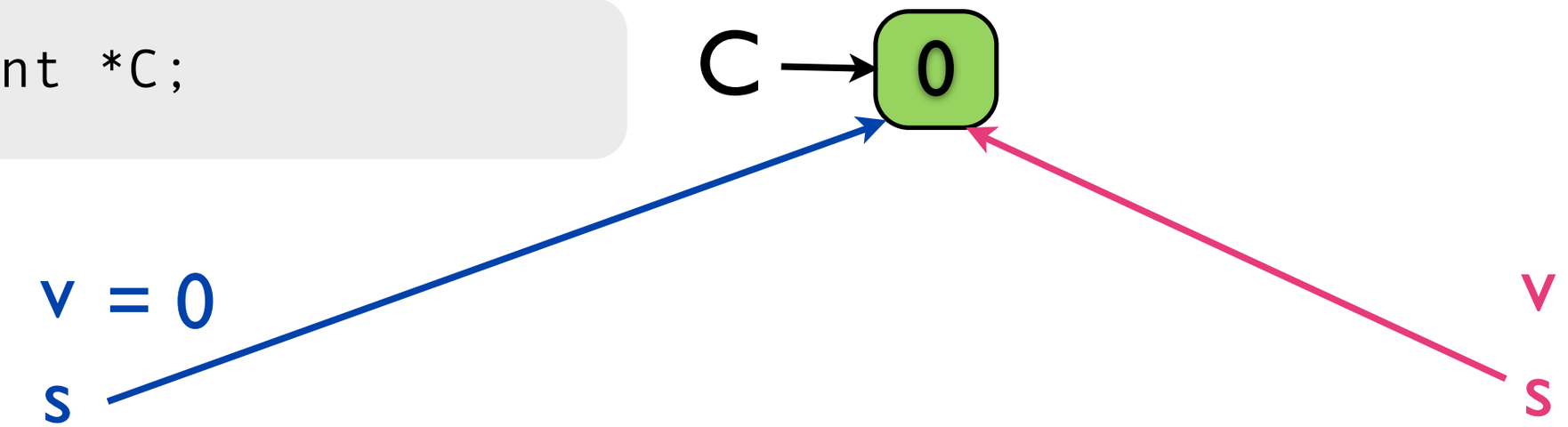
s

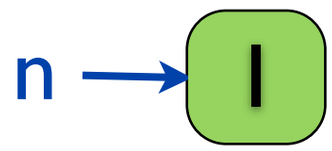
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

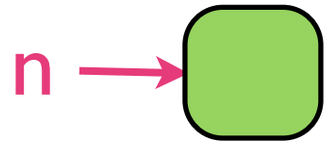
problem: memory safety

```
int *C;
```





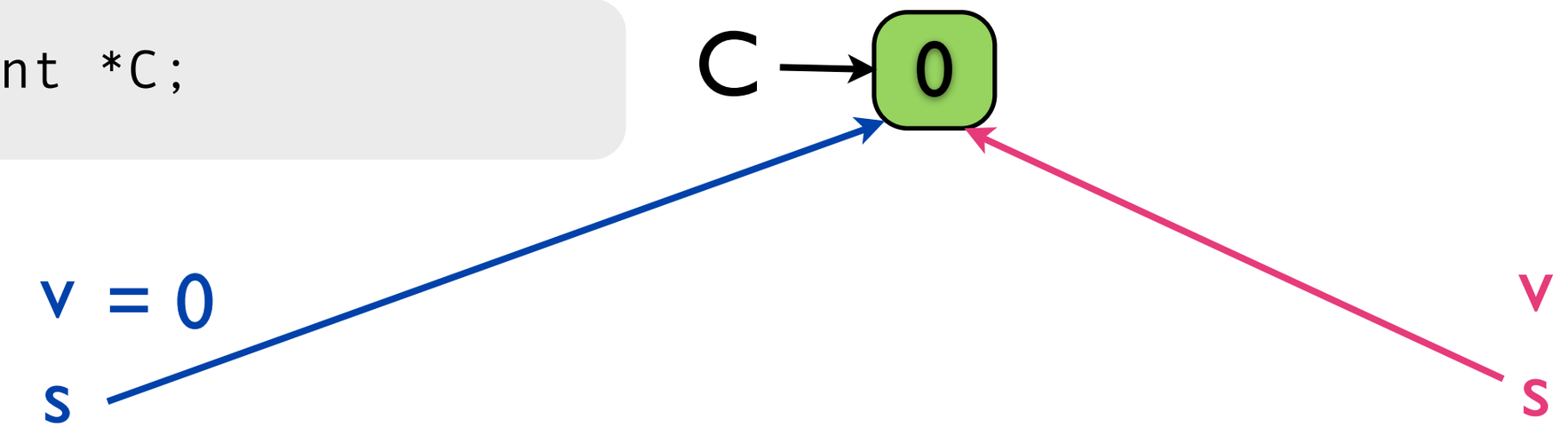

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

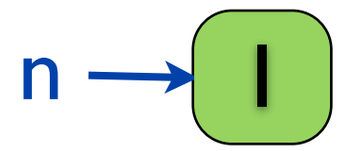



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

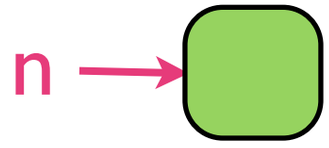
problem: memory safety

```
int *C;
```





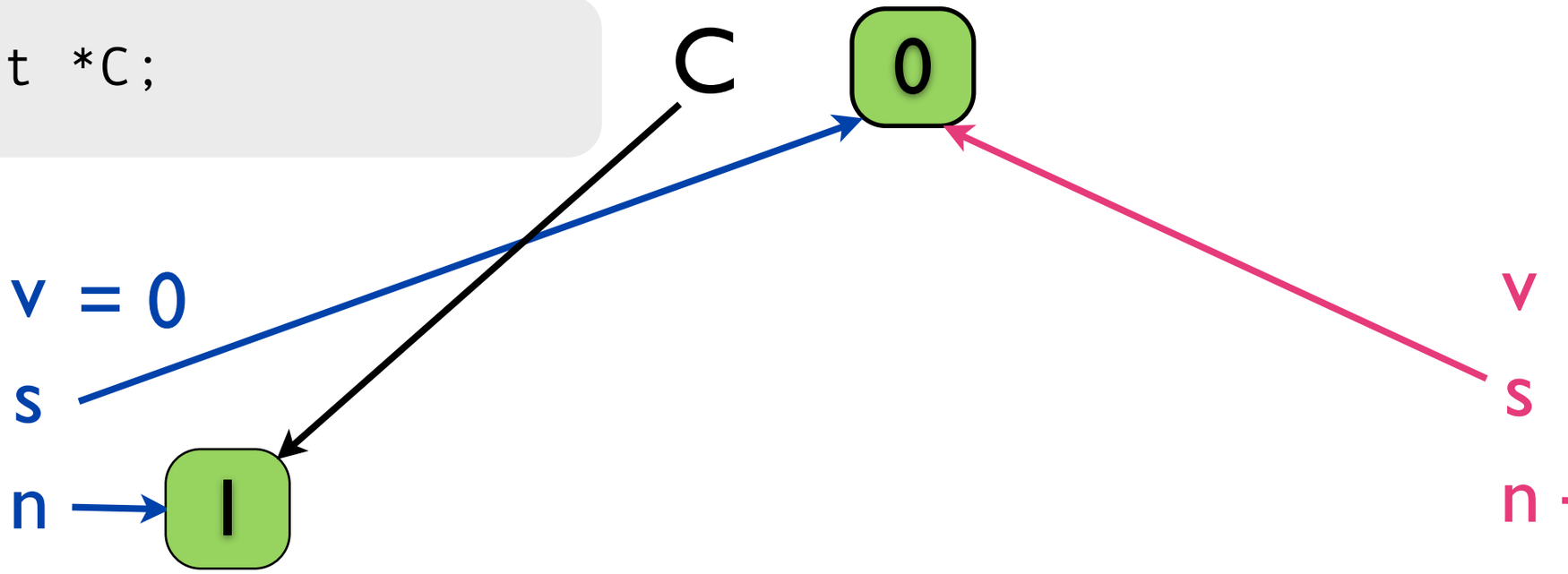

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

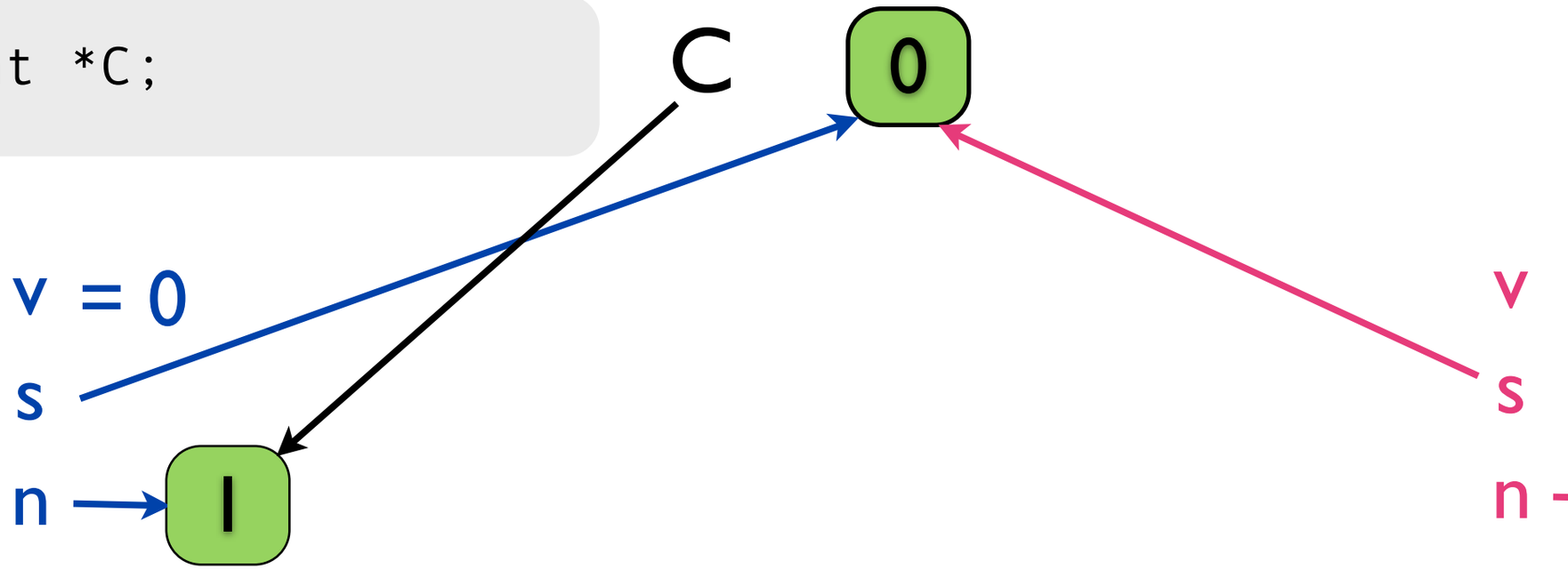



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

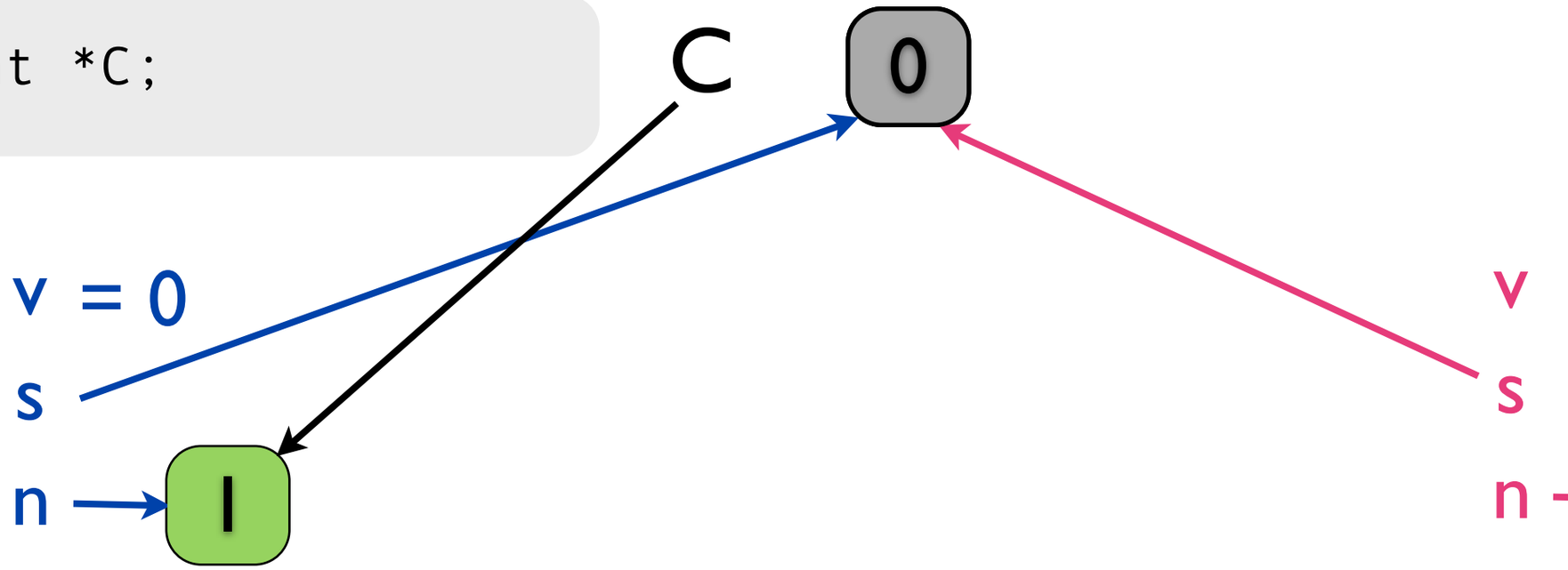



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

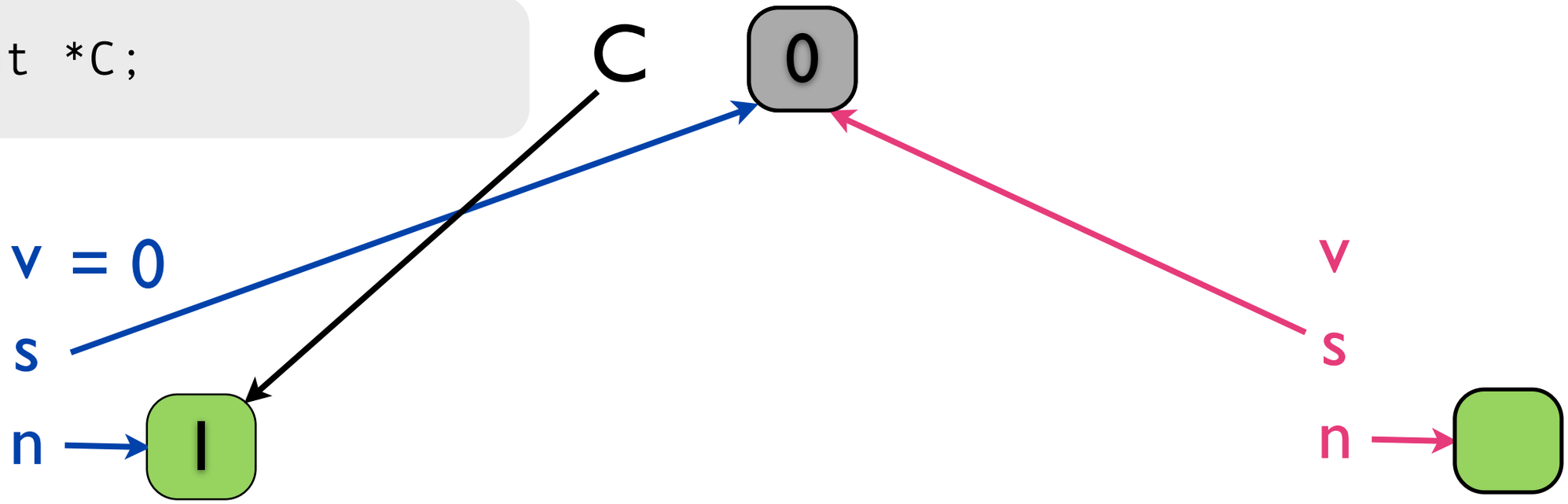



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

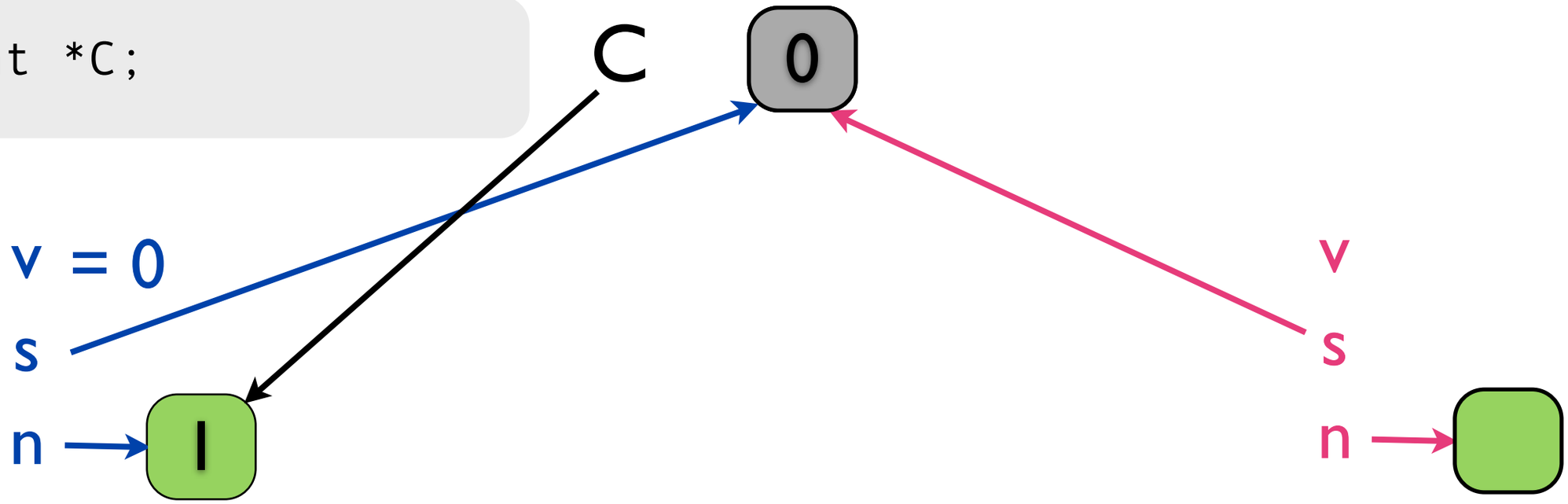



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

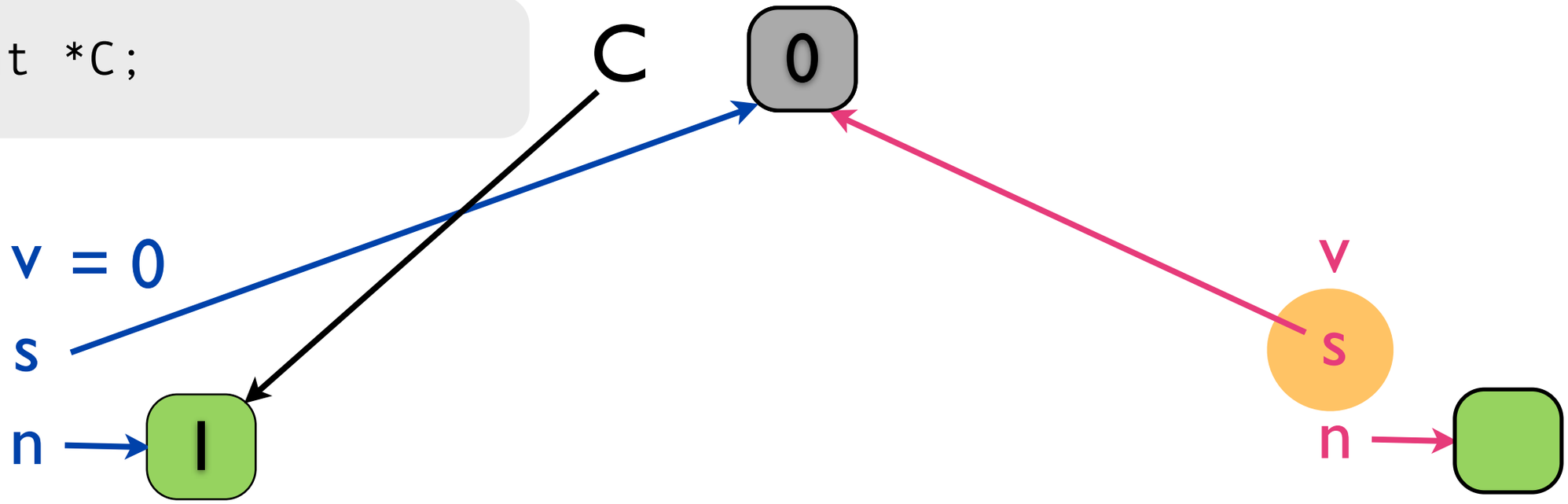



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: memory safety

```
int *C;
```

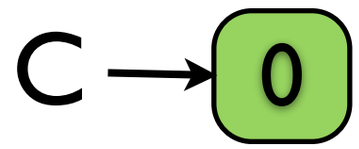



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

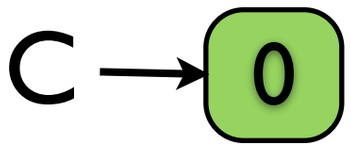



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

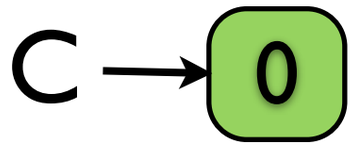



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



v

s

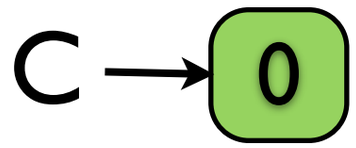
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



v

s

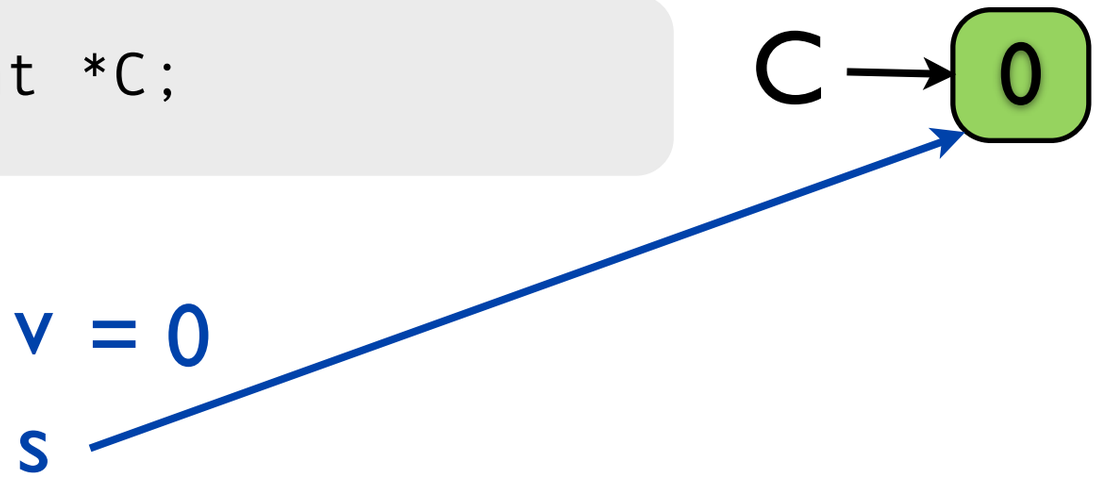
n

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

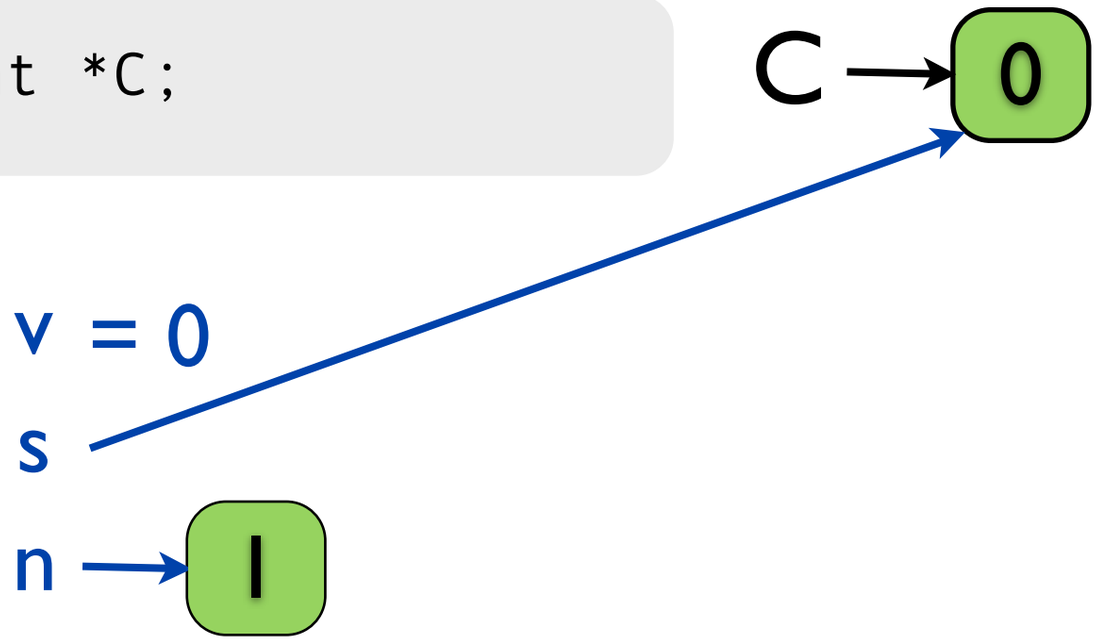



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

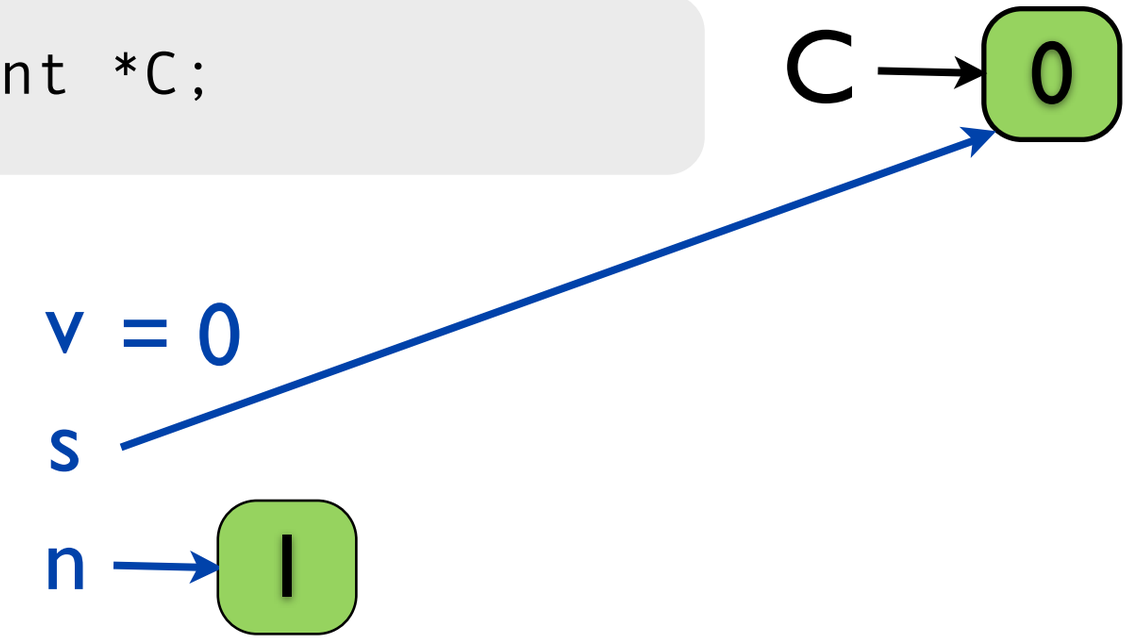
```
int *C;
```


```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```


```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



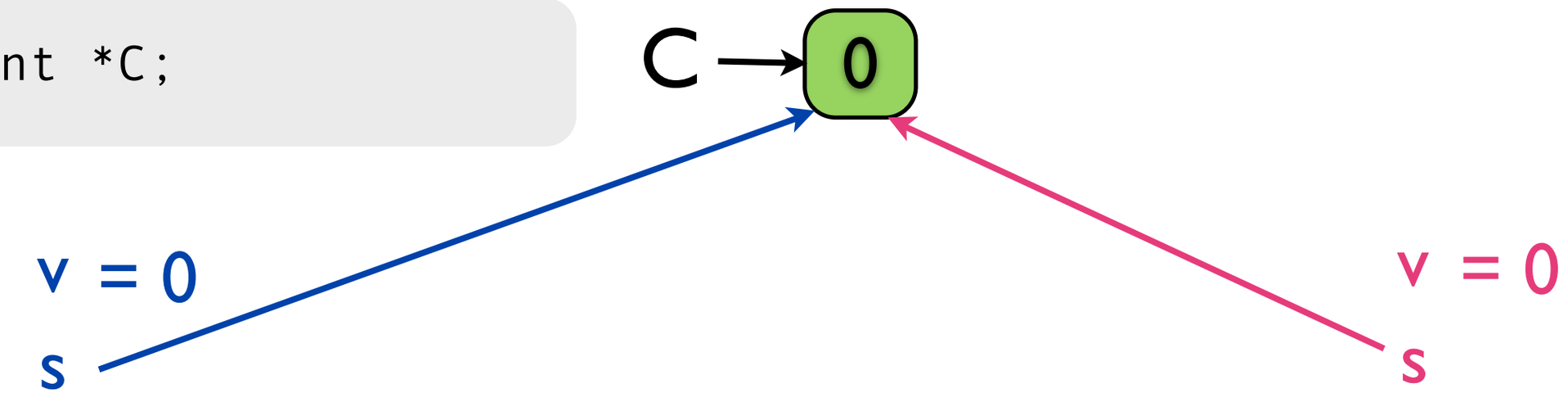

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

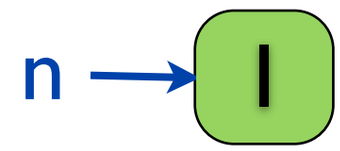


```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

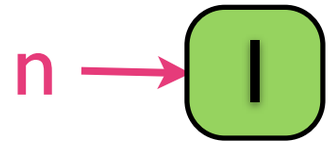
problem: correctness

```
int *C;
```





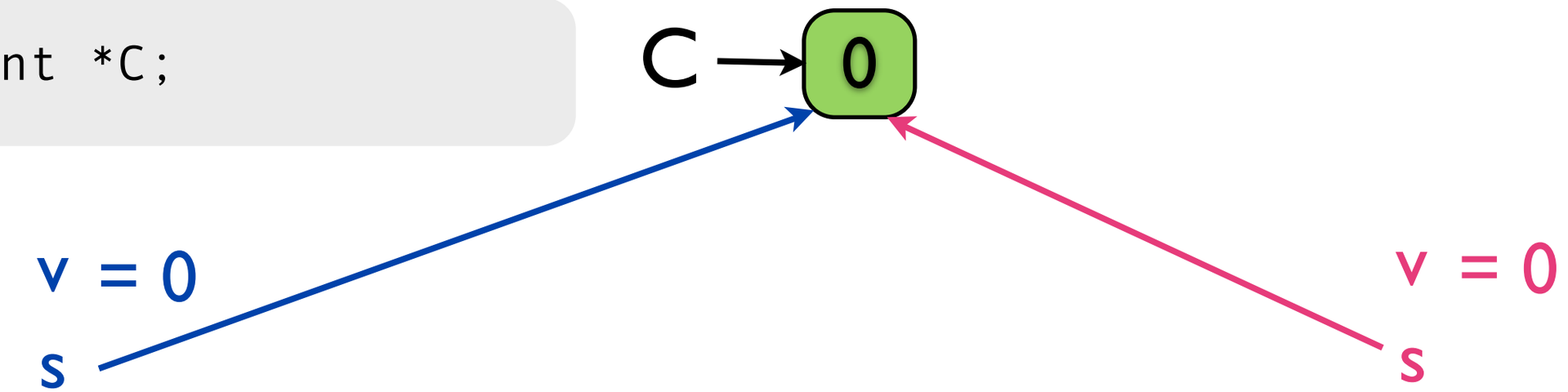

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

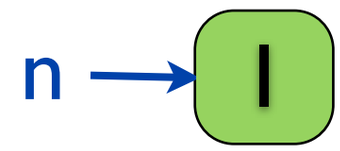



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

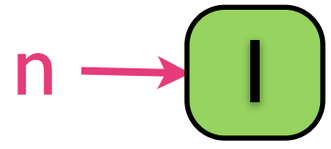
problem: correctness

```
int *C;
```





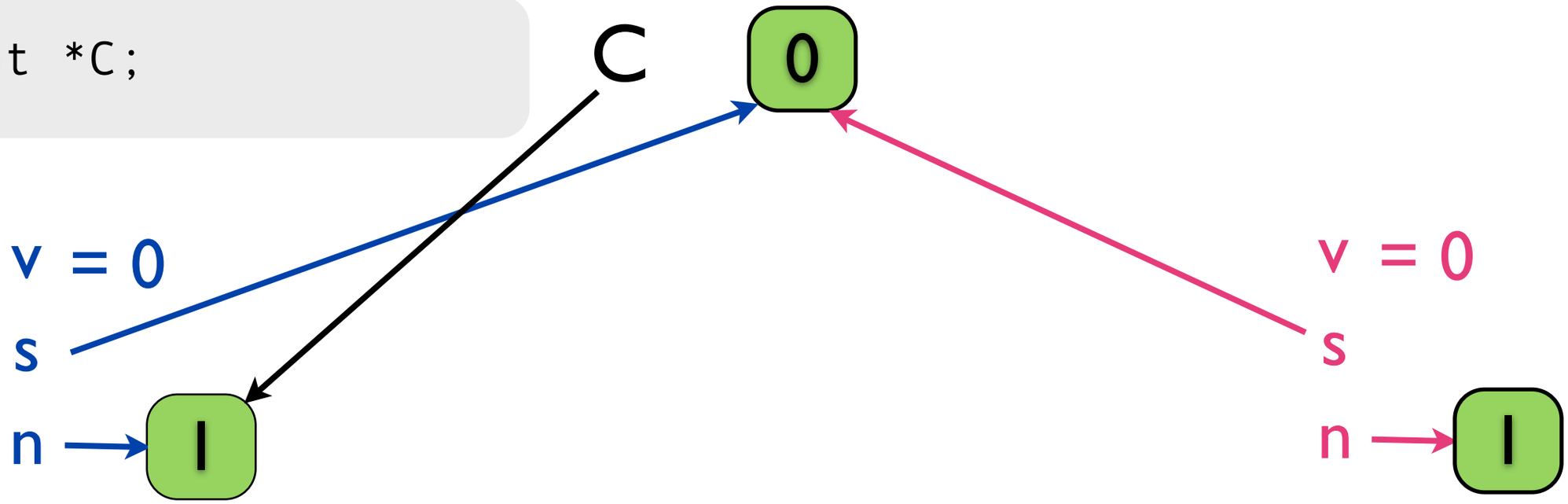

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

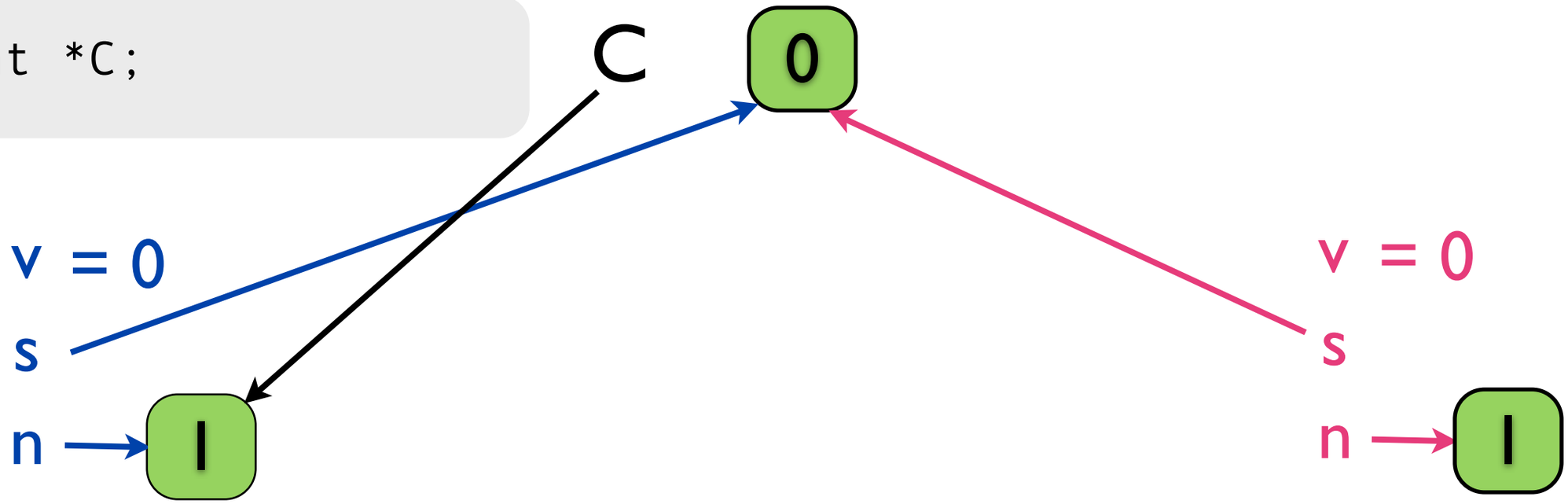



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

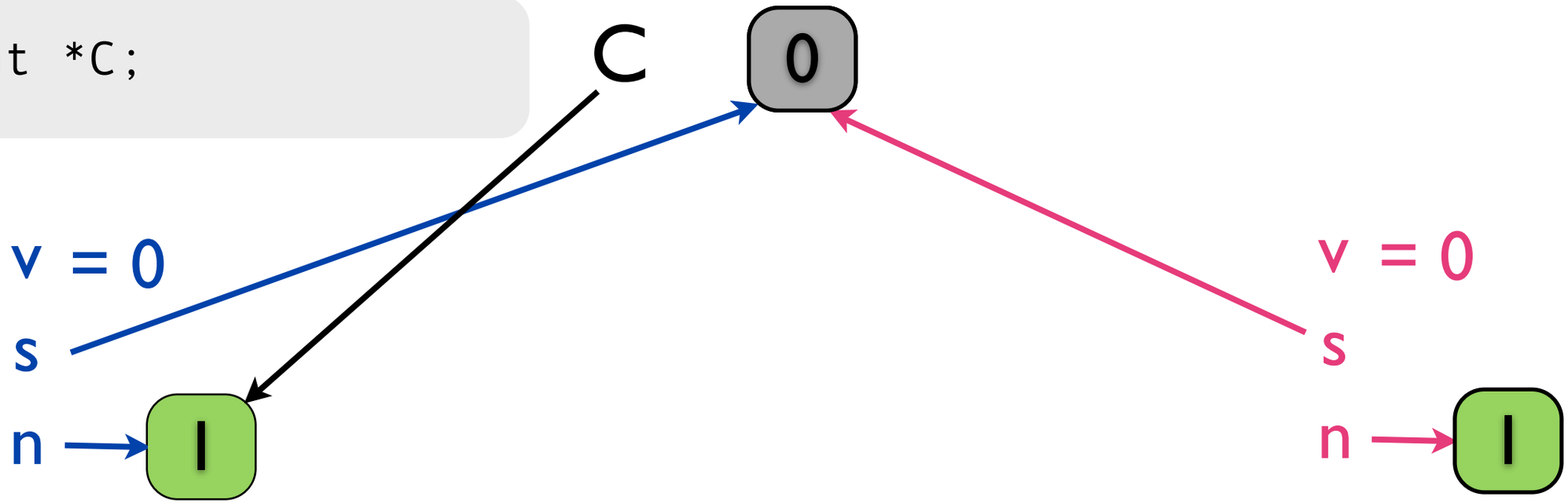



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

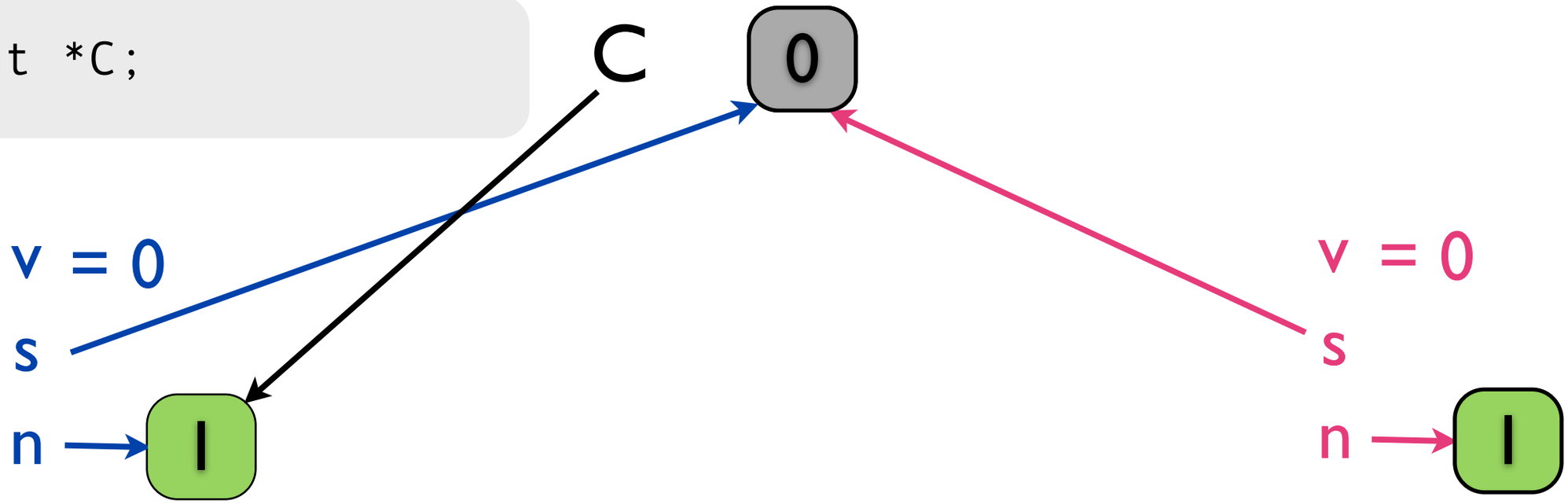



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

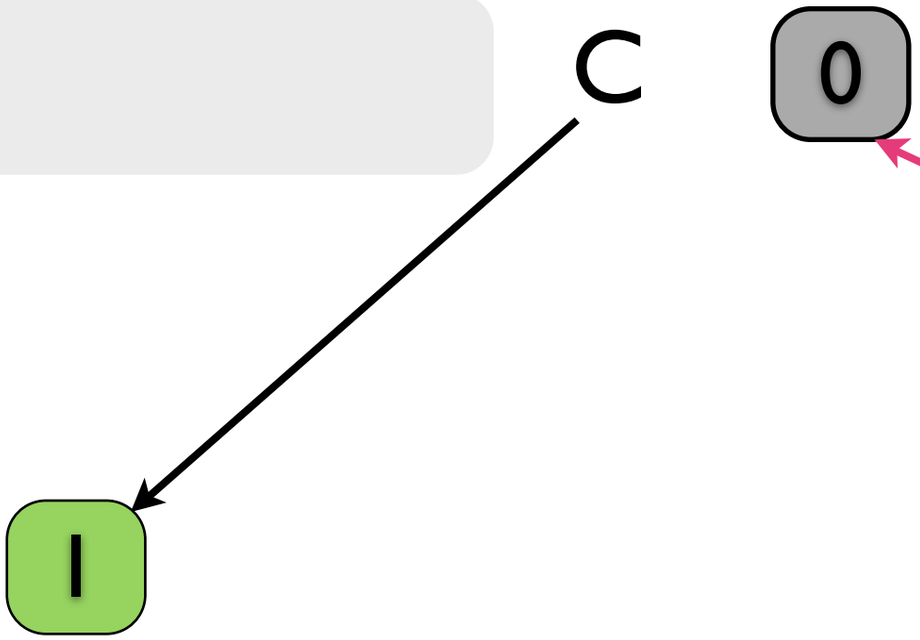



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

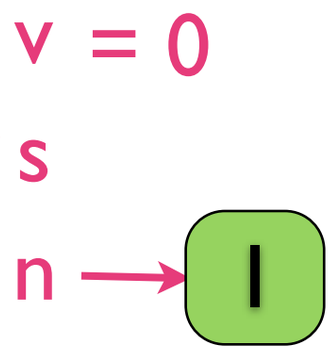
```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



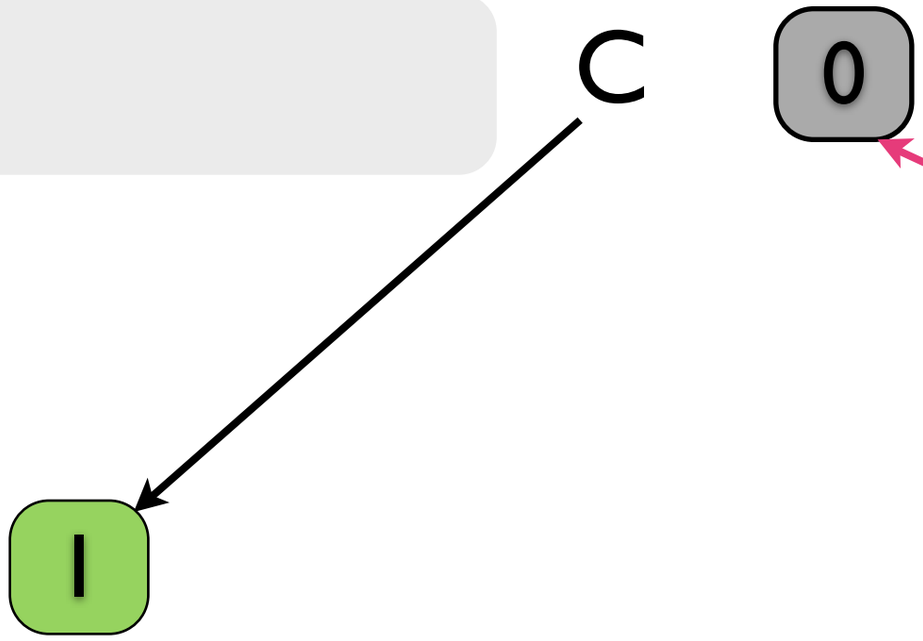

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

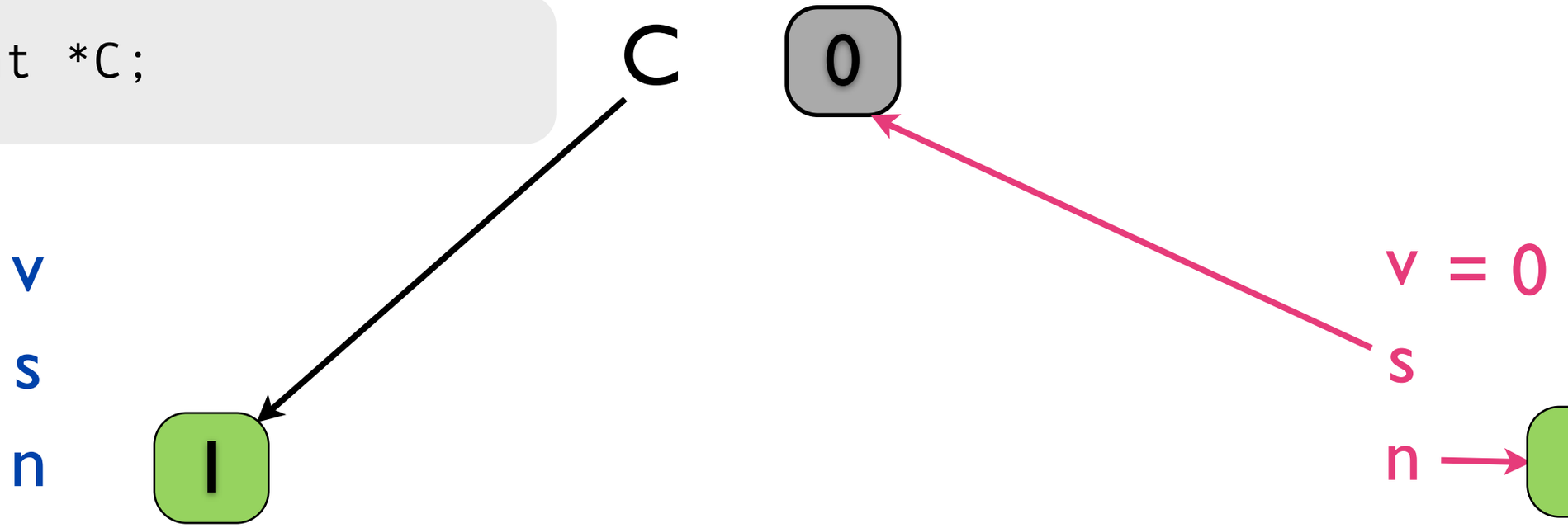



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

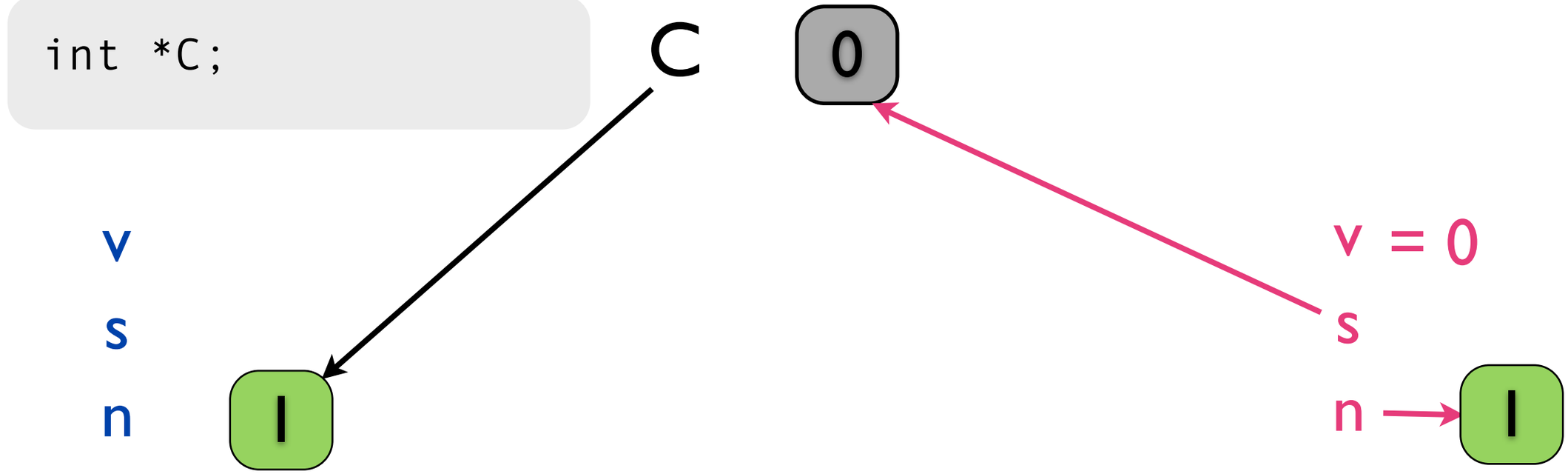



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

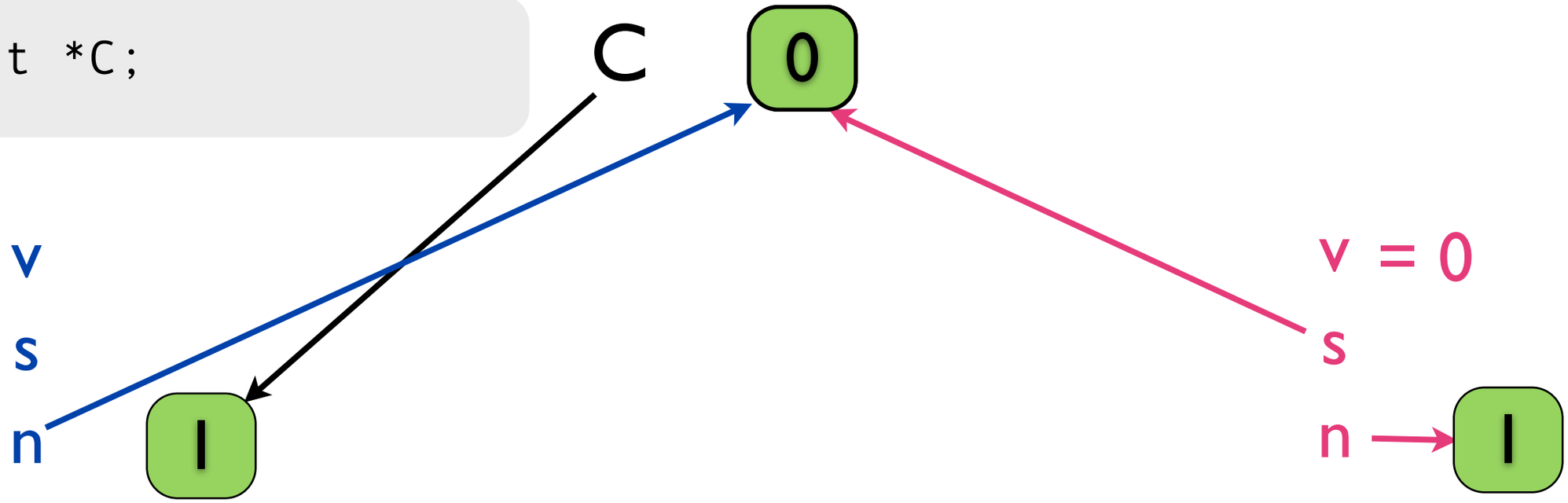



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

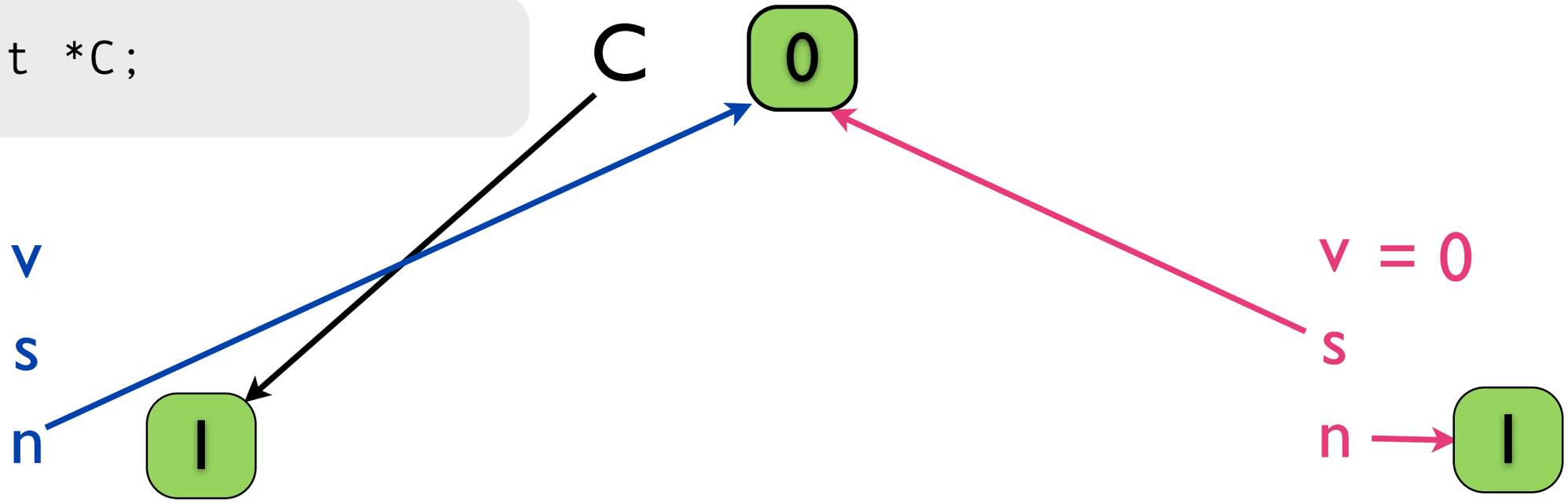



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

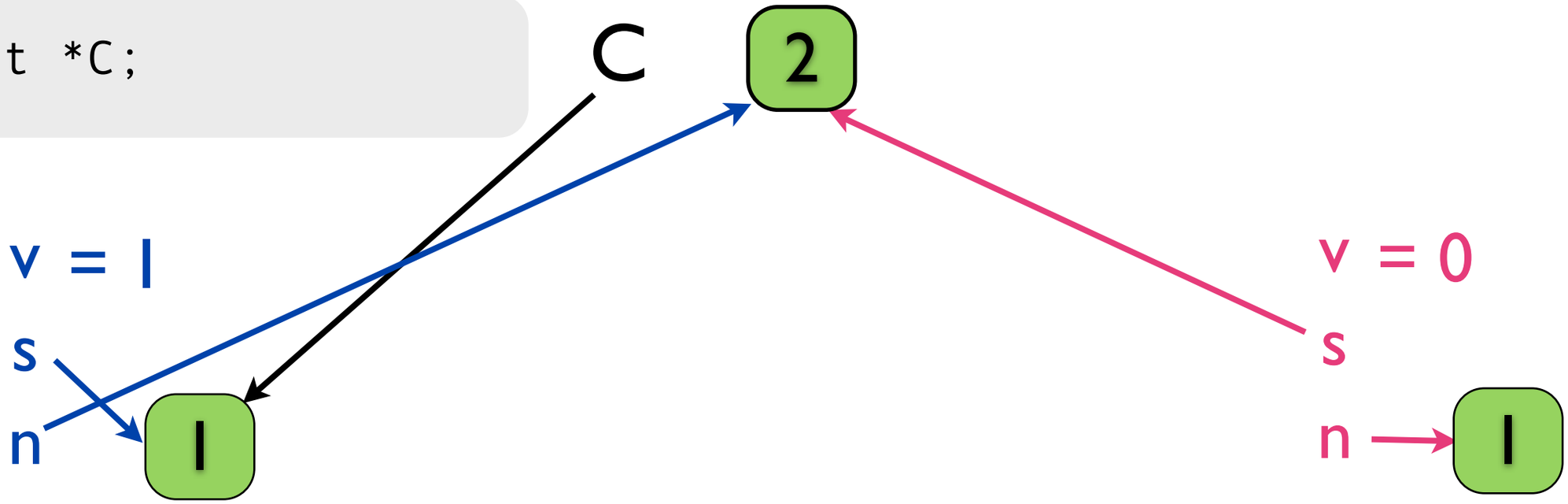



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

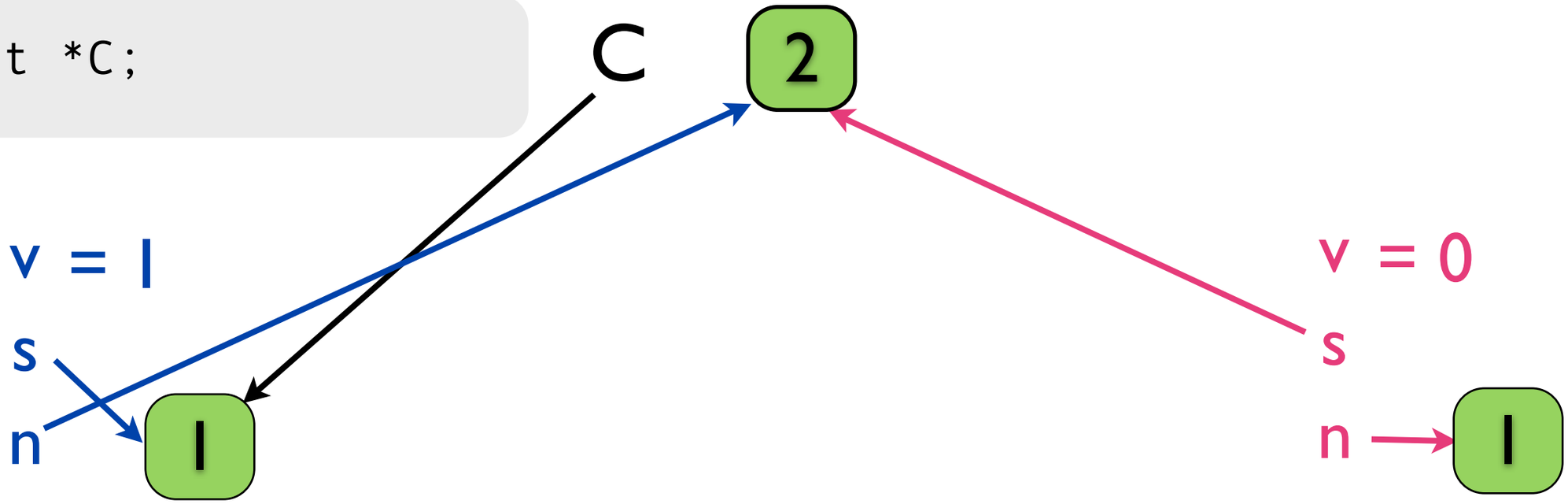



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

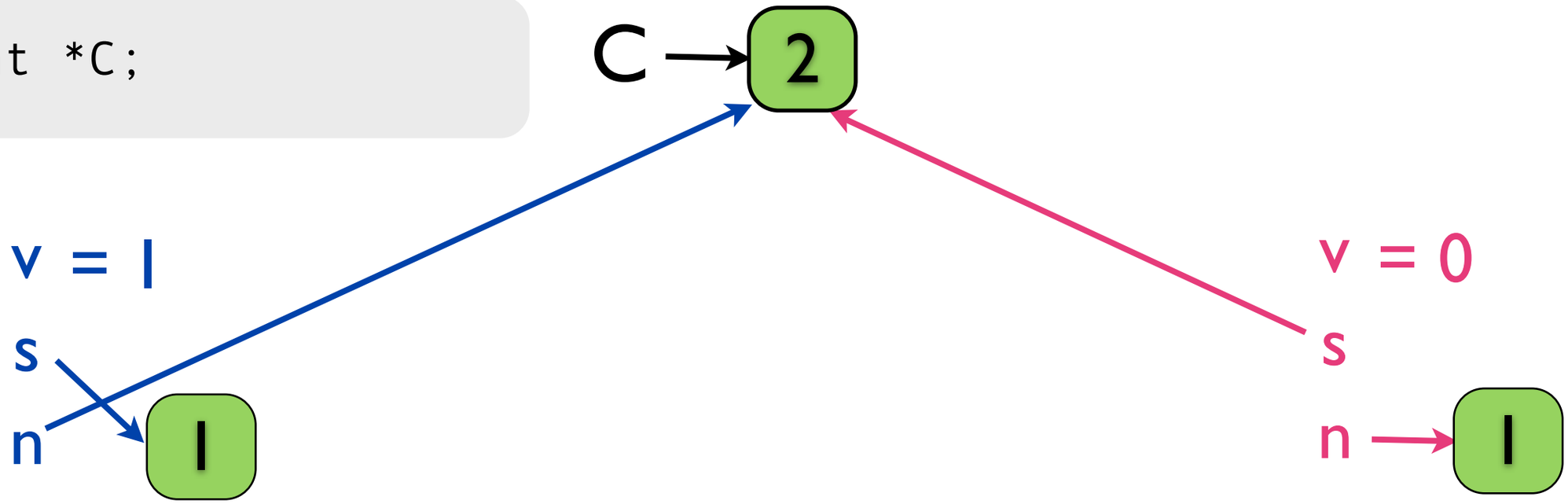



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

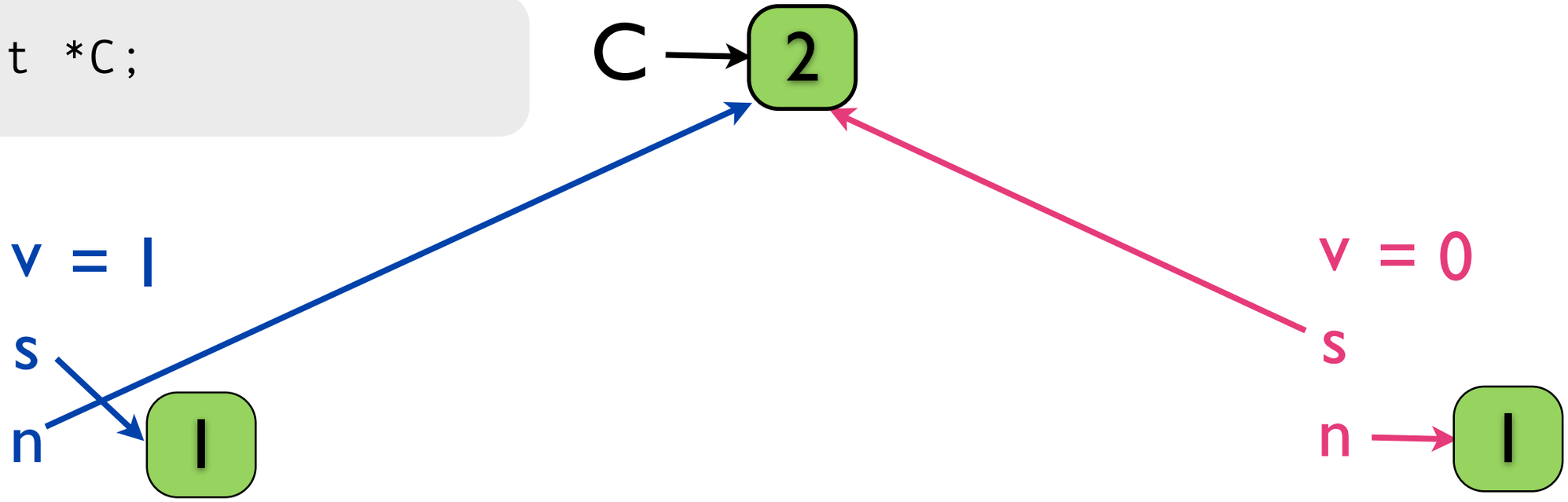



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```

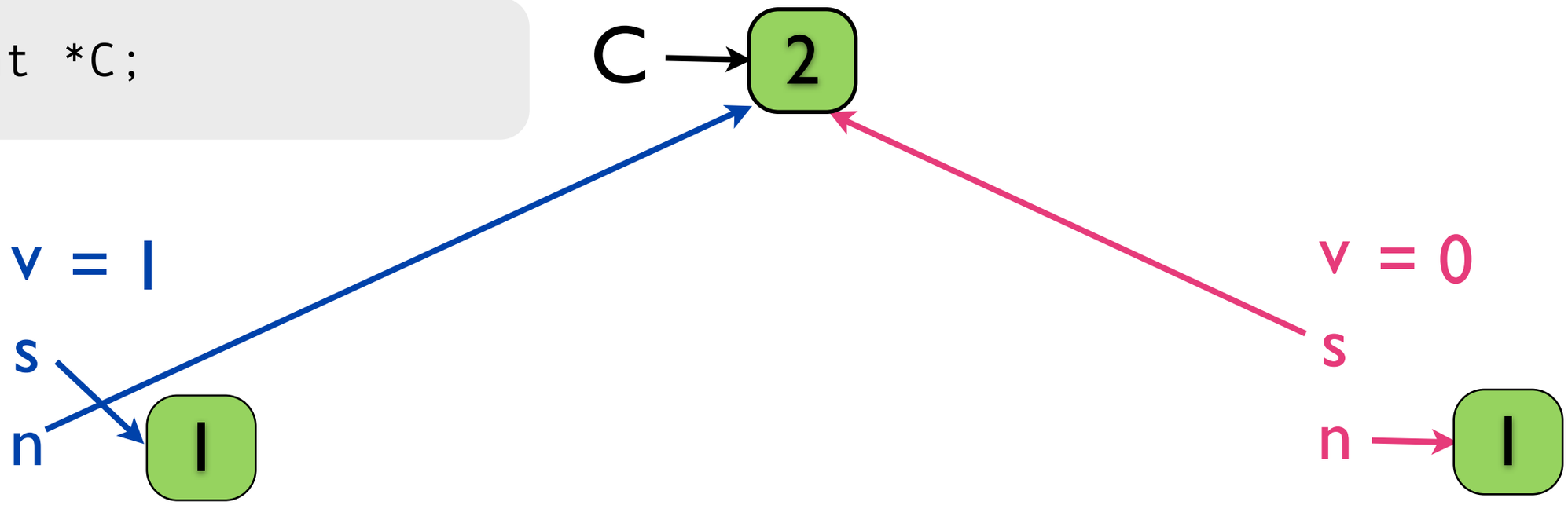



```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



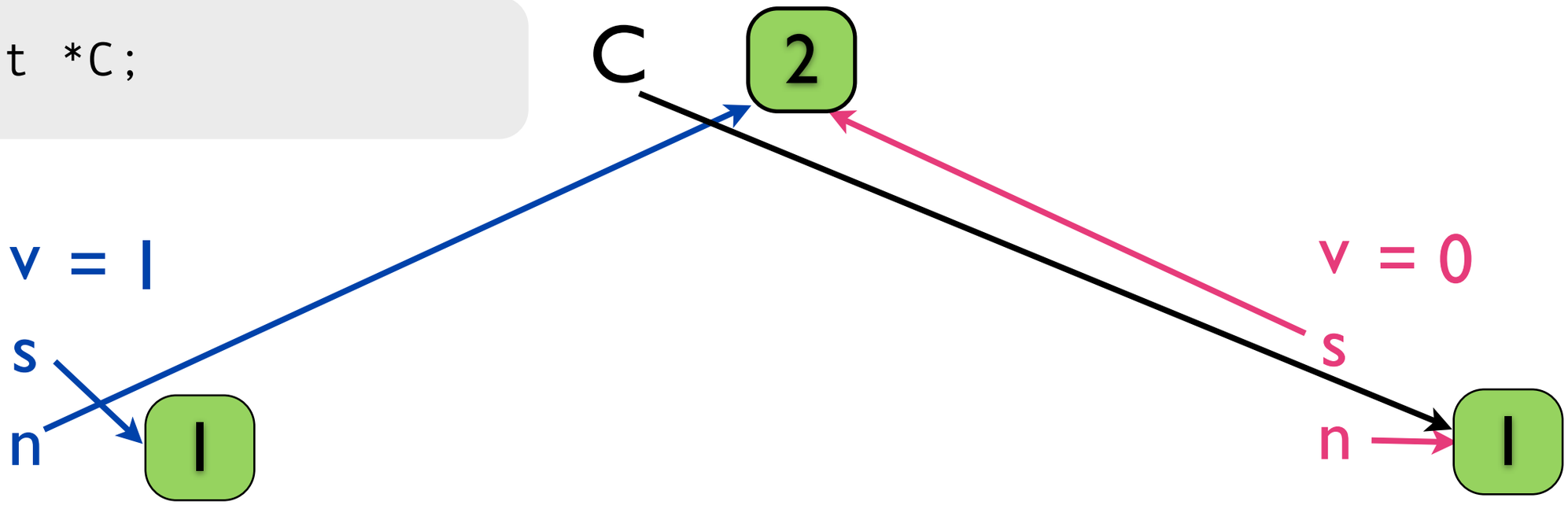

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



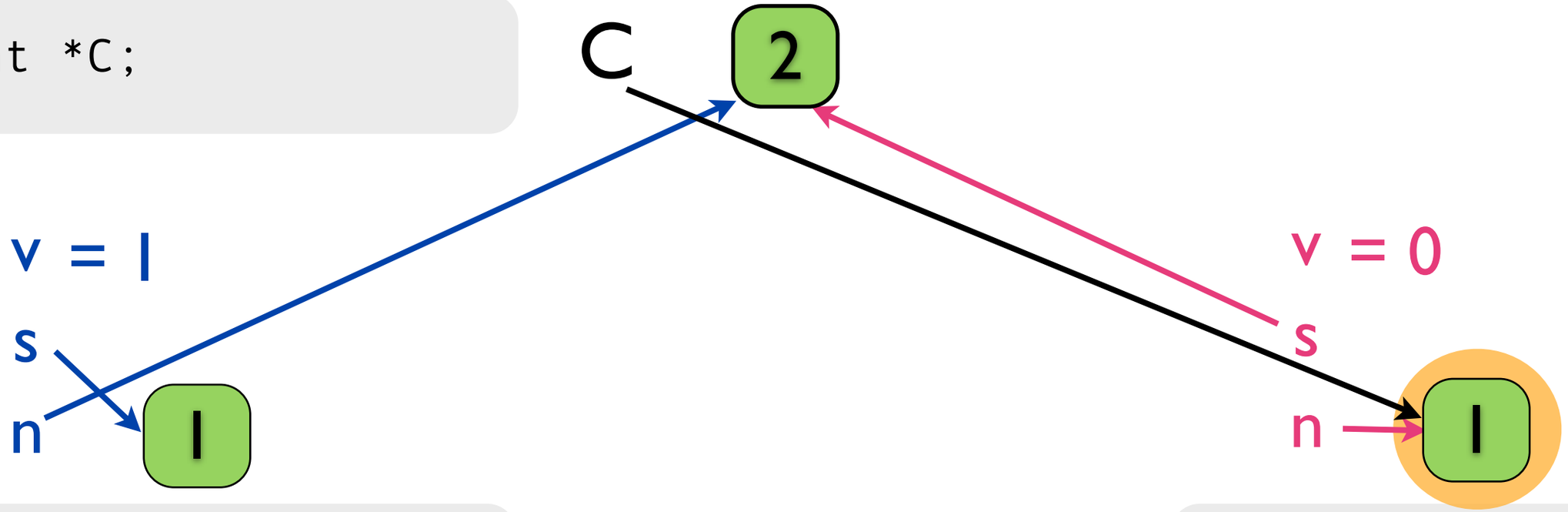

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```



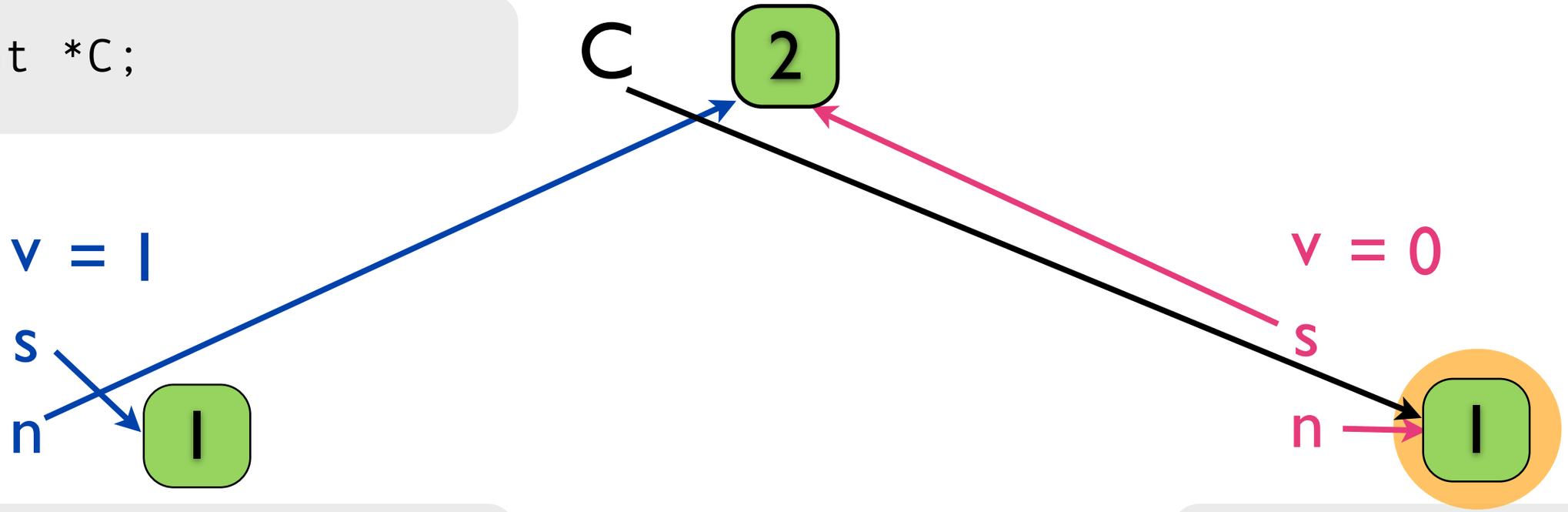

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
CAS(&C,s,n)≔
 <if(C==s)
    C=n;
    return true
  else
    return false>
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

problem: correctness

```
int *C;
```




```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

ABA anomaly

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s)
 return v;
}
```

reclamation algorithms (paradigms)

- **Hazard Pointers** [Michael'02]
- **RCU** [McKenney+'98]
- **Epoch** [Fraser+'03]

main insight

- **Hazard Pointers** [Michael'02]
- **RCU** [McKenney+'98]
- **Epoch** [Fraser+'03]

- pattern of temporal synchronization invariant
  - grace periods
- same logical idiom for invariant preservation

main contributions

- exposing algorithmic core
  - pattern for spatial-temporal synchronization
  - unified view based on grace periods

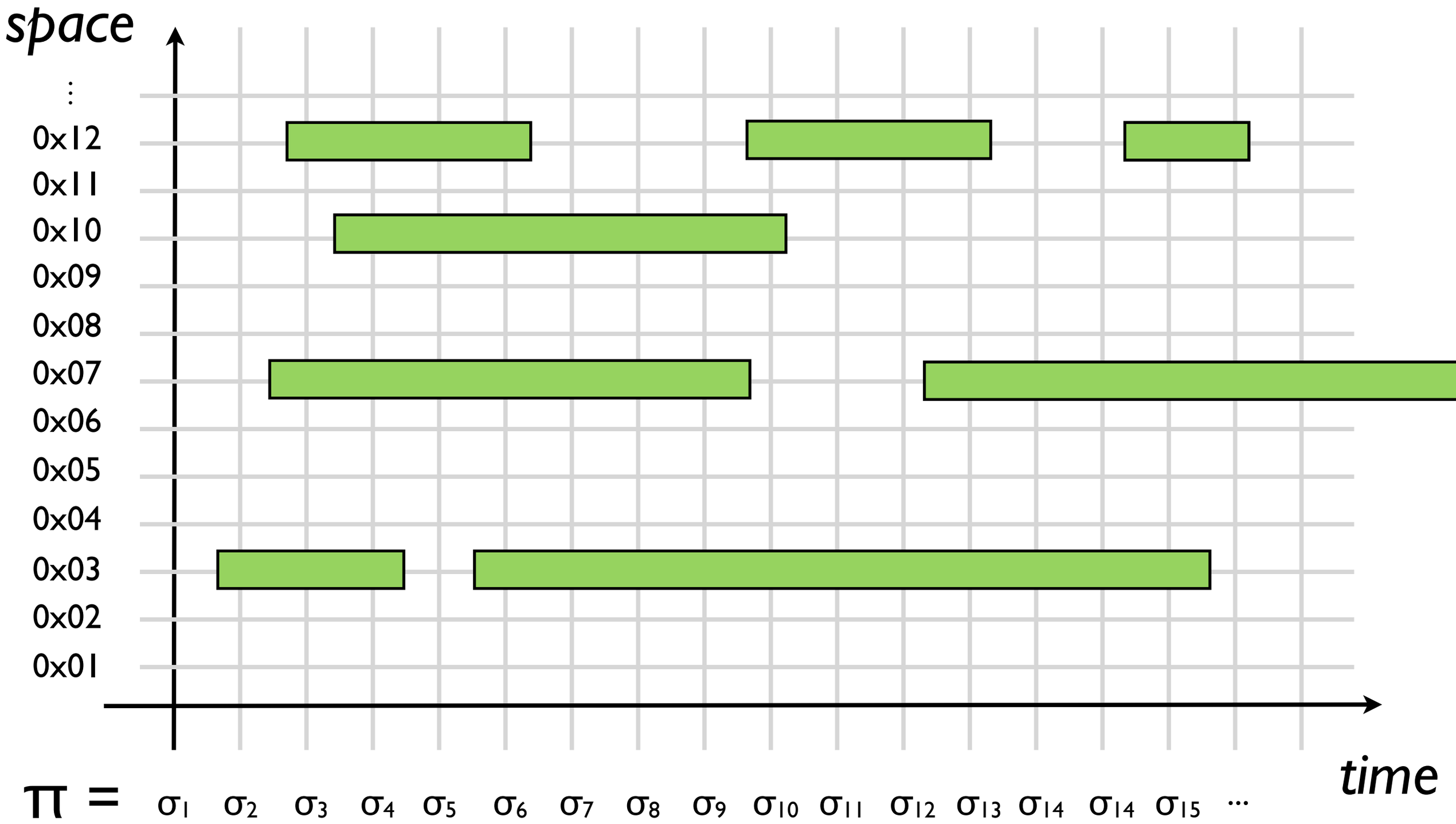
- pattern-based modular verification
  - past temporal separation logic (*since*)
  - uniform proofs of realistic algorithms

main contributions

- exposing algorithmic core
  - pattern for spatial-temporal synchronization
    - unified view based on grace periods

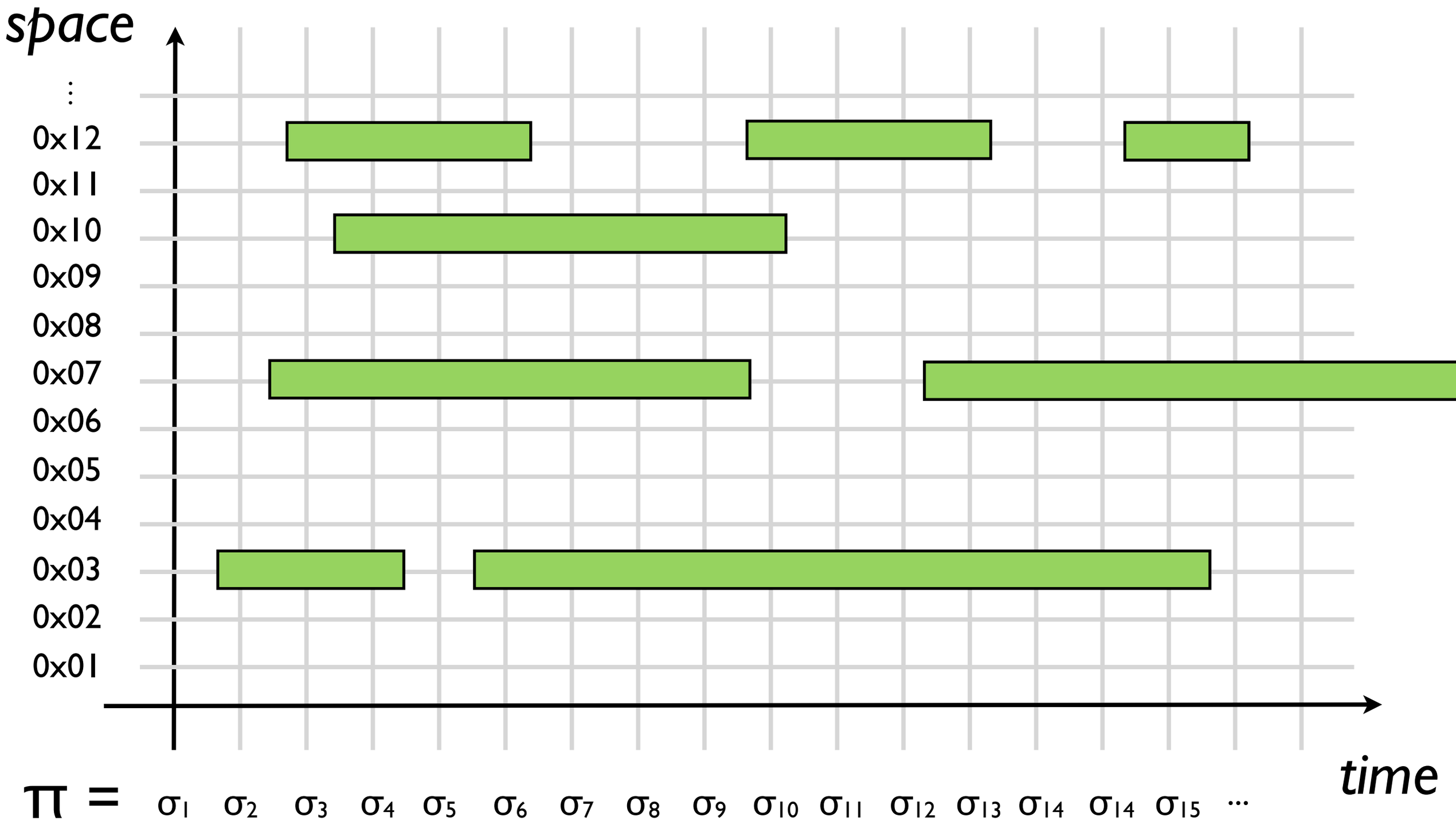
- pattern-based modular verification
  - past temporal separation logic (*since*)
  - uniform proofs of realistic algorithms

dual views: space / time



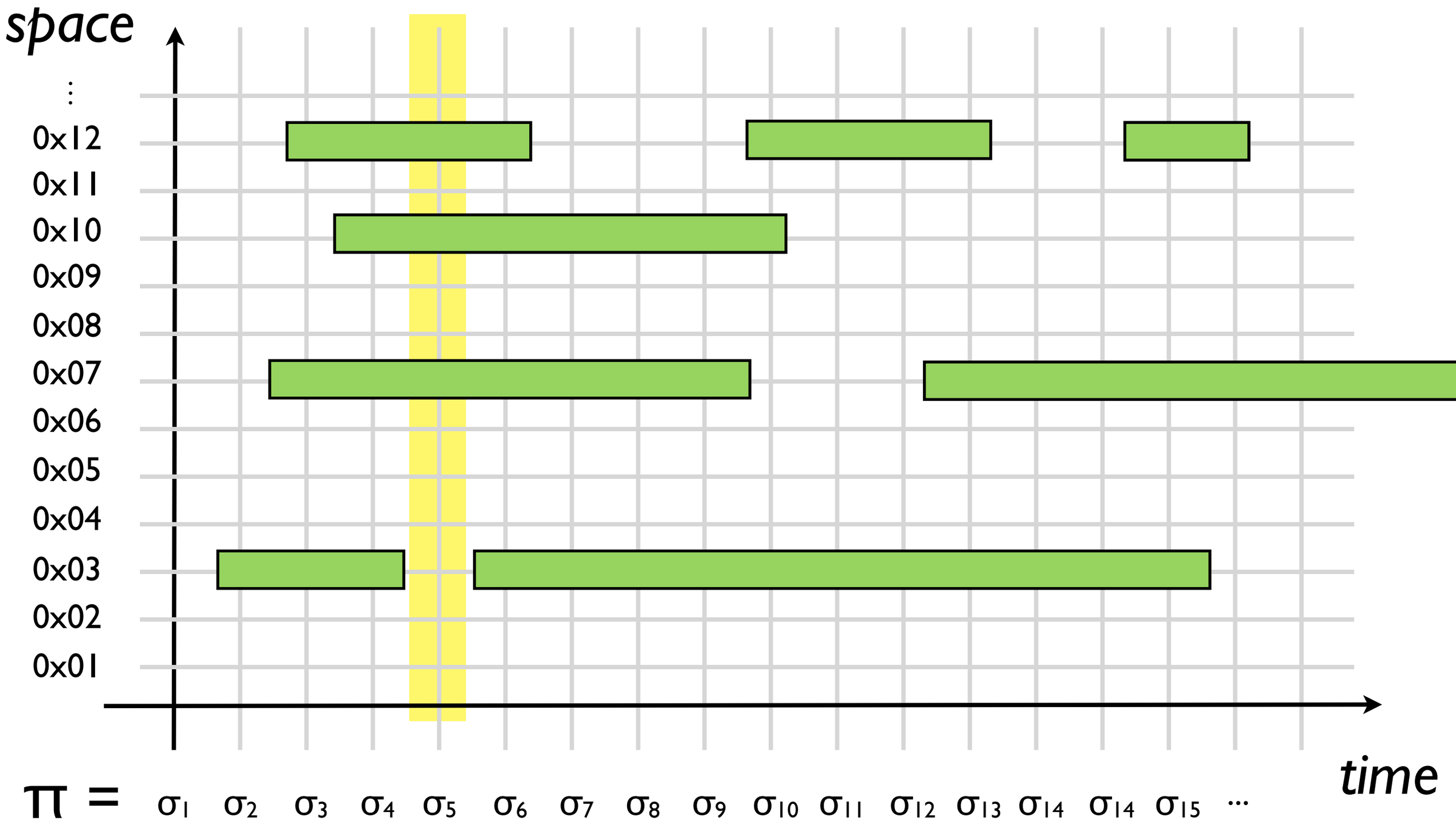
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



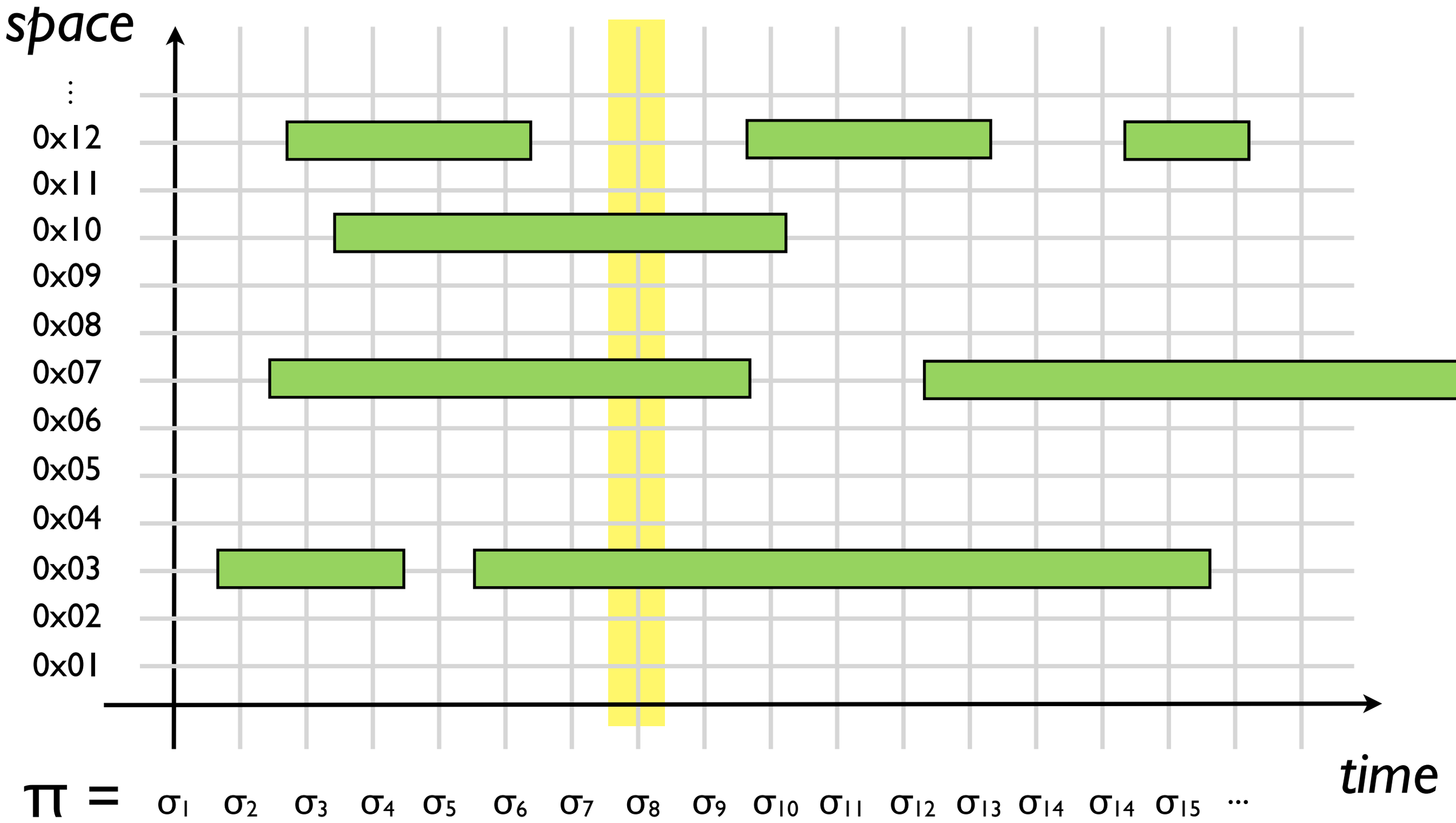
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



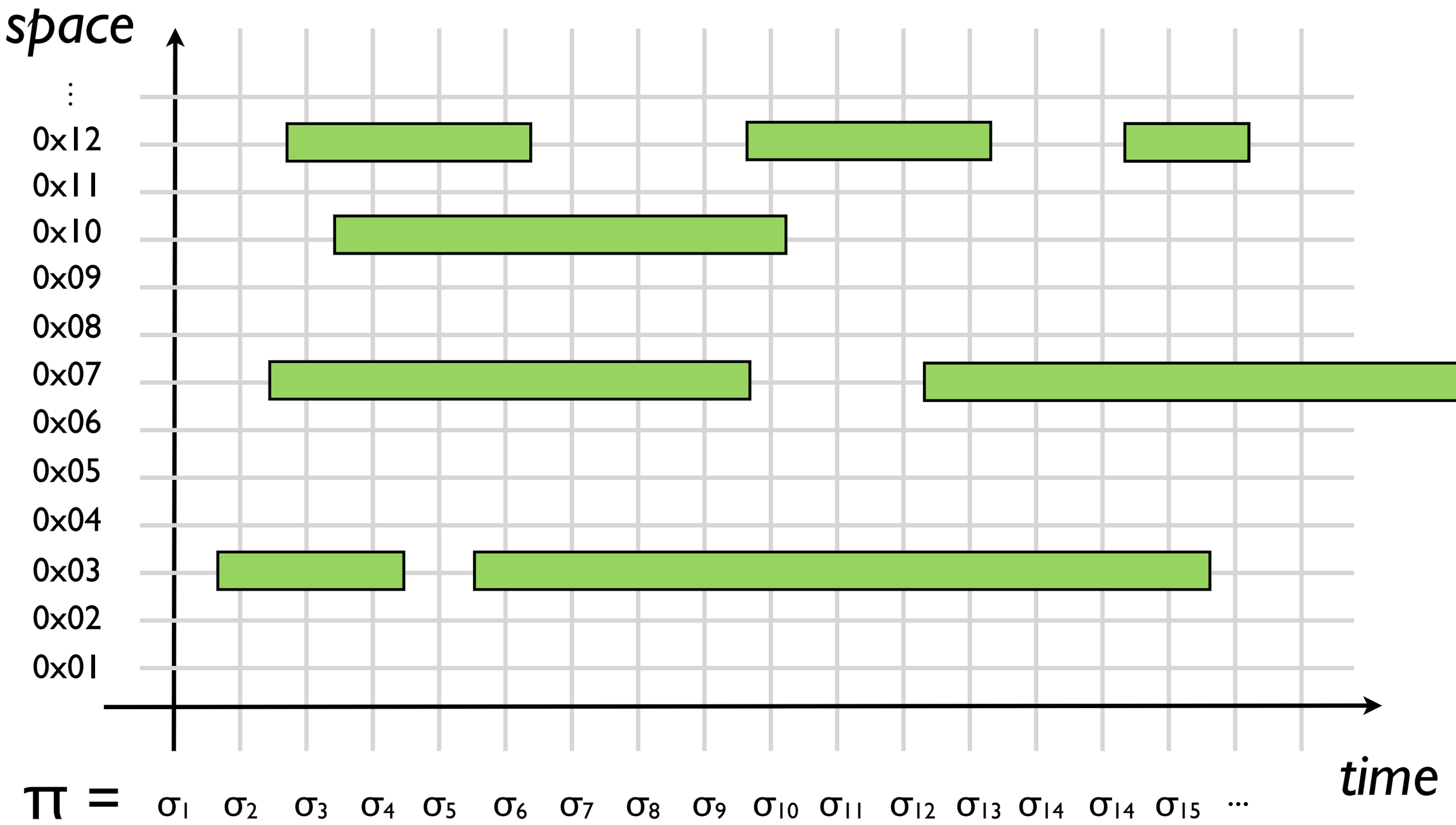
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



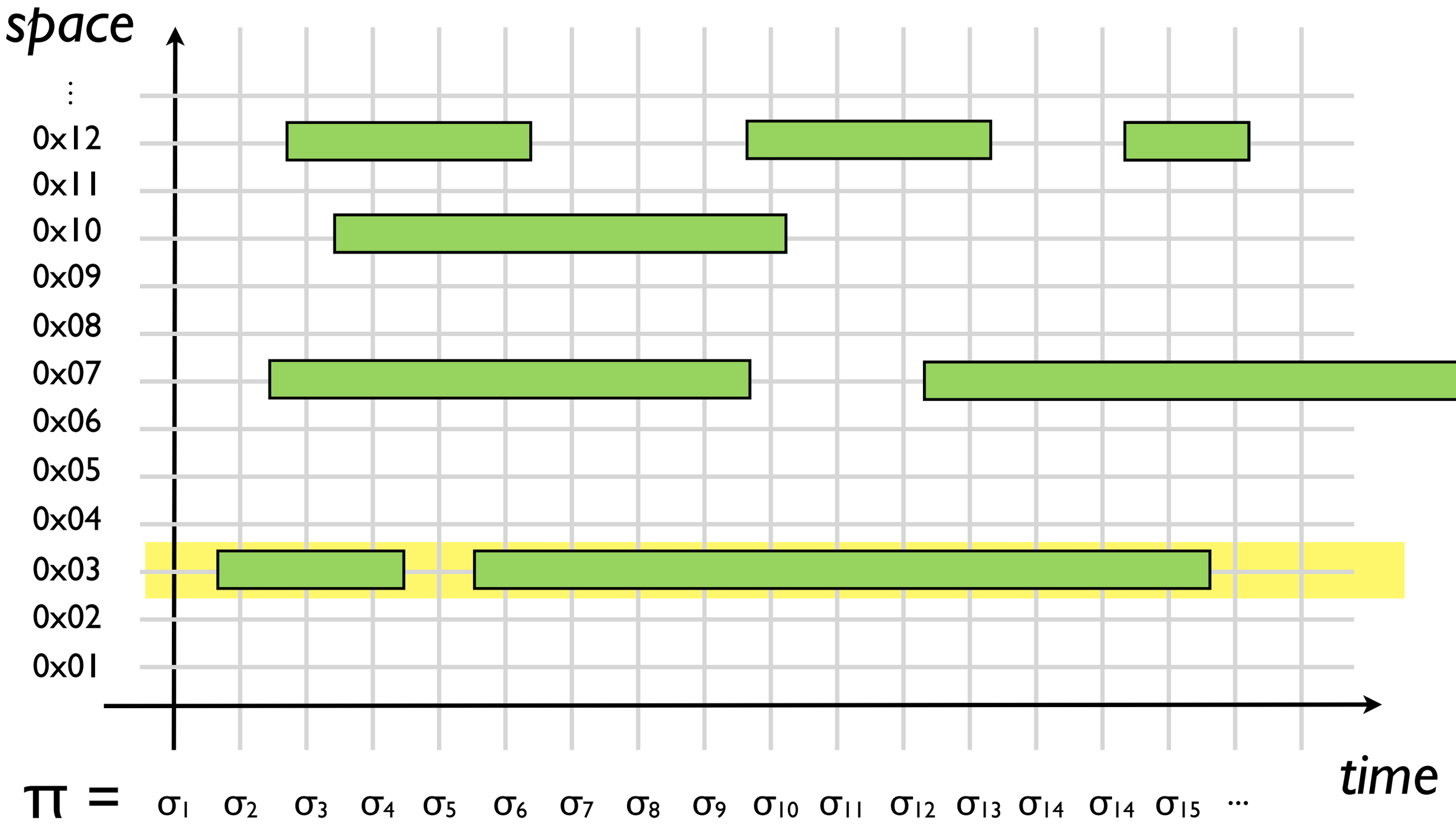
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



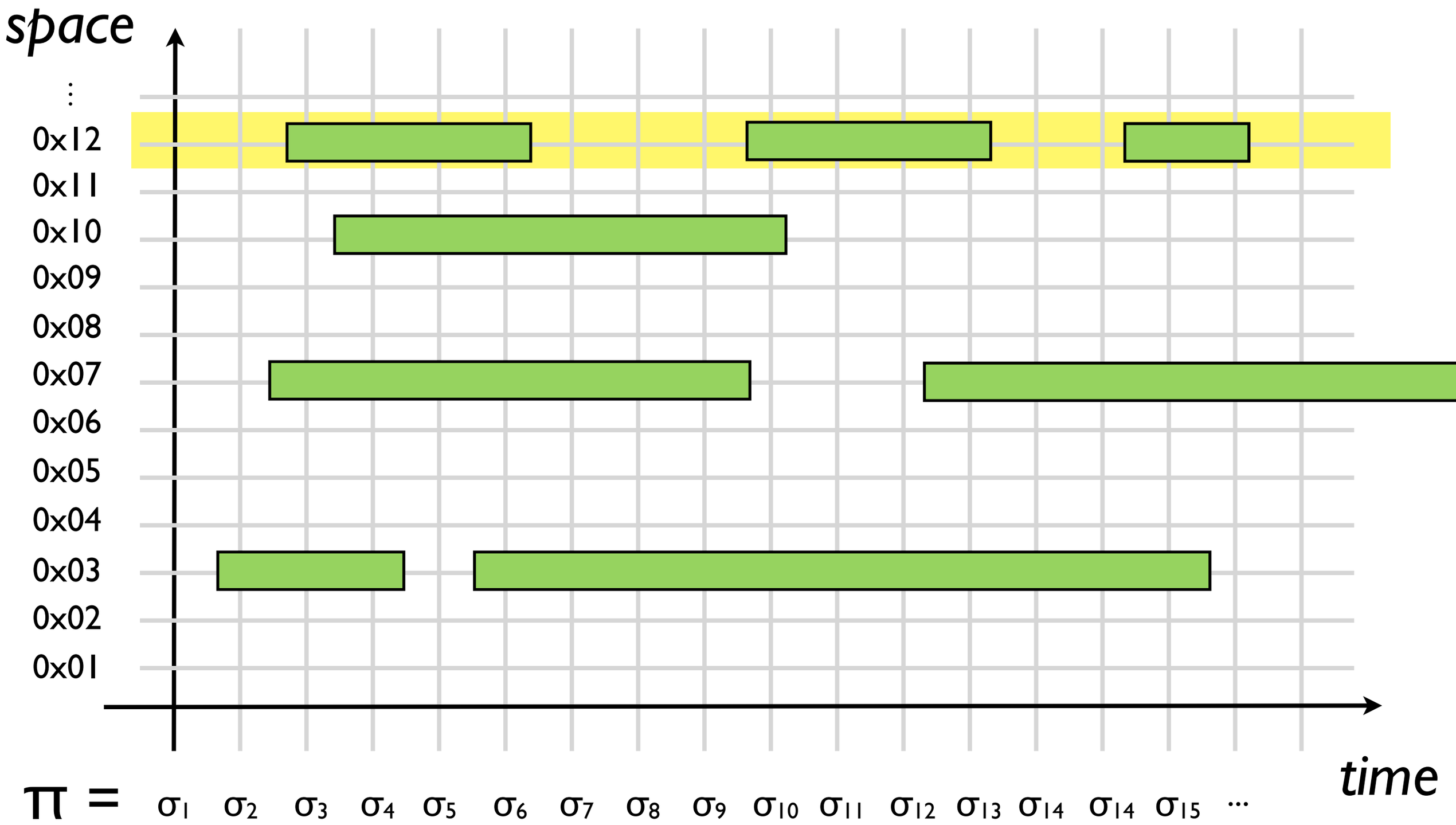
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



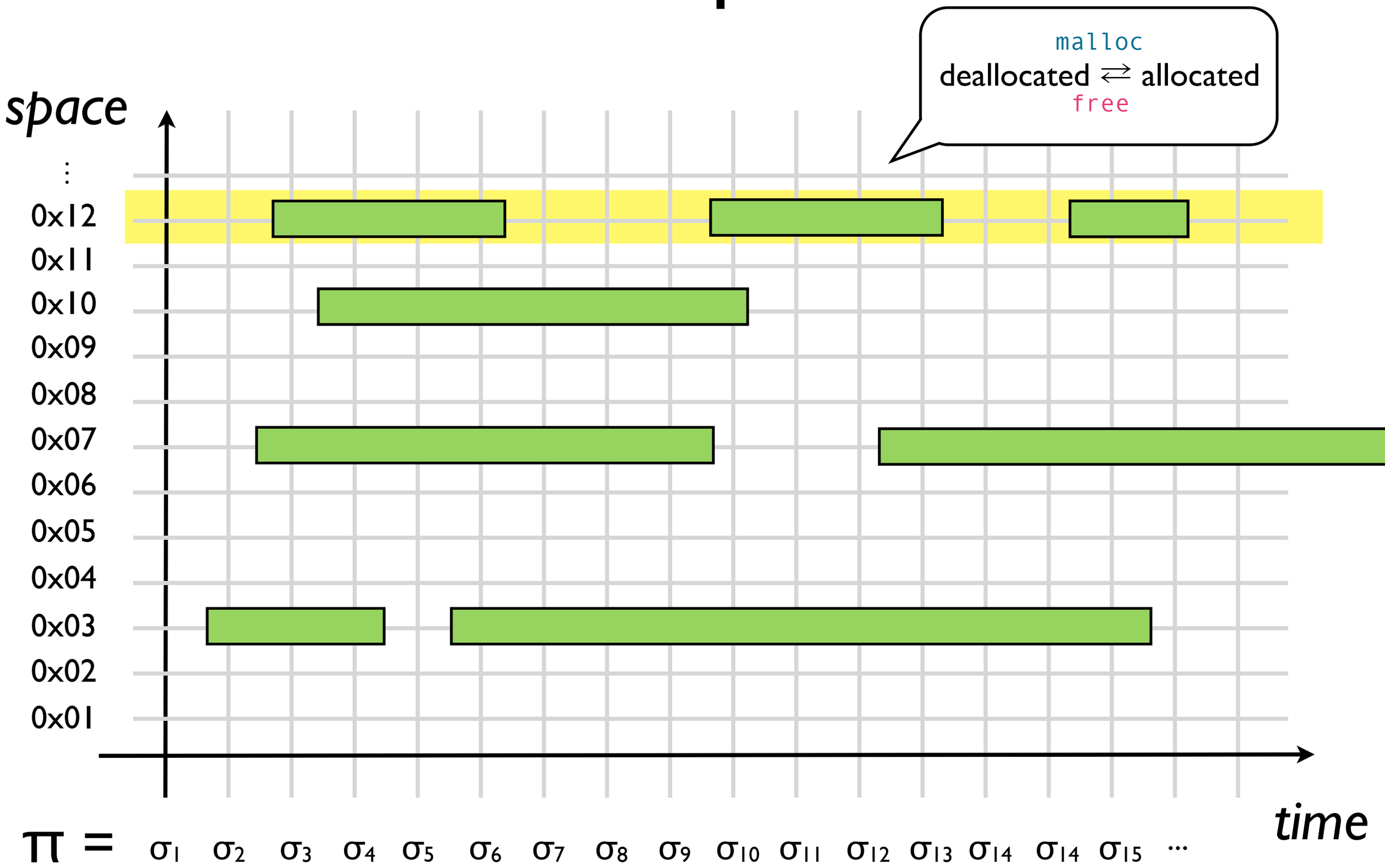
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



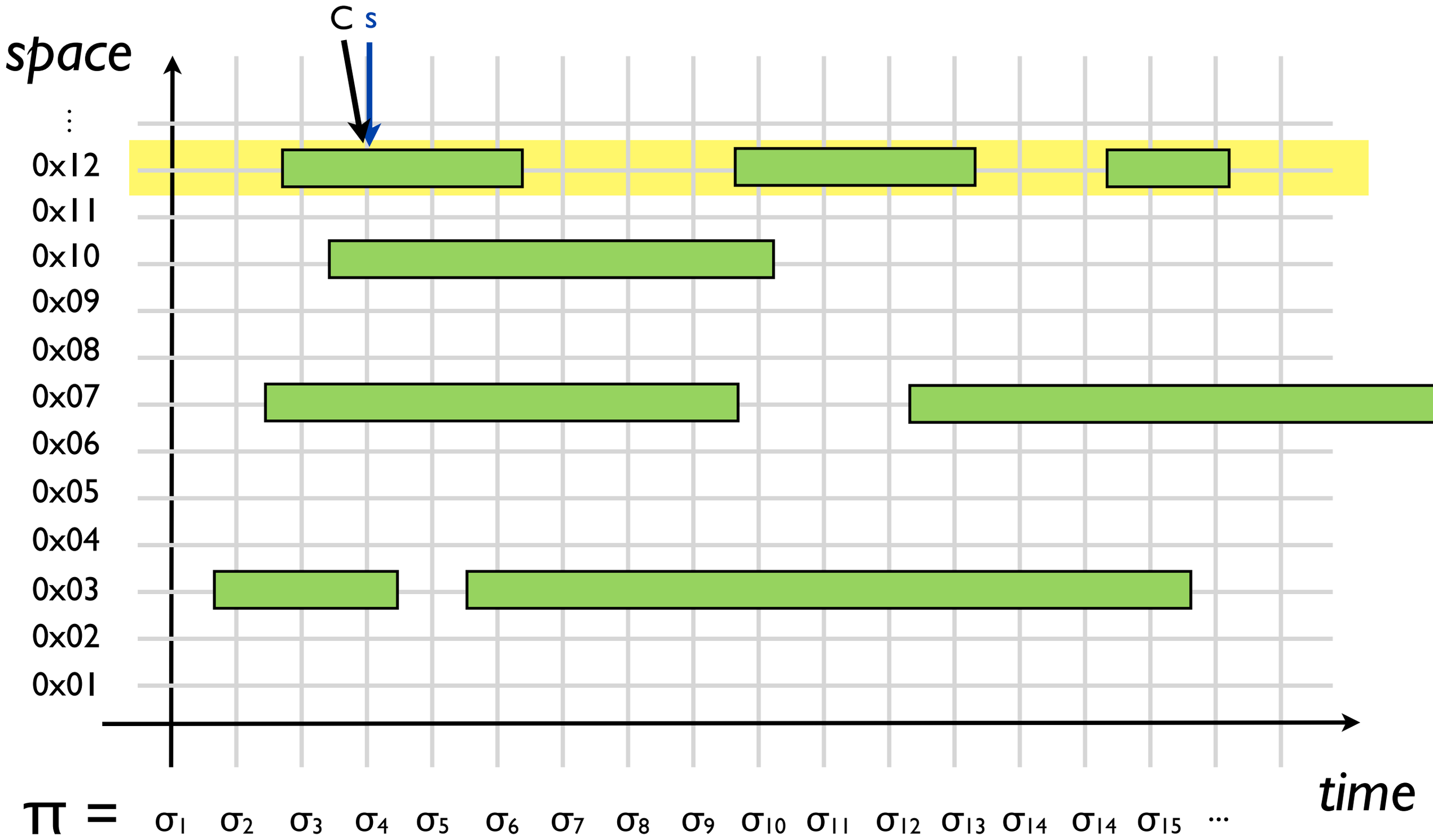
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

dual views: space / time



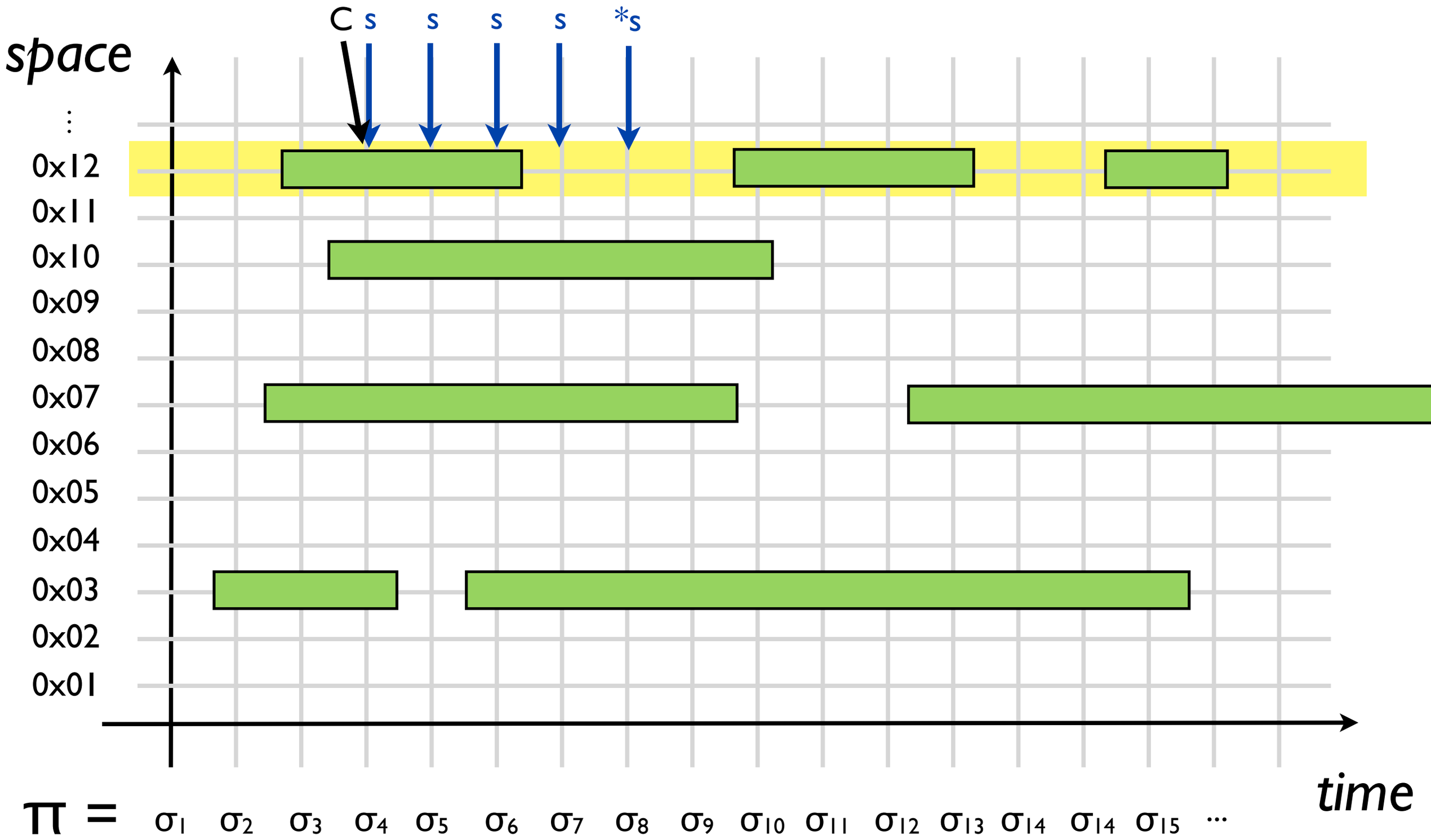
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

memory safety (temporal view)



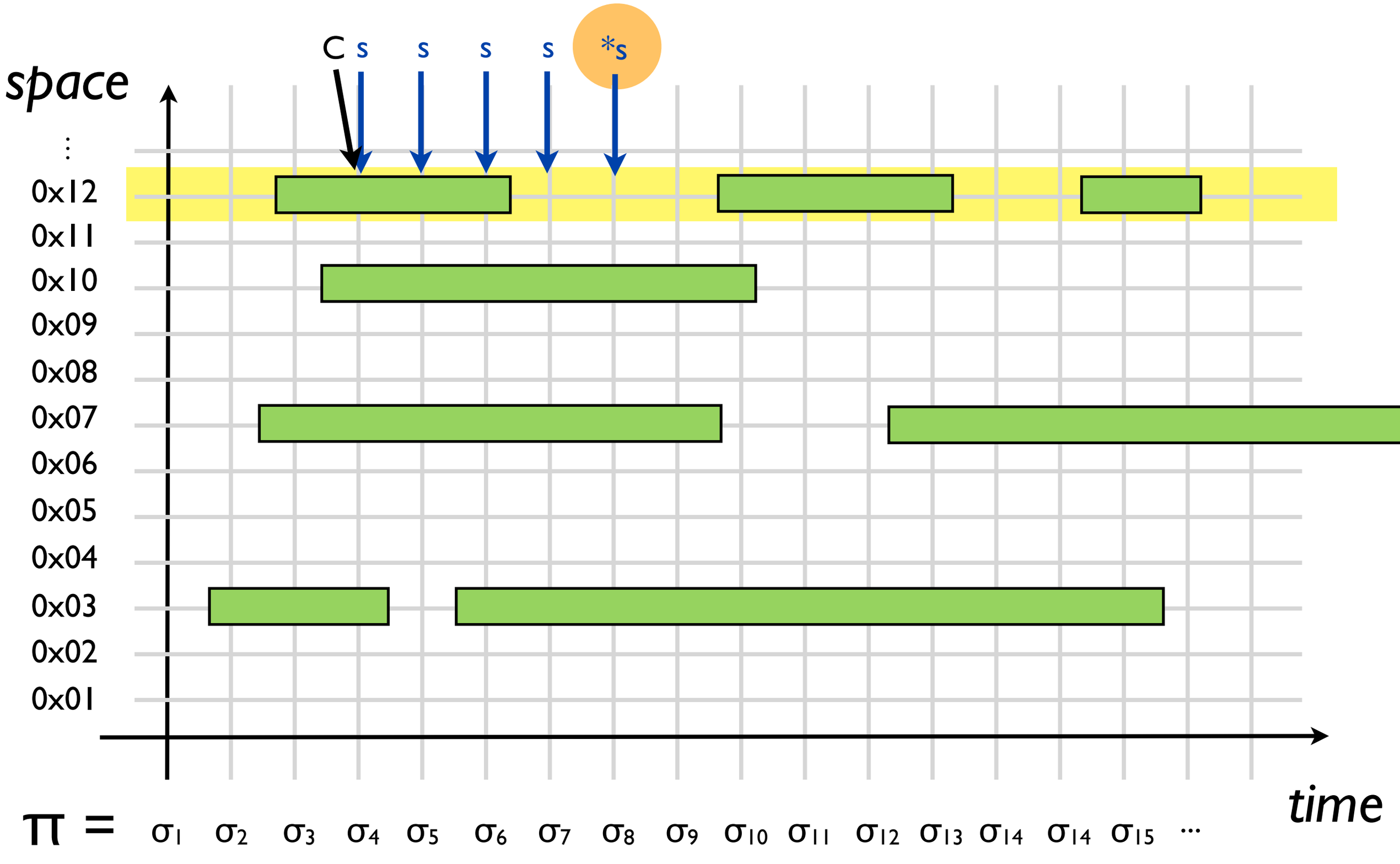
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

memory safety (temporal view)



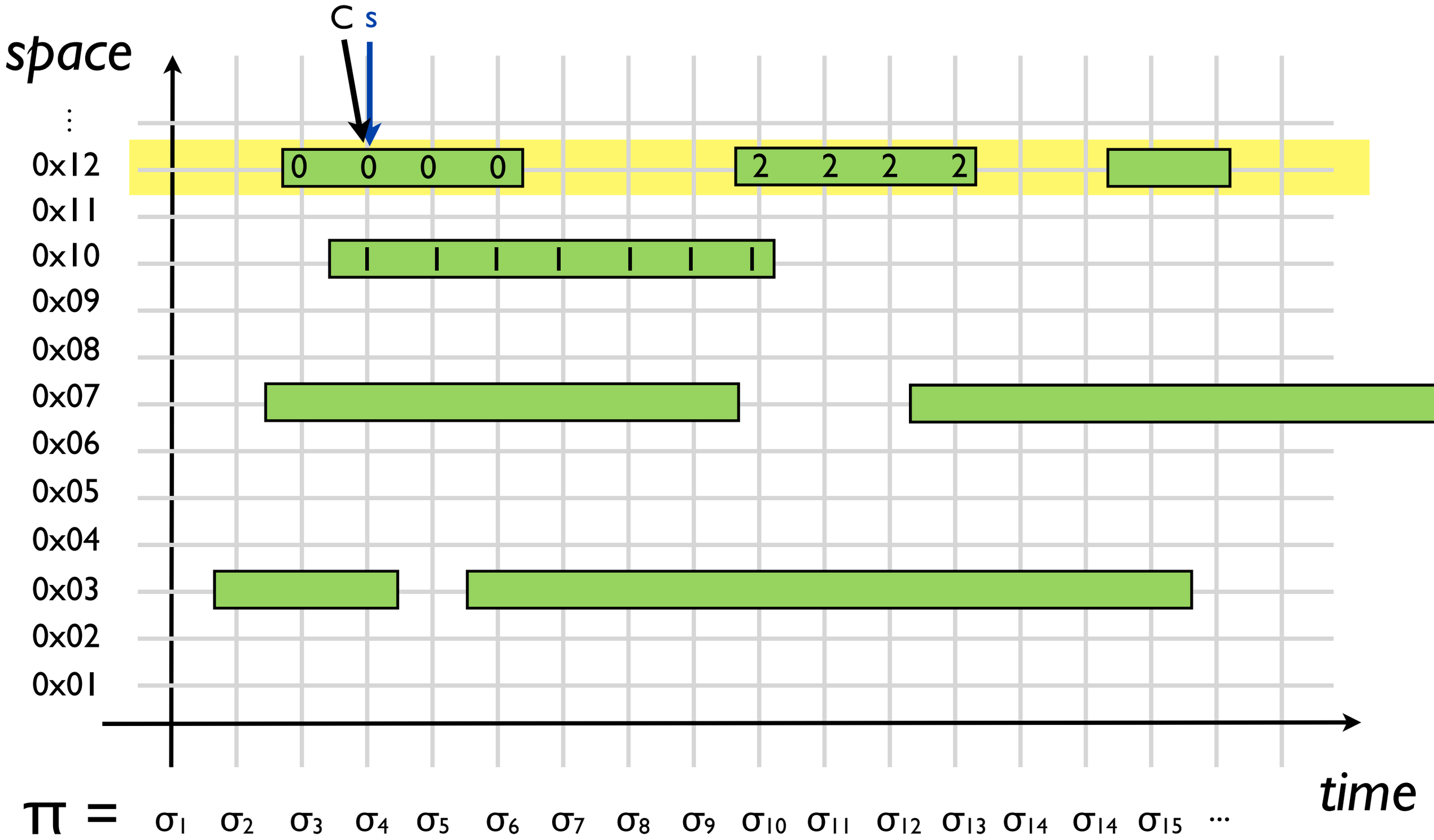
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

memory safety (temporal view)



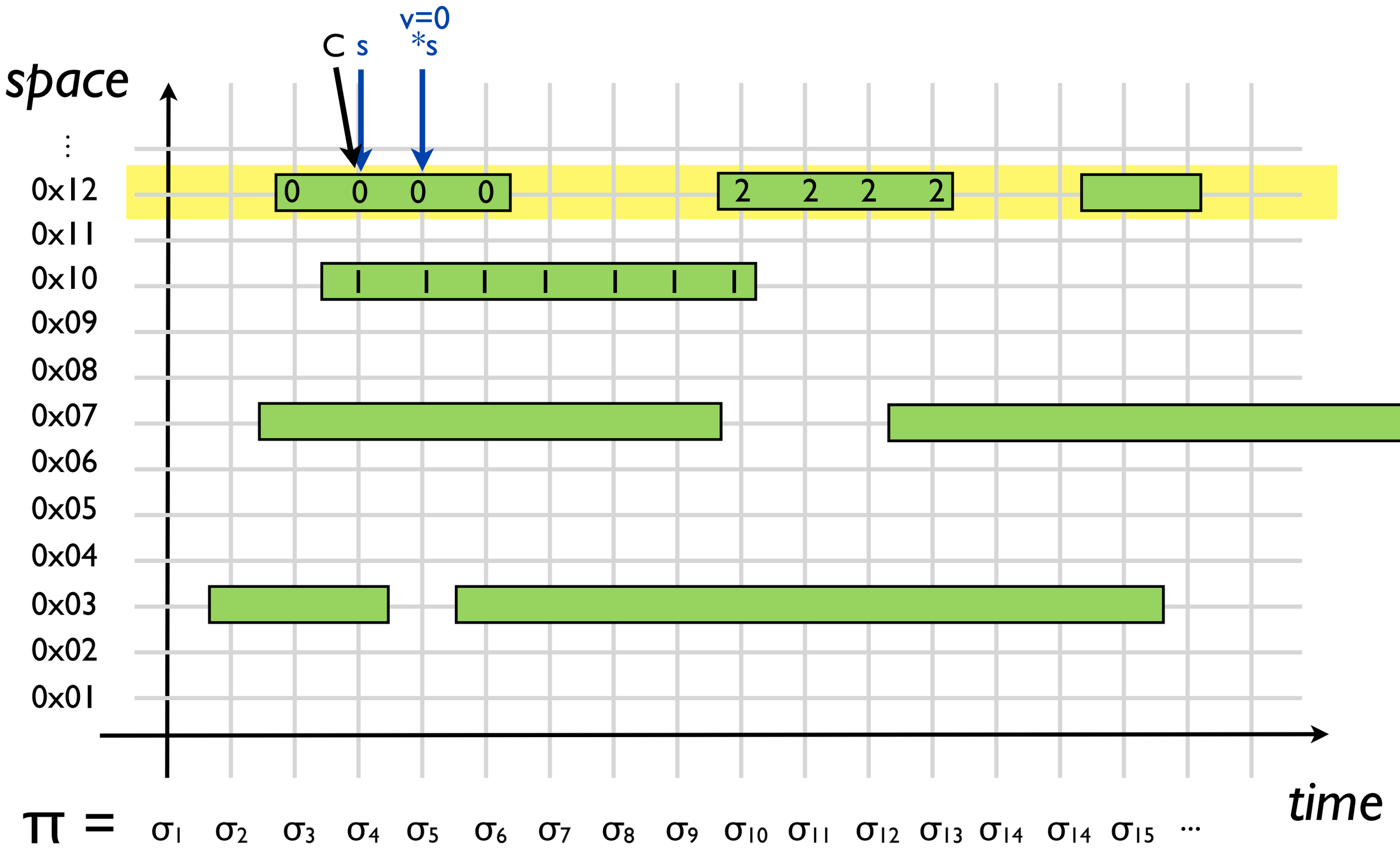
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

ABA (temporal view)



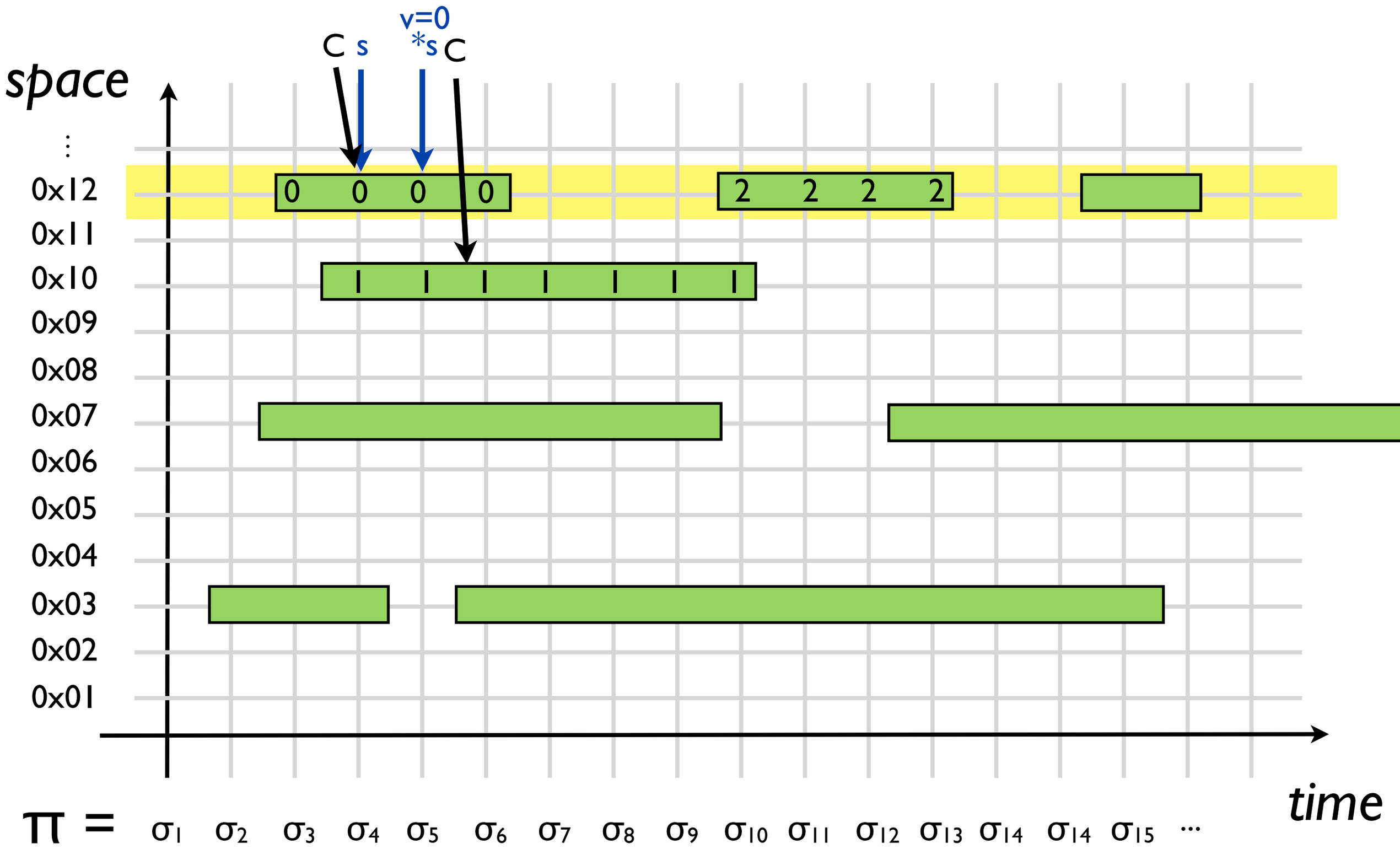
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

ABA (temporal view)



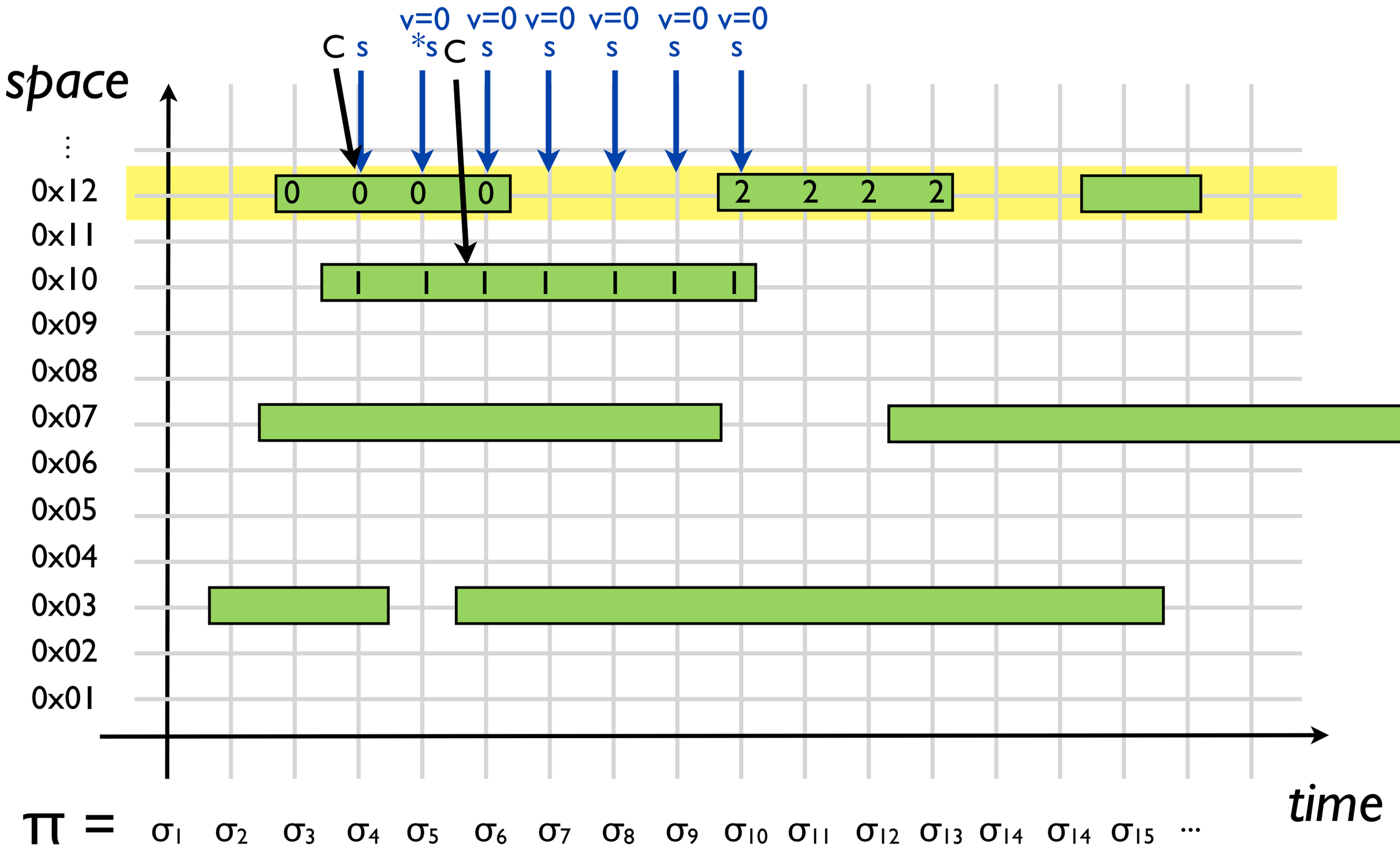
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

ABA (temporal view)



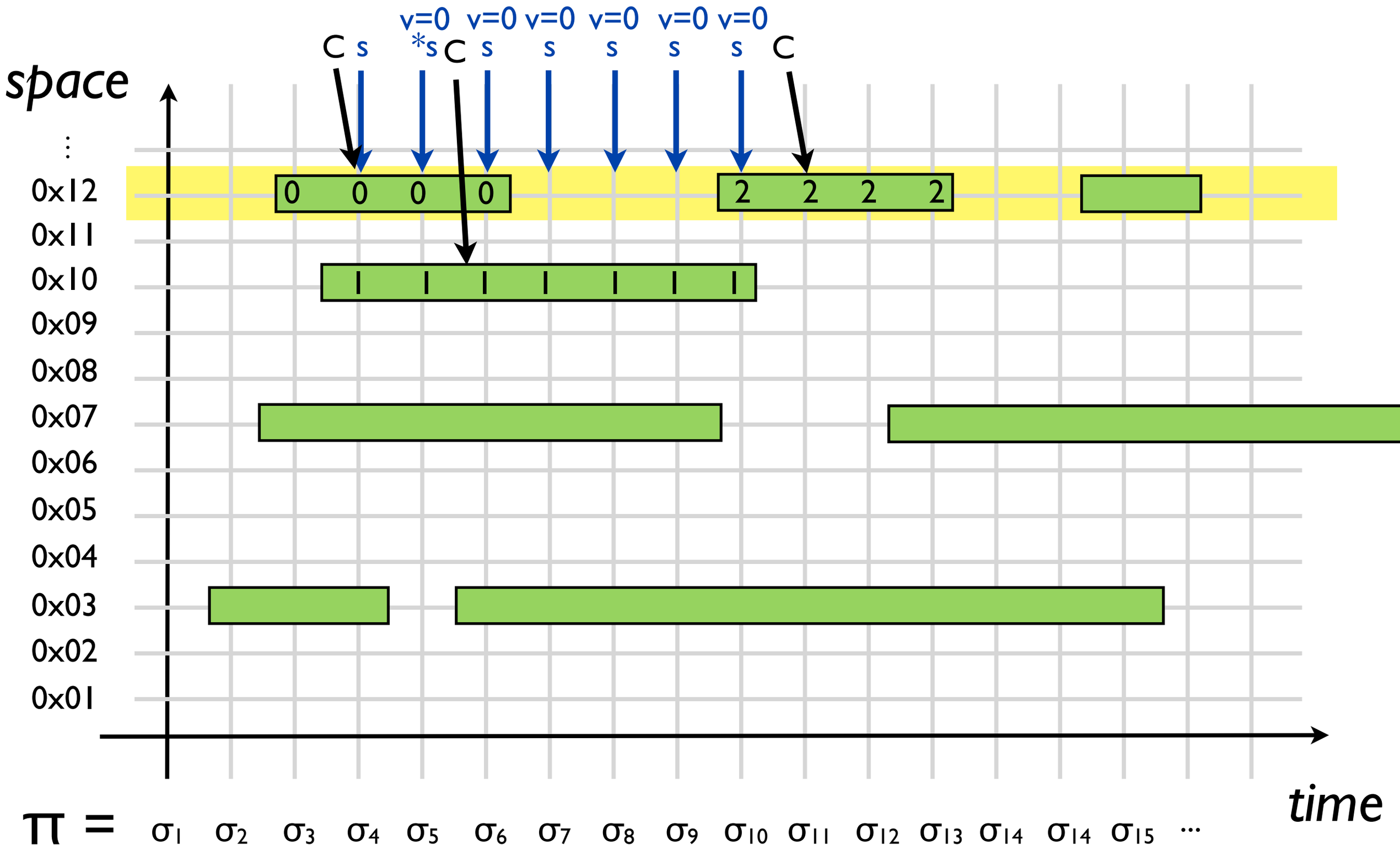
$\pi =$ σ_1 σ_2 σ_3 σ_4 σ_5 σ_6 σ_7 σ_8 σ_9 σ_{10} σ_{11} σ_{12} σ_{13} σ_{14} σ_{14} σ_{15} …

ABA (temporal view)



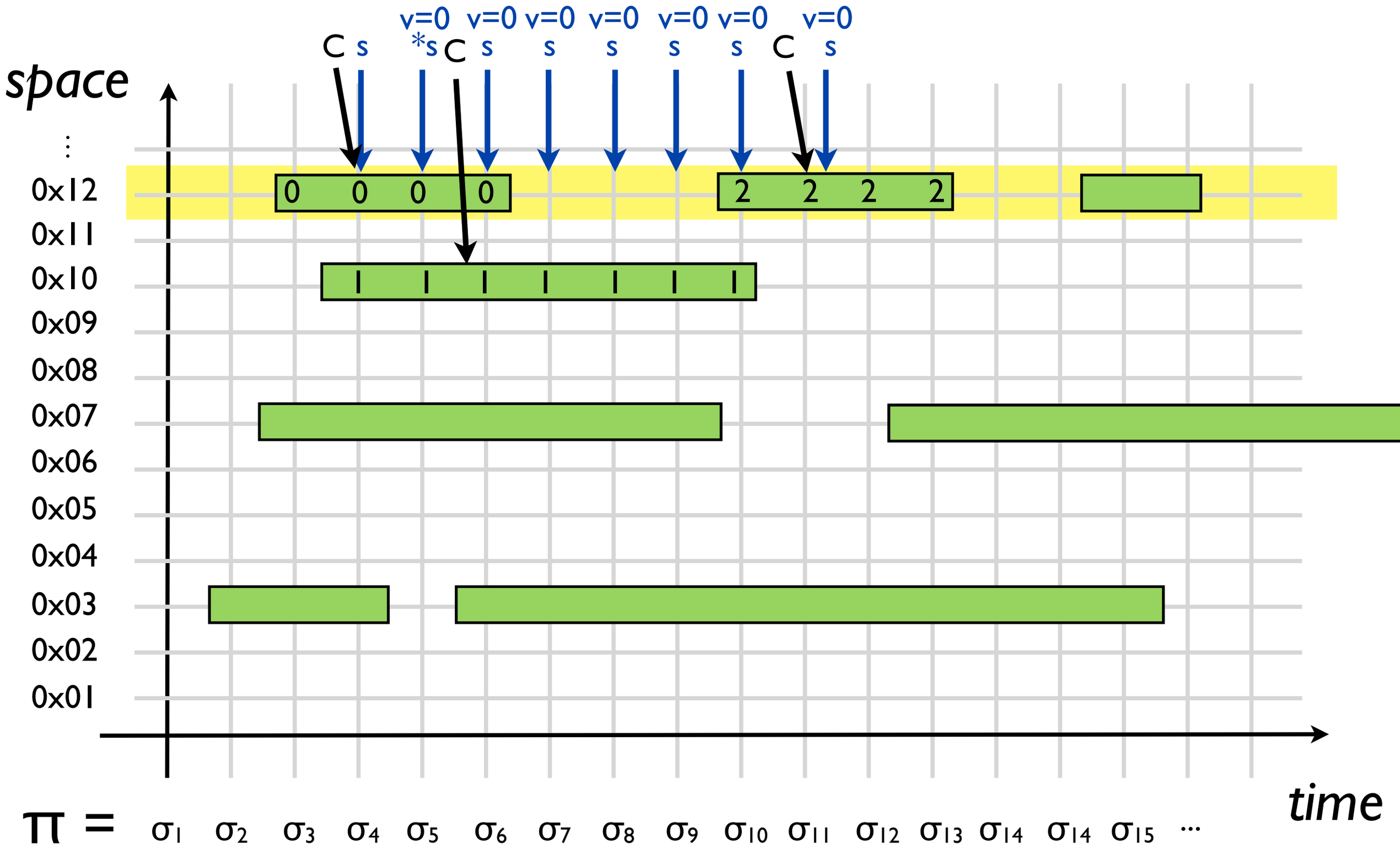
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

ABA (temporal view)



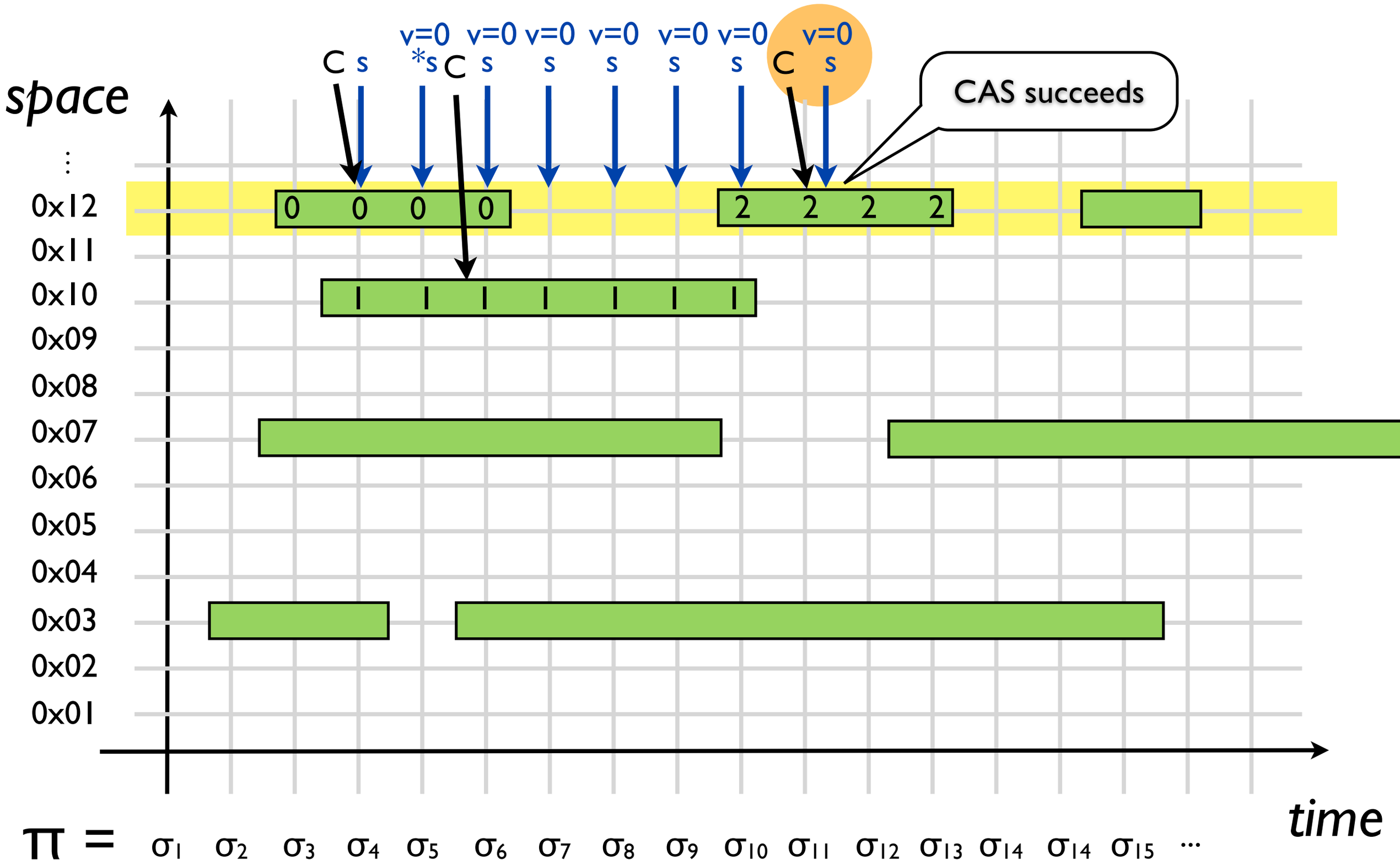
$\pi = \sigma_1\ \sigma_2\ \sigma_3\ \sigma_4\ \sigma_5\ \sigma_6\ \sigma_7\ \sigma_8\ \sigma_9\ \sigma_{10}\ \sigma_{11}\ \sigma_{12}\ \sigma_{13}\ \sigma_{14}\ \sigma_{14}\ \sigma_{15}\ \ldots$

ABA (temporal view)



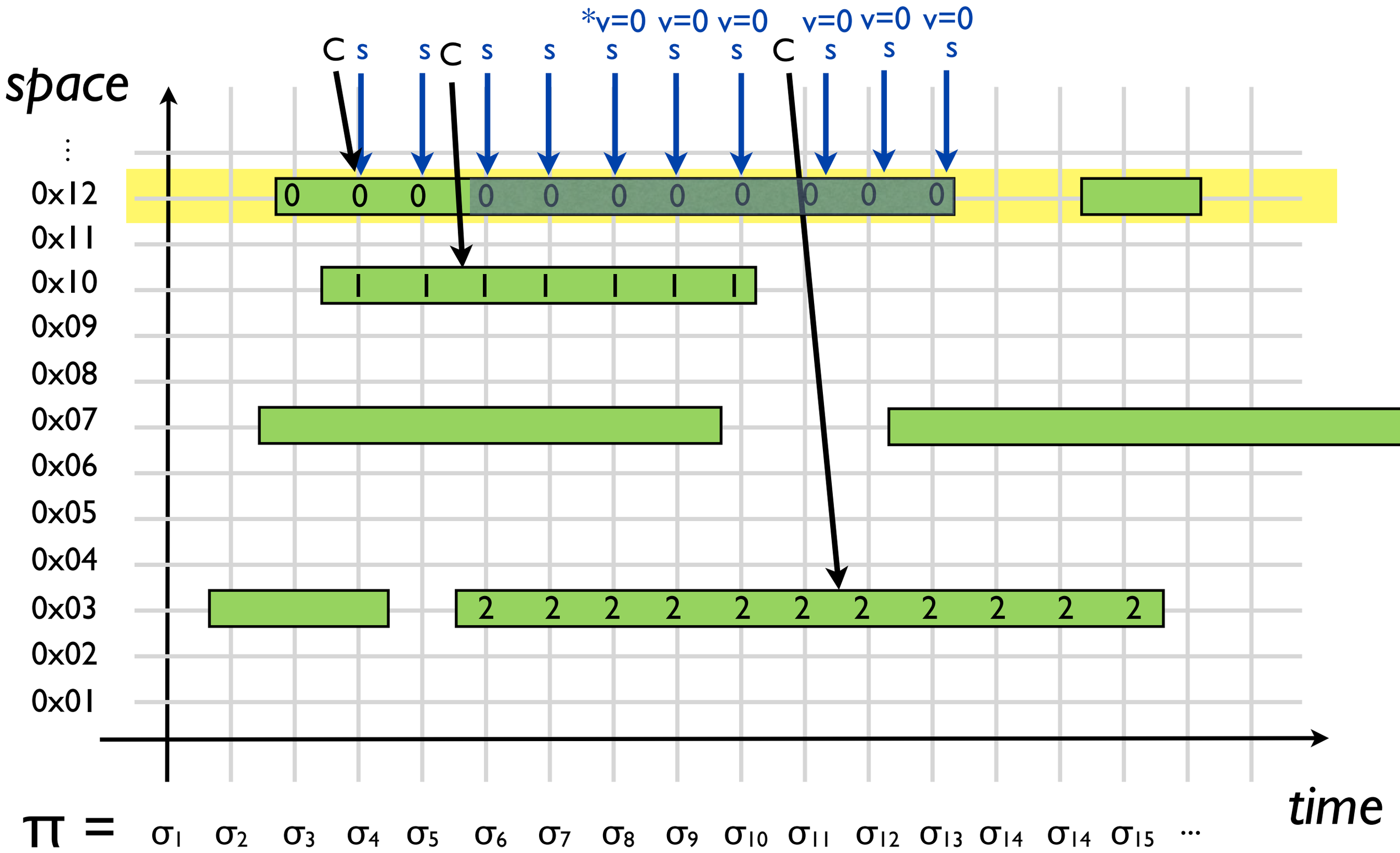
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

ABA (temporal view)



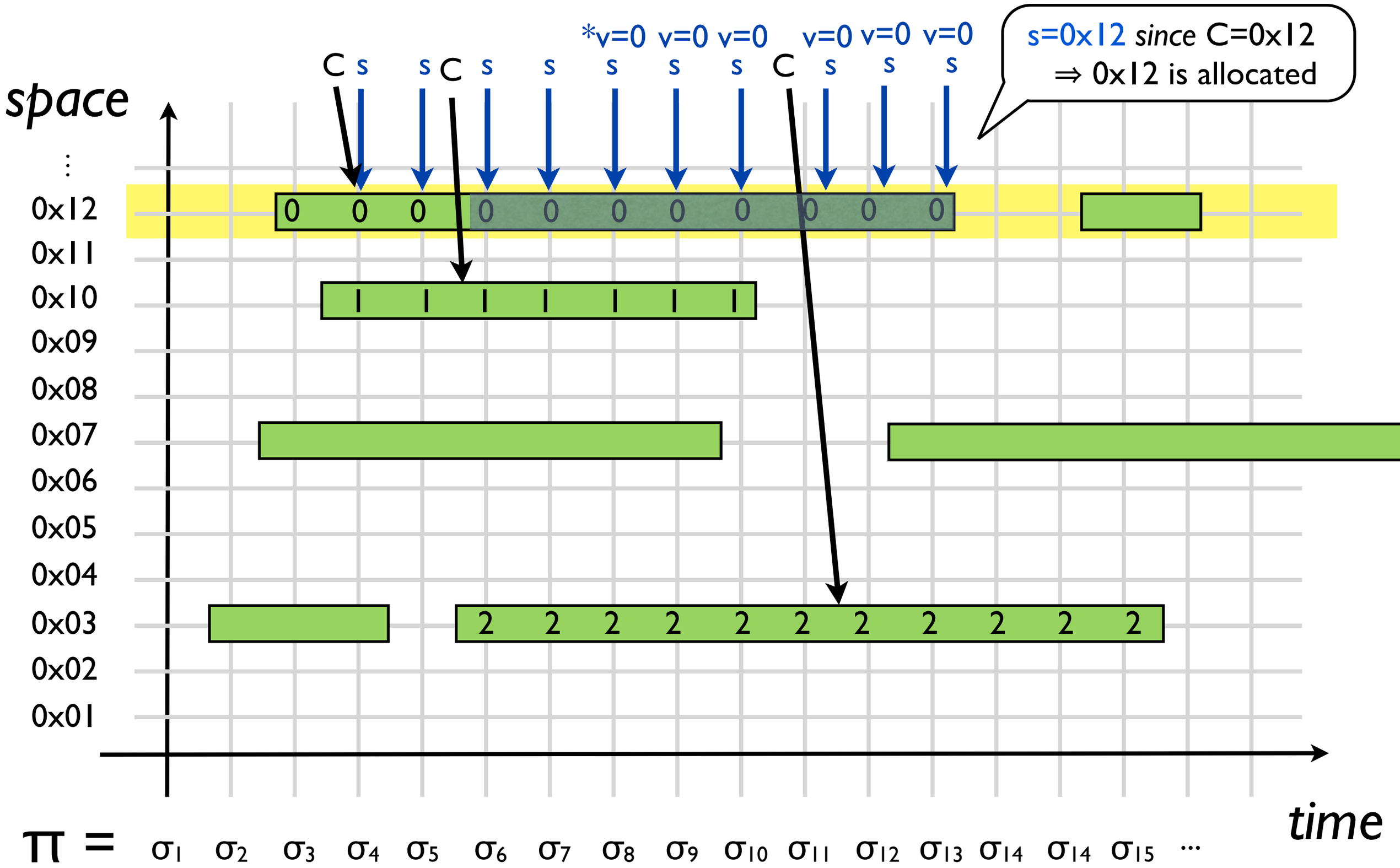
$\pi = \sigma_1\ \sigma_2\ \sigma_3\ \sigma_4\ \sigma_5\ \sigma_6\ \sigma_7\ \sigma_8\ \sigma_9\ \sigma_{10}\ \sigma_{11}\ \sigma_{12}\ \sigma_{13}\ \sigma_{14}\ \sigma_{14}\ \sigma_{15}\ \ldots$

what we want, ideally



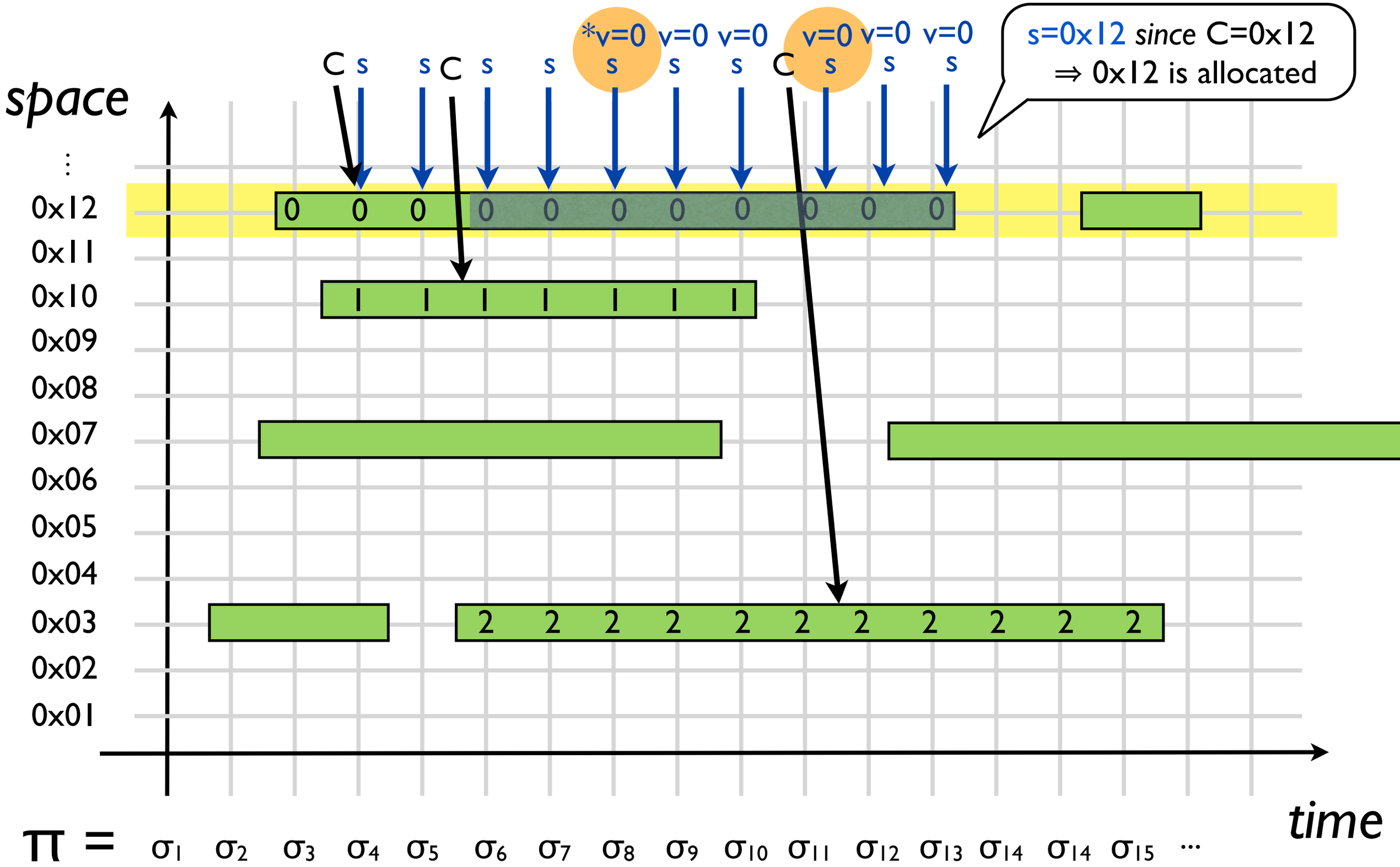
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

what we want, ideally



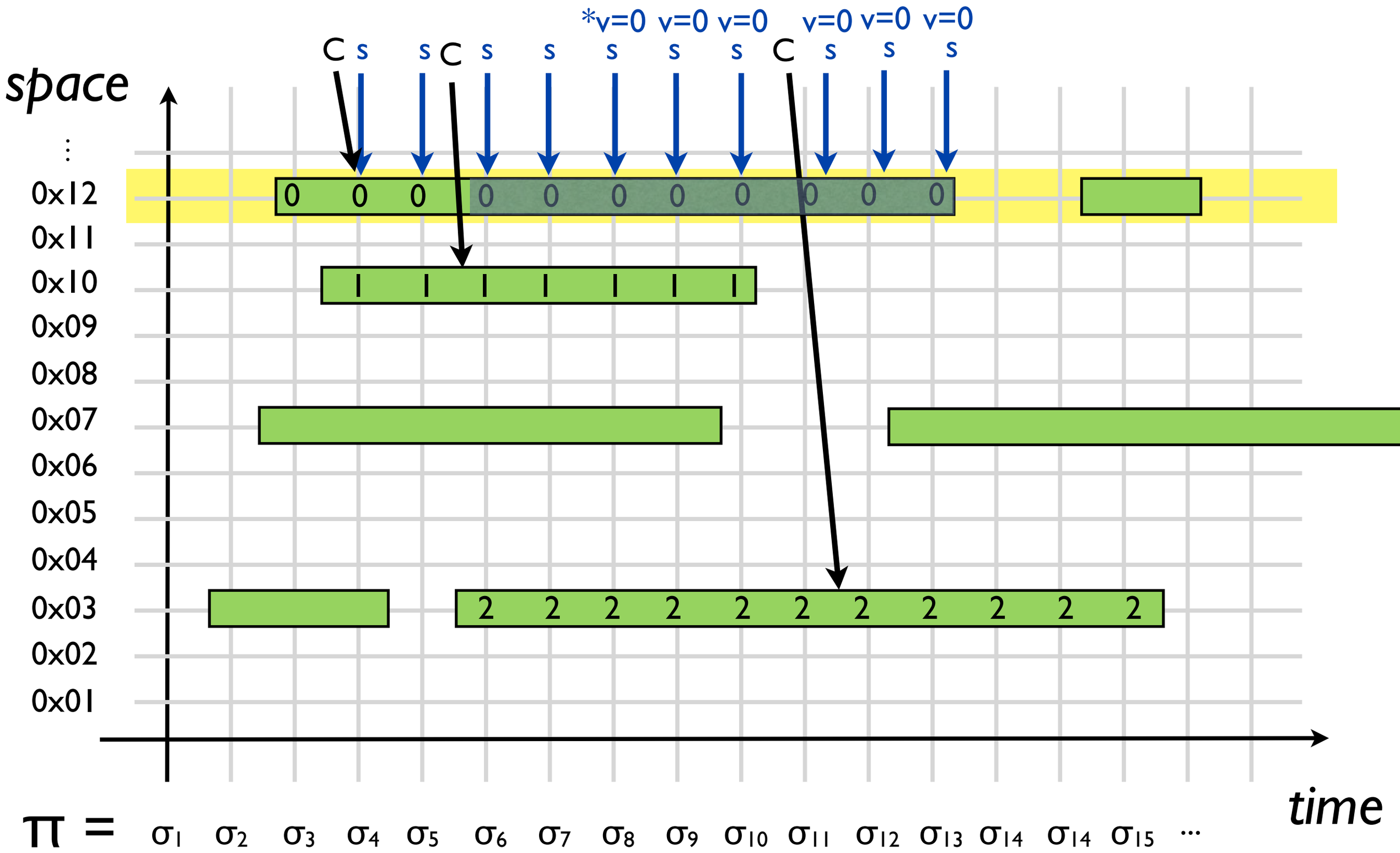
$\pi = \sigma_1\ \sigma_2\ \sigma_3\ \sigma_4\ \sigma_5\ \sigma_6\ \sigma_7\ \sigma_8\ \sigma_9\ \sigma_{10}\ \sigma_{11}\ \sigma_{12}\ \sigma_{13}\ \sigma_{14}\ \sigma_{14}\ \sigma_{15}\ \cdots$

what we want, ideally



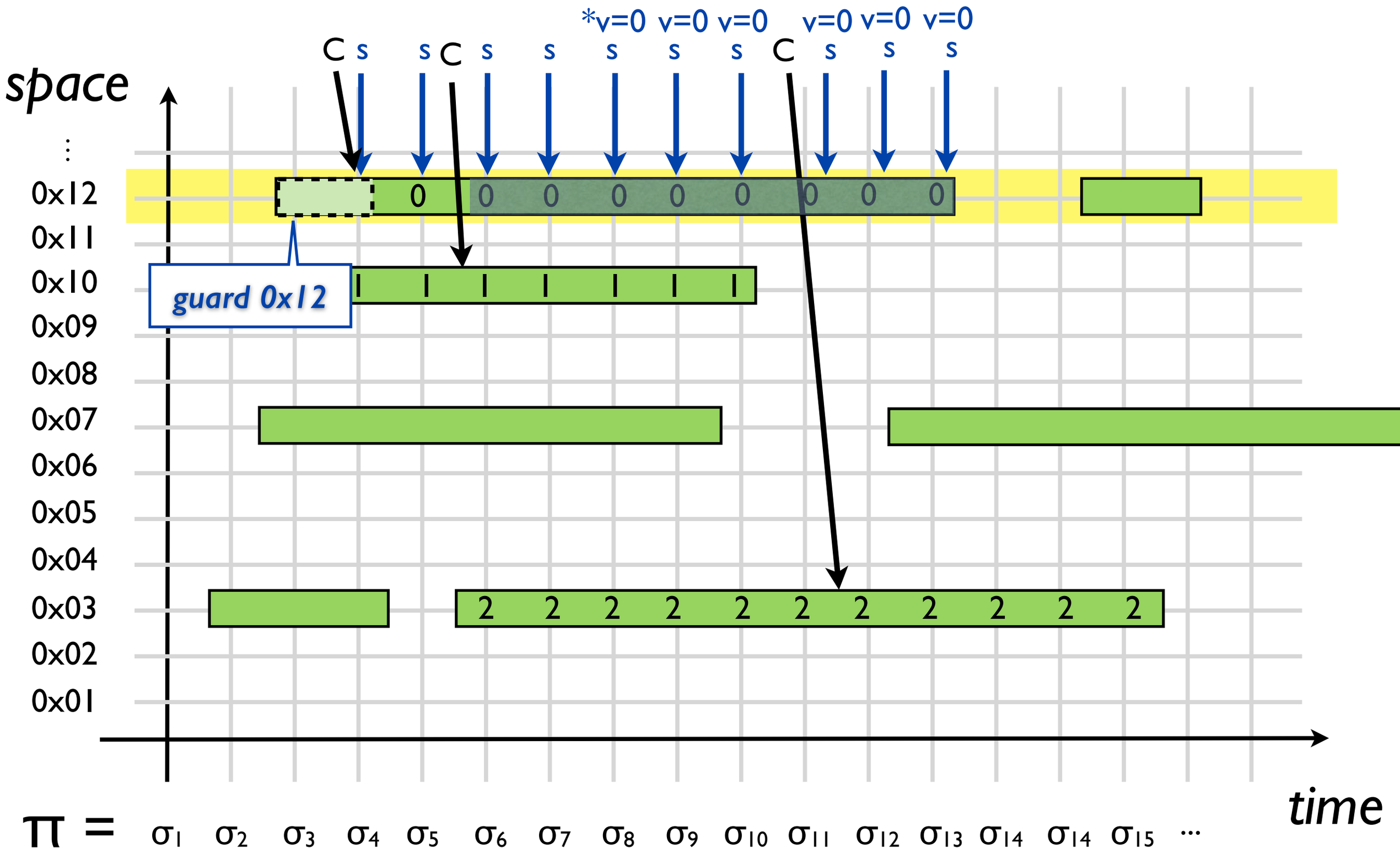
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

grace-based synchronization



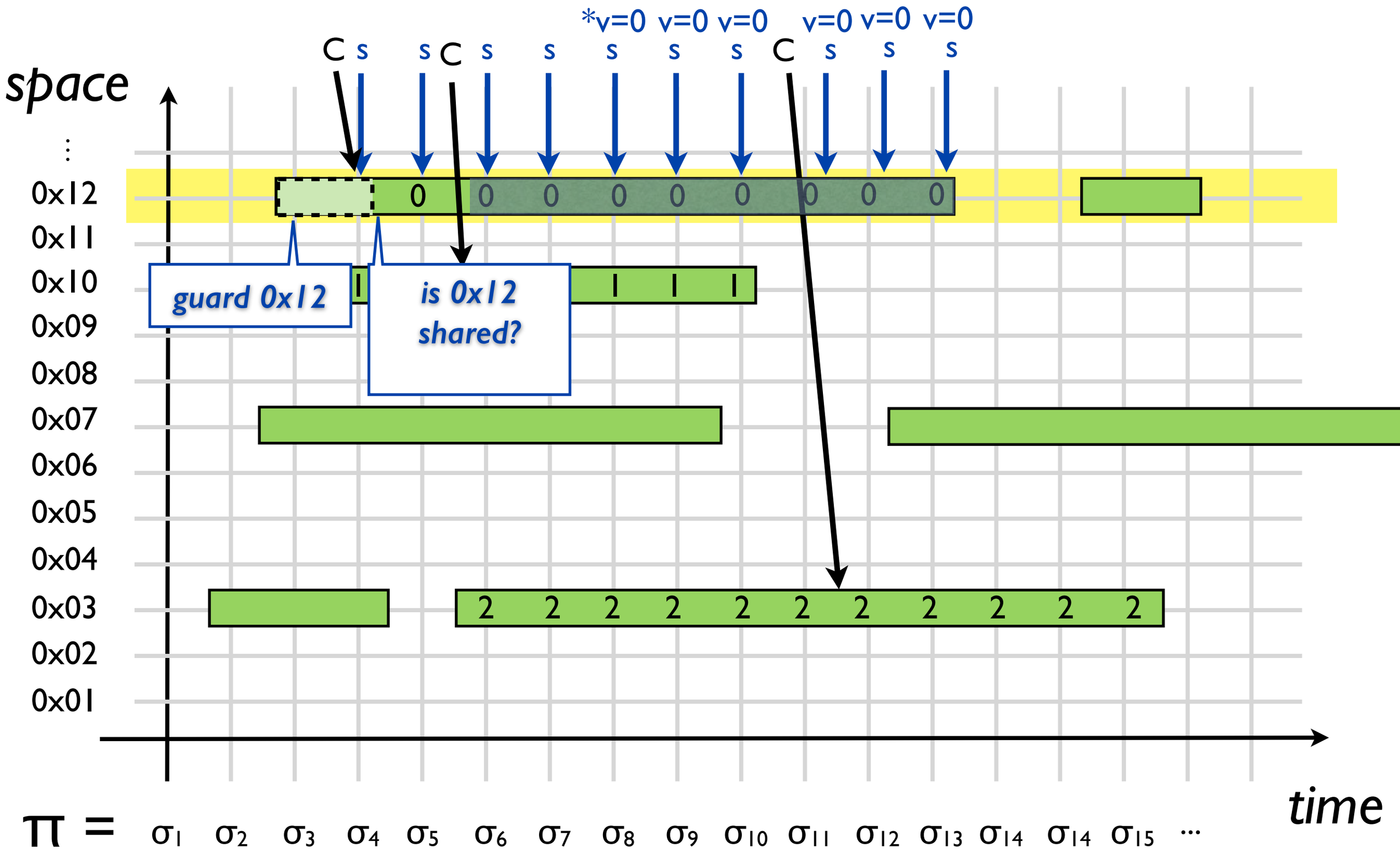
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

grace-based synchronization



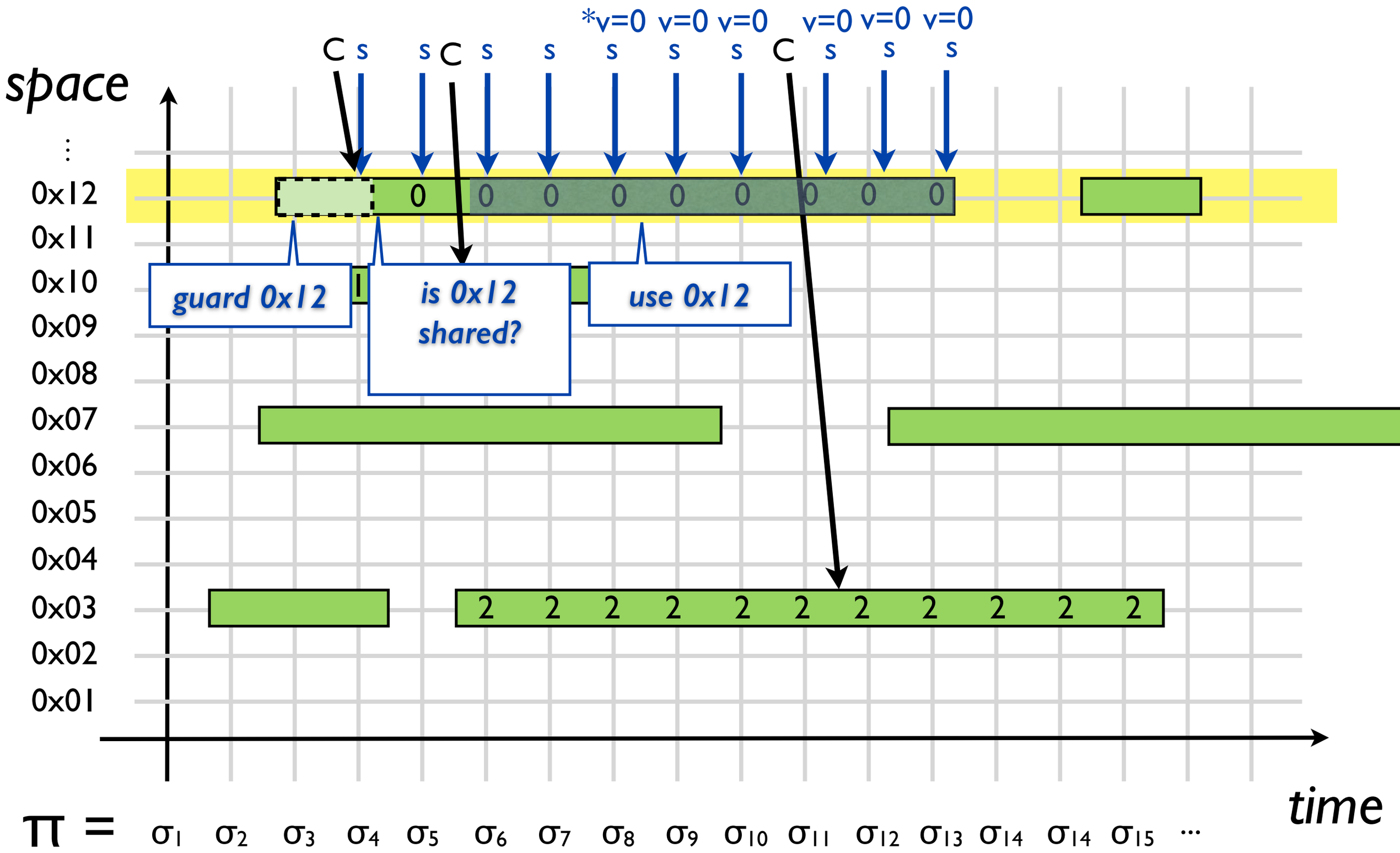
$\pi = \sigma_1\ \sigma_2\ \sigma_3\ \sigma_4\ \sigma_5\ \sigma_6\ \sigma_7\ \sigma_8\ \sigma_9\ \sigma_{10}\ \sigma_{11}\ \sigma_{12}\ \sigma_{13}\ \sigma_{14}\ \sigma_{14}\ \sigma_{15}\ \cdots$

grace-based synchronization



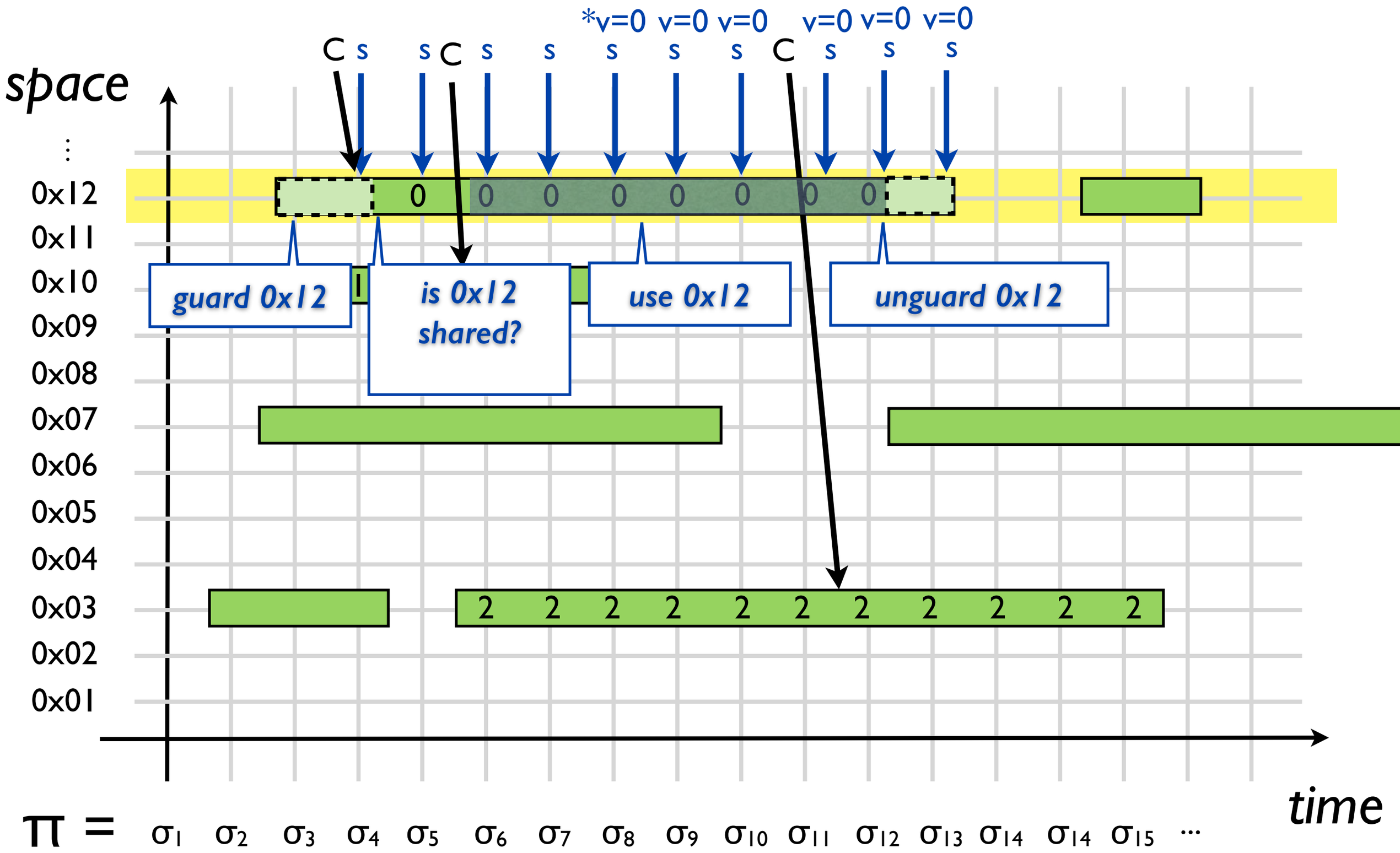
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \ldots$

grace-based synchronization



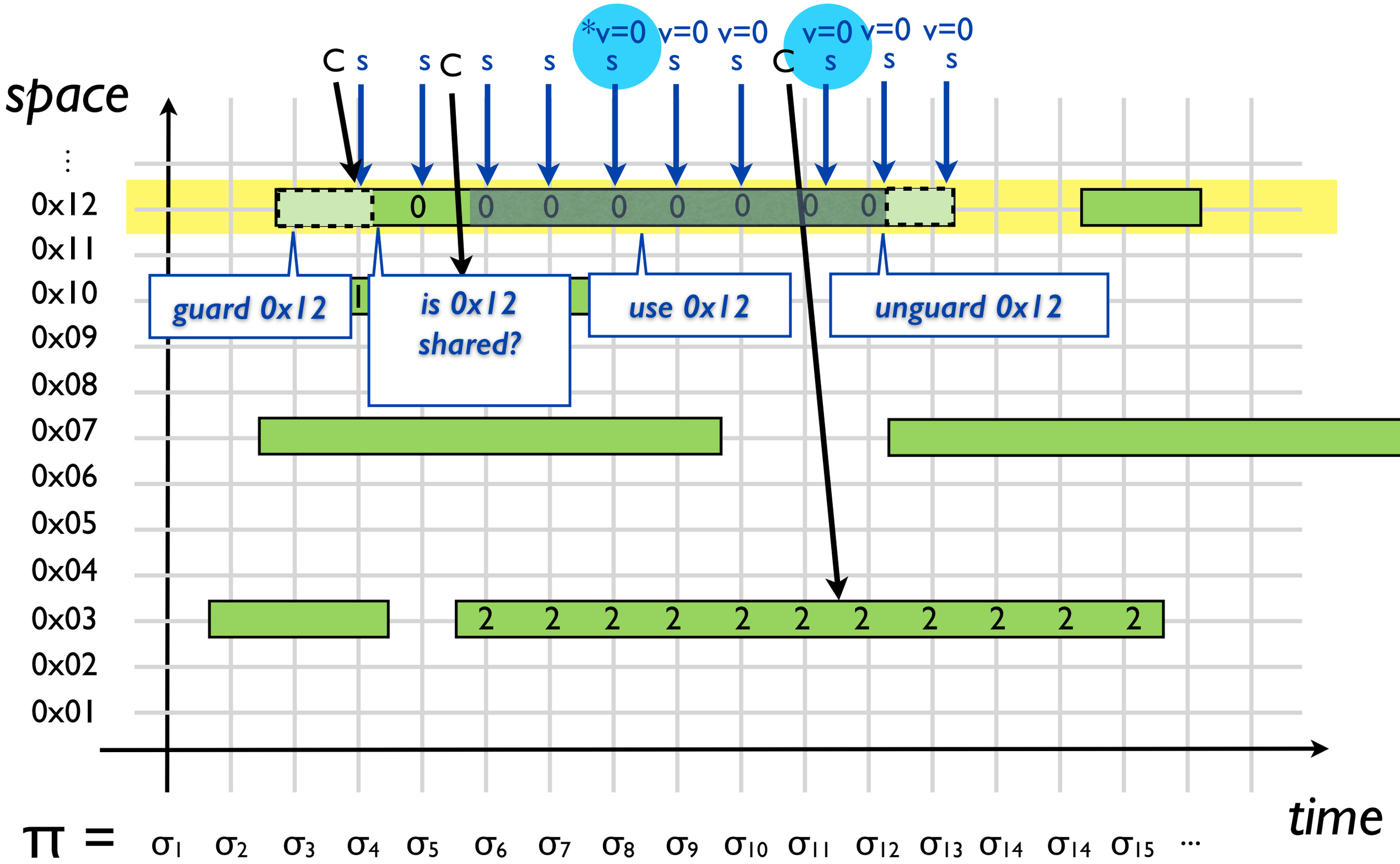
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

grace-based synchronization



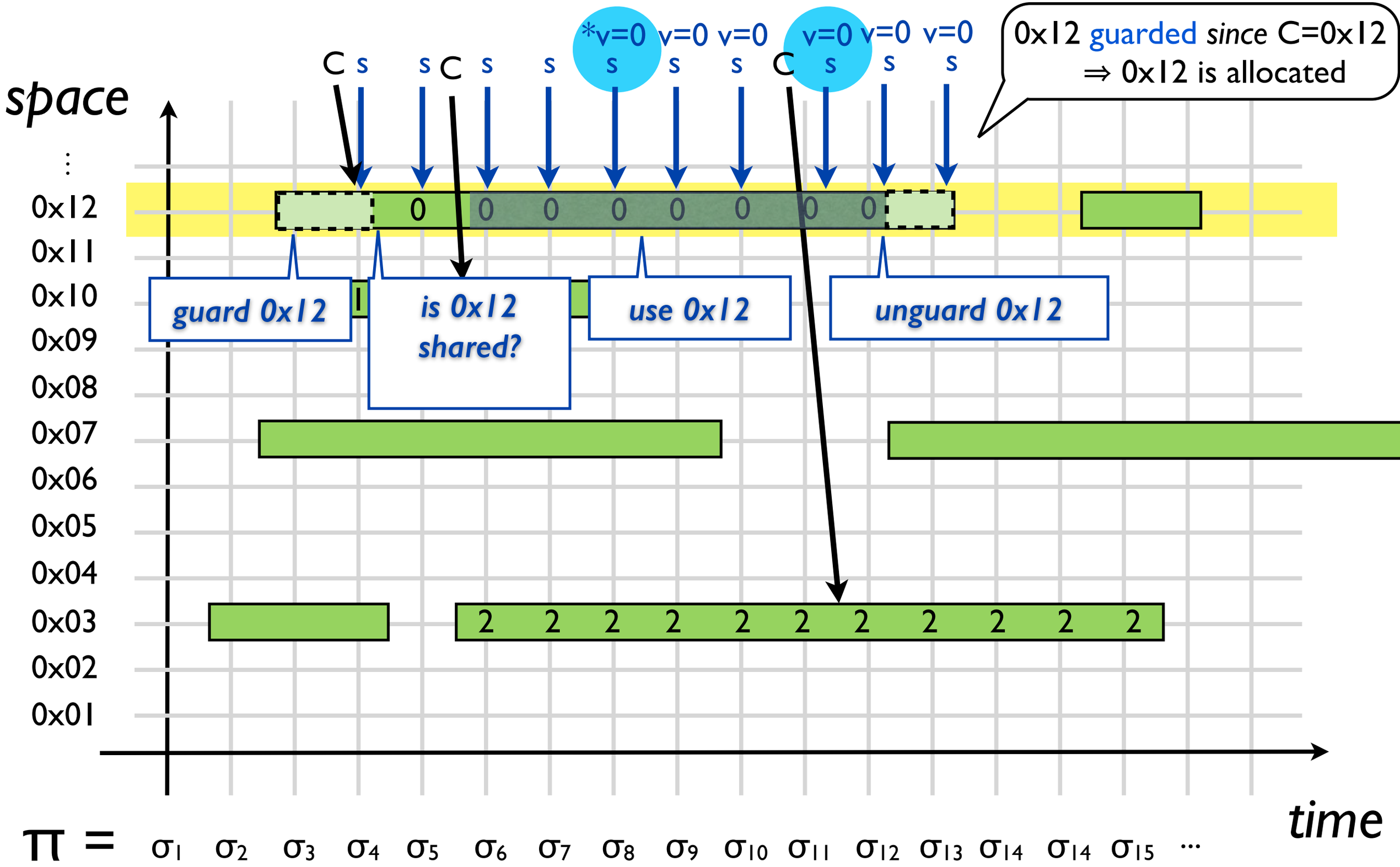
$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

grace-based synchronization



$\pi = \sigma_1 \ \sigma_2 \ \sigma_3 \ \sigma_4 \ \sigma_5 \ \sigma_6 \ \sigma_7 \ \sigma_8 \ \sigma_9 \ \sigma_{10} \ \sigma_{11} \ \sigma_{12} \ \sigma_{13} \ \sigma_{14} \ \sigma_{14} \ \sigma_{15} \ \cdots$

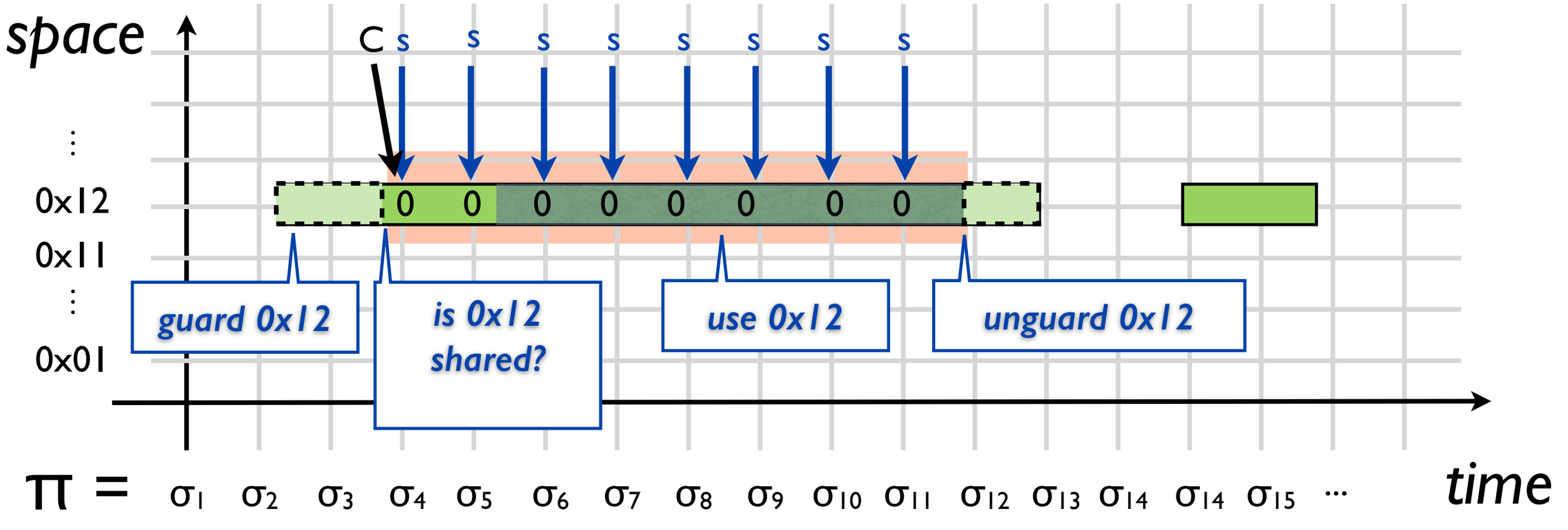
grace-based synchronization



$\pi = \sigma_1\ \sigma_2\ \sigma_3\ \sigma_4\ \sigma_5\ \sigma_6\ \sigma_7\ \sigma_8\ \sigma_9\ \sigma_{10}\ \sigma_{11}\ \sigma_{12}\ \sigma_{13}\ \sigma_{14}\ \sigma_{14}\ \sigma_{15}\ \cdots$

grace-based synchronization

- **grace period**$_{t,s}$ the period of time during which a thread *t* can access a shared node *s* without a fear that *s* might get deallocated



grace-based synchronization

- **grace period**$_{t,s}$ the period of time during which a thread *t* can access a shared node *s* without a fear that *s* might get deallocated

➡ *t* access node *s* during its grace period for *s*

➡ *t* reclaim node *s* after the grace periods of all *t'* for *s*

temporal invariant

- **grace period$_{t,s}$**
  - if t guards s *since* s was shared then s is allocated

temporal invariant

- **grace invariant:** ∀t,s. **grace period**$_{t,s}$
  - ∀t,s. t guards s *since* s was shared then s is allocated
  - ∀t,s. $G_{t,s}$ *since* R_s ⇒ A_s

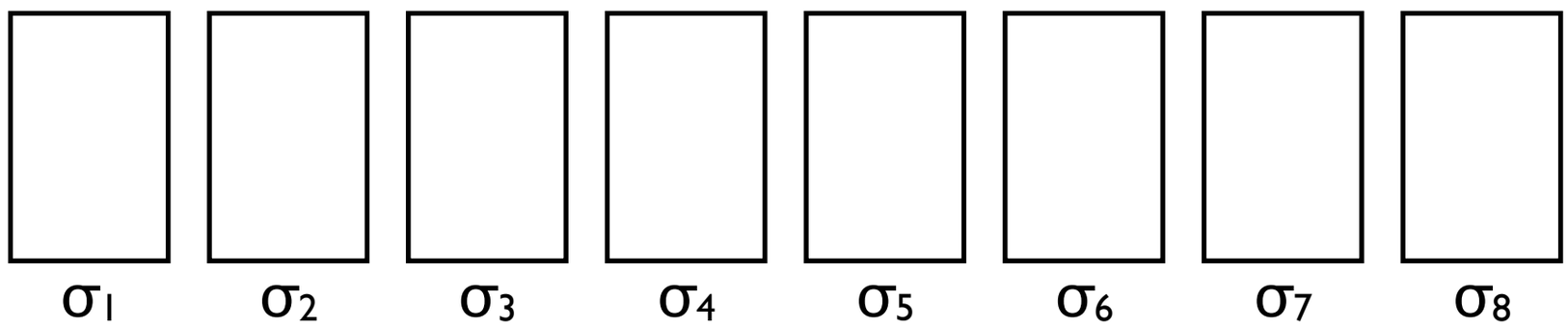
temporal invariant

- **grace invariant:** ∀t,s. **grace period**$_{t,s}$
  - ∀t,s. t guards s *since* s was shared then s is allocated
  - ∀t,s. $G_{t,s}$ *since* R_s ⇒ A_s

- **Hazard Pointers** [Michael'02]
- **RCU** [McKenney+'98]
- **Epoch** [Fraser+'03]

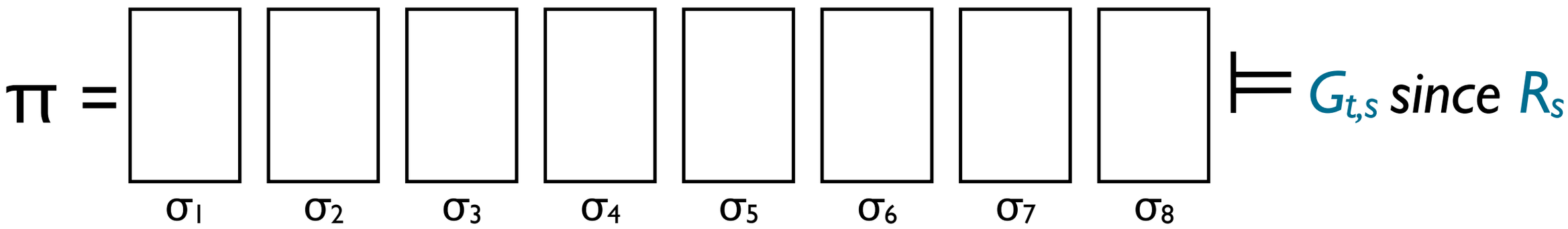
temporal invariant

- **grace invariant:** ∀t,s. **grace period**$_{t,s}$
  - ∀t,s. t guards s *since* s was shared then s is allocated
  - ∀t,s. $G_{t,s}$ *since* R_s ⇒ A_s

π = 

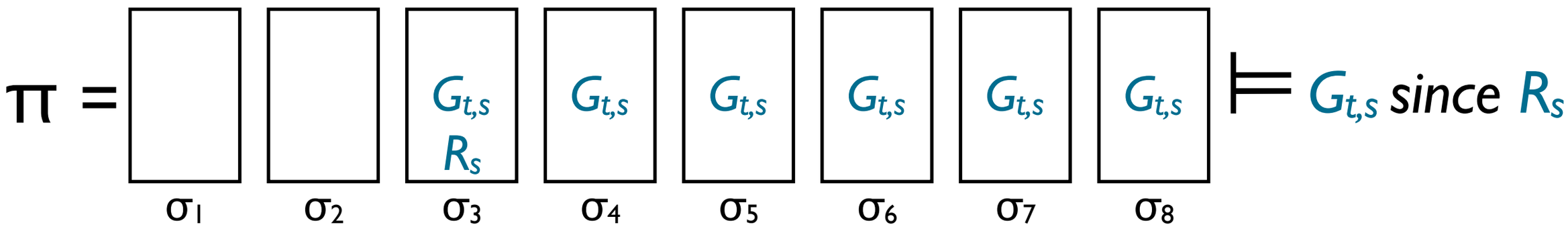
temporal invariant

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s



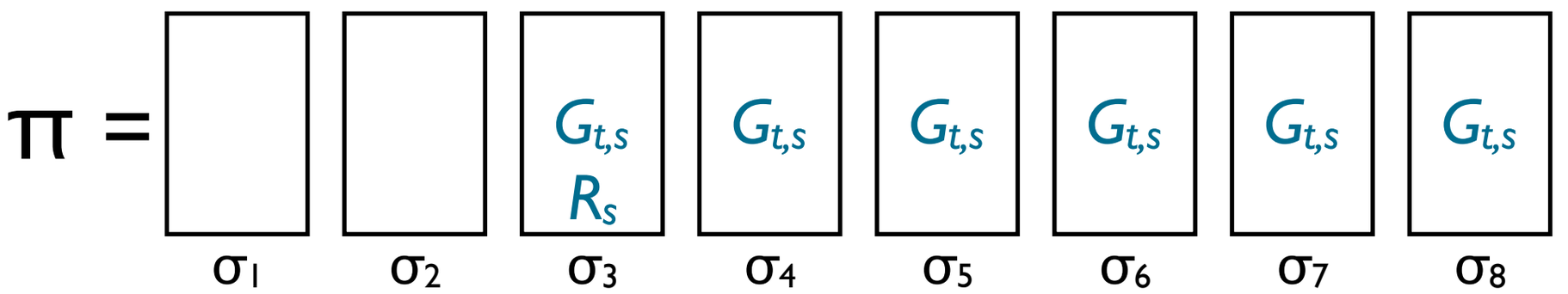
temporal invariant

- **grace invariant:** ∀t,s. **grace period**$_{t,s}$
  - ∀t,s. t guards s *since* s was shared then s is allocated
  - ∀t,s. $G_{t,s}$ *since* R_s ⇒ A_s

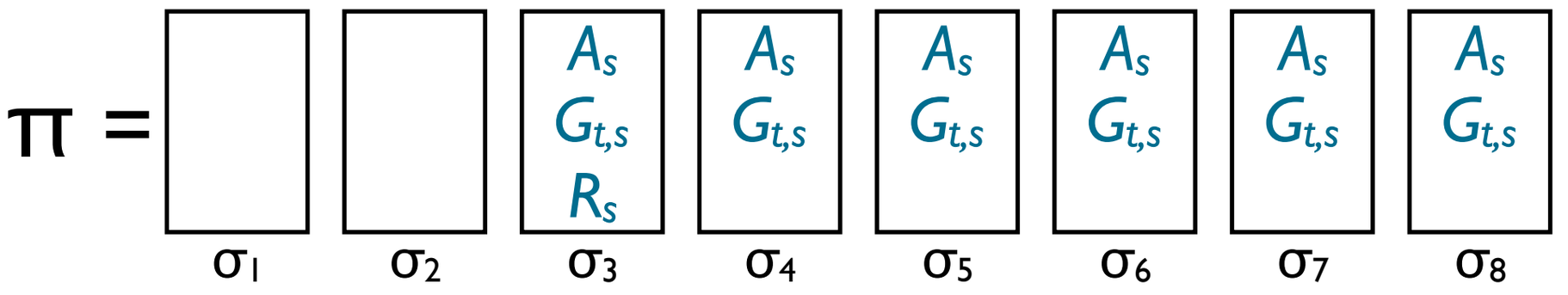


temporal invariant

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s



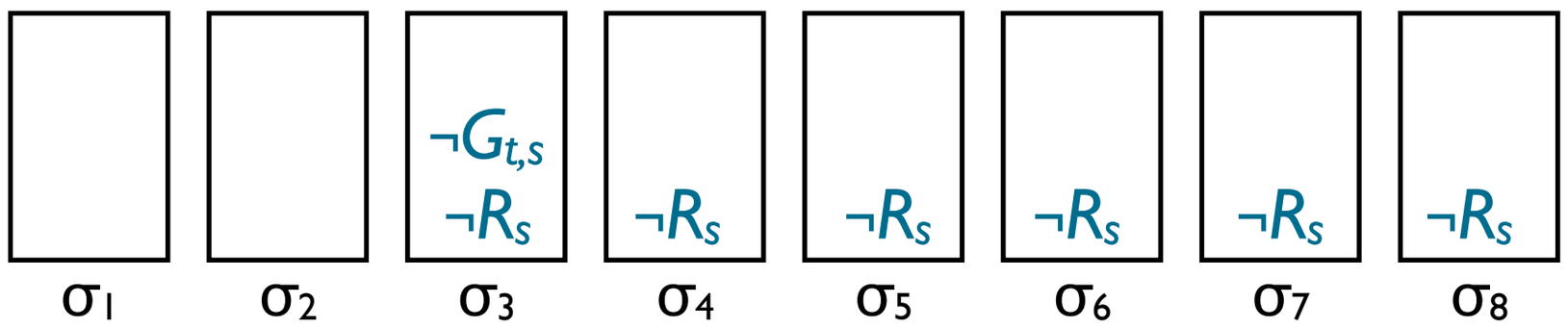
$\models G_{t,s}$ *since* R_s



$\models G_{t,s}$ *since* $R_s \Rightarrow A_s$

temporal invariant

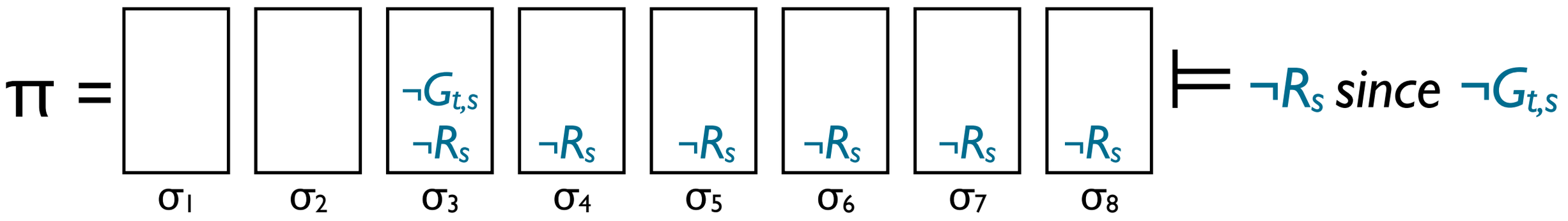
- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

$\pi =$ 

$\models \neg R_s$ *since* $\neg G_{t,s}$

temporal invariant

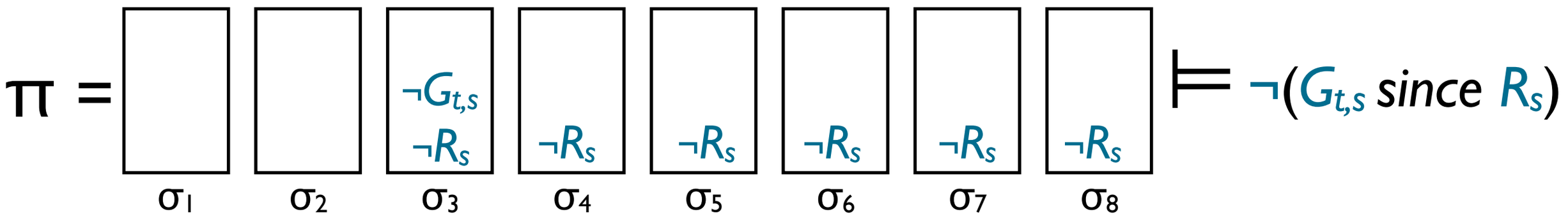
- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s



tautology: $(\neg R_s \text{ since } \neg G_{t,s}) \Rightarrow \neg(G_{t,s} \text{ since } R_s)$

temporal invariant

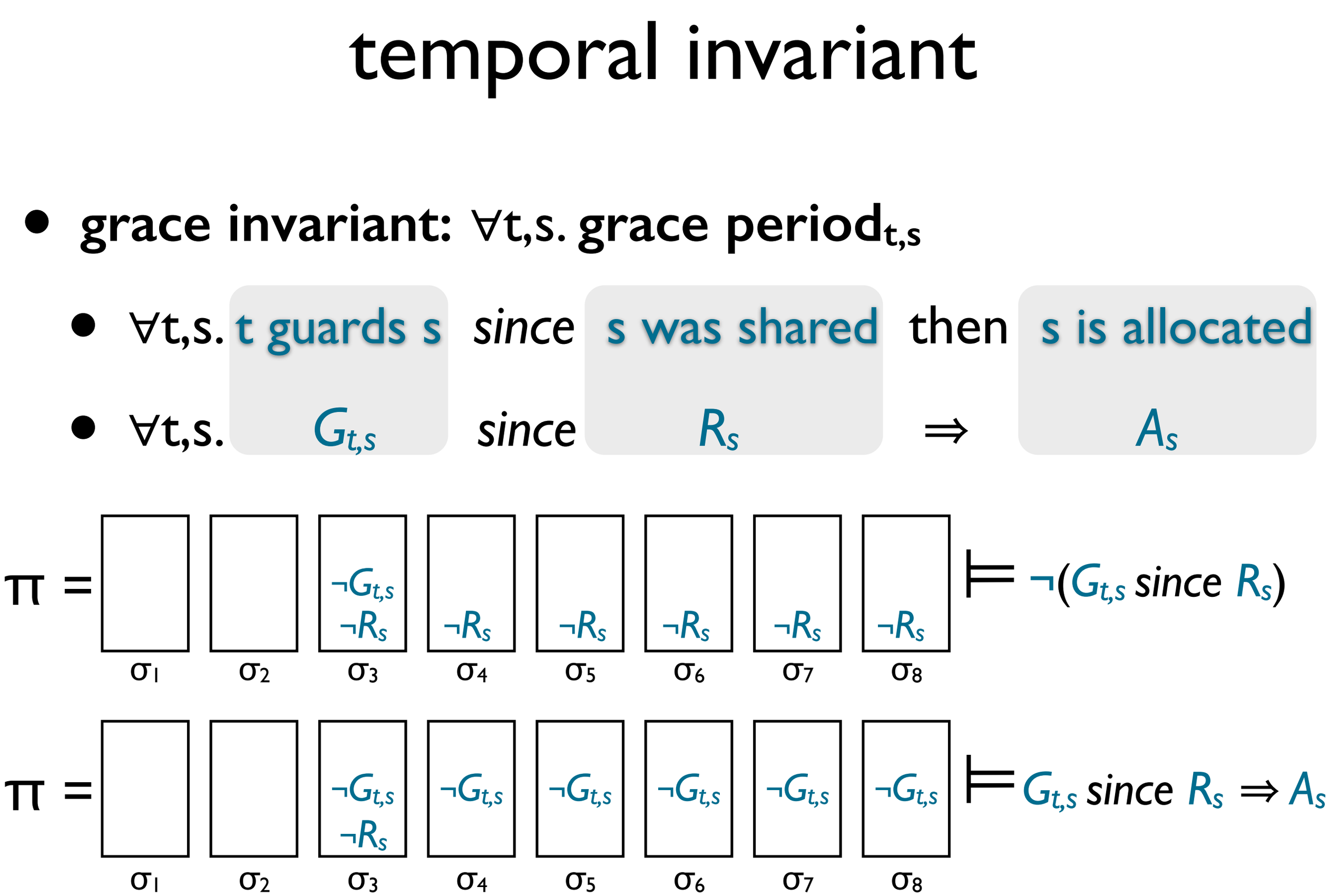
- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s



tautology: $(\neg R_s \text{ since } \neg G_{t,s}) \Rightarrow \neg(G_{t,s} \text{ since } R_s)$

temporal invariant

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

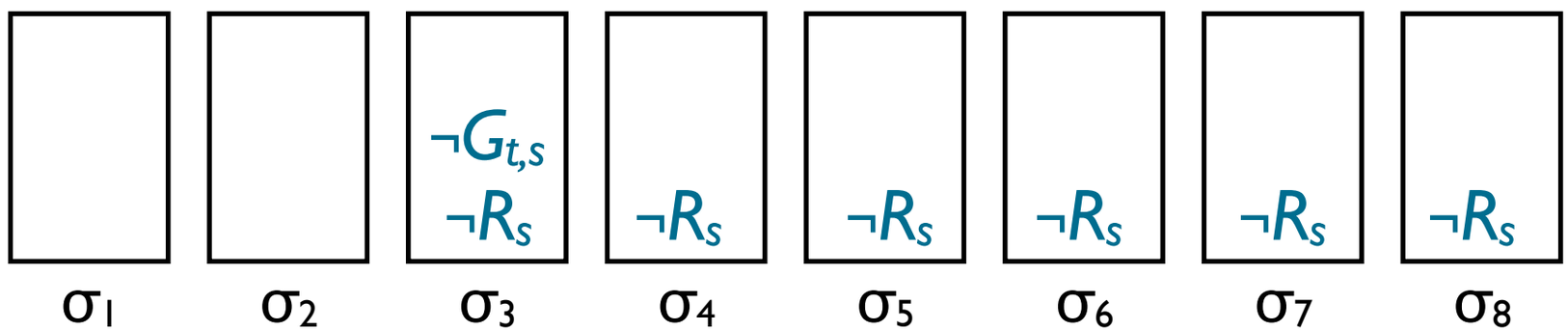


temporal invariant

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

$\pi =$ 

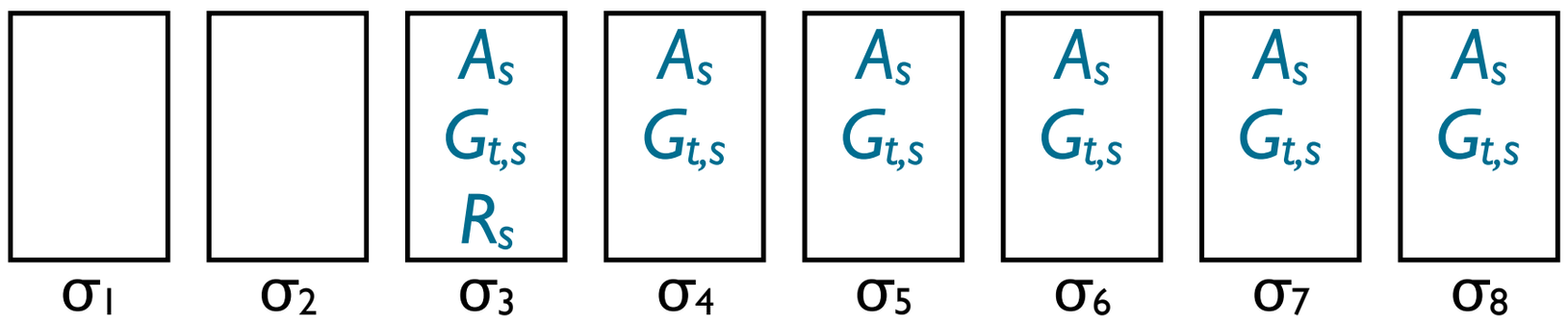
$\models G_{t,s}$ *since* $R_s \Rightarrow A_s$

$\pi =$ 

$\models G_{t,s}$ *since* $R_s \Rightarrow A_s$

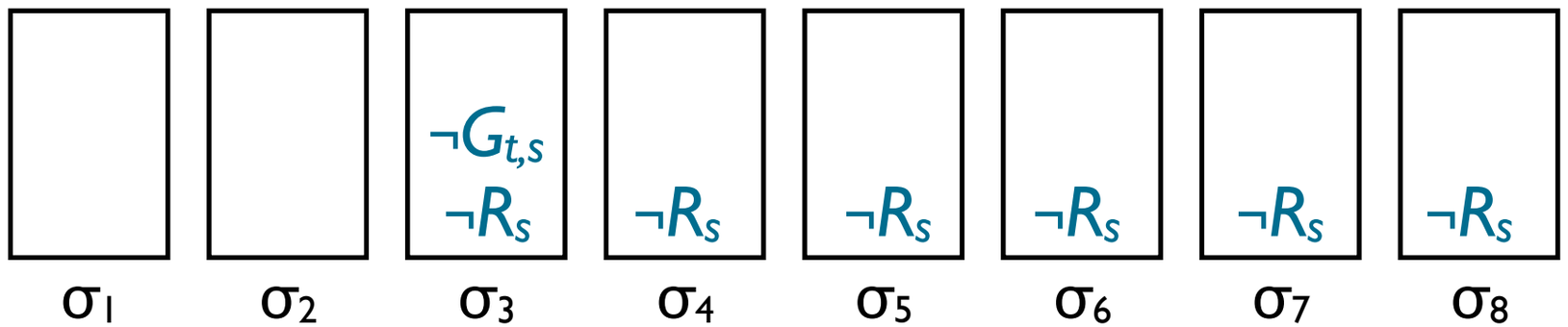
temporal invariant

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

$\pi =$ 

$\models G_{t,s}$ *since* $R_s \Rightarrow A_s$

$(G_{t,s}$ *since* $R_s)$

$\pi =$ 

$\models G_{t,s}$ *since* $R_s \Rightarrow A_s$

$\neg(G_{t,s}$ *since* $R_s)$

grace-based synchronization

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

➡ ***t*** access ***s***: $G_{t,s}$ *since* R_s

➡ ***t*** reclaim ***s***: $\forall t'.\neg R_s$ *since* $\neg G_{t',s}$

grace-based synchronization

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

➡ *t* access *s*: $G_{t,s}$ *since* R_s ($\Rightarrow$ s is allocated)

➡ *t* reclaim *s*: $\forall t'.\neg R_s$ *since* $\neg G_{t,s}$ ($\Rightarrow$ s can be deallocated)

★tautology: $(\neg R_s$ *since* $\neg G_{t,s}) \Rightarrow \neg(G_{t,s}$ *since* $R_s)$

reclamation algorithms (paradigms)

- **Hazard Pointers** [Michael'02]
- **RCU** [McKenney Slingwine'98]
- **Epoch** [Fraser Haris'03]

- pattern of temporal synchronization invariant
  - grace periods
- same idiom for invariant preservation
  - $(\neg R_s \ \mathit{since}\ \neg G_{t,s}) \Rightarrow \neg(G_{t,s}\ \mathit{since}\ R_s)$

reclamation algorithms (paradigms)

- **Hazard Pointers** [Michael'02]
- **RCU** [McKenney Slingwine'98]
- **Epoch** [Fraser Haris'03]

- pattern of temporal synchronization invariant
  - grace periods
- same idiom for invariant preservation
  - $(\neg R_s \ \mathit{since}\ \neg G_{t,s}) \Rightarrow \neg(G_{t,s}\ \mathit{since}\ R_s)$

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;



  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);

 return v;
}
```

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int inc() {
 int v, *s, *n;
 n = new int;
 do{
  s = C;
  v = *s;
  *n = v + 1;
 }while(!CAS(&C,s,n));
 free(s);
 return v;
}
```

```
int *C;
int *HP[TNUM];
```

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

guard s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

guard s1

is s1 shared?
(C=s1?)

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
    v = *s1;
    *n = v + 1;
  }while(!CAS(&C,s1,n));
  HP[tid] = null;
  reclaim(s1);
  return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

guard s1

is s1 shared?
(C=s1?)

use s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
   v = *s1;
   *n = v + 1;
  }while(!CAS(&C,s1,n));
  HP[tid] = null;
  reclaim(s1);
  return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
 }while(HP[i]== s);

 i = i + 1;
}

free(s);
}
```

guard s1

is s1 shared?
(C=s1?)

use s1

unguard s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

assuming C ≠ s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

is s1 shared? (C=s1?)

use s1

unguard s1

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

assuming C ≠ s

t_i does not guard s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

is s1 shared? (C=s1?)

use s1

unguard s1

partial ownership

- proof hinges on C ≠ s during retire(s)

- ensured by ownership annotations
  - the thread which removes s can
    - C = s
    - free (s)
  - other threads can only access s

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

$\forall t,s.\ (HP[t]=s\ \textit{since}\ C=s) \Rightarrow allocated(s)$

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

HP[tid]=s1
∧ C=s2

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

HP[tid]=s1
since C=s2

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

HP[tid]=s1 *since* C=s2

HP[tid]=s1 *since* C=s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
   }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

HP[tid]=s1 *since* C=s2

HP[tid]=s1 *since* C=s1

s1 is allocated *(memory safety)*

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
    s1 = C;
    HP[tid] = s1;
    s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

HP[tid]=s1

HP[tid]=s1 *since* C=s2

HP[tid]=s1 *since* C=s1

s1 is allocated *(memory safety)*

s1 is allocated since C=s *(no ABA)*

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
    s1 = C;
    HP[tid] = s1;
    s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified)

counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

$\forall t,s.\ (HP[t]=s\ \textit{since}\ C=s) \Rightarrow allocated(s)$

```
reclaim(s) {
 i = 0;
 do{
   skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
      s1 = C;
      HP[tid] = s1;
      s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

counter with hazard pointers *(simplified)*

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C≠s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C≠s

∀t<i.C≠s *since* HP[t] ≠s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C≠s

∀t<i.C≠s *since* HP[t] ≠s

∀t.C≠s *since* HP[t] ≠s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

(simplified) counter with hazard pointers

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

∀t,s. (HP[t]=s *since* C=s)⇒allocated(s)

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C≠s

∀t<i.C≠s *since* HP[t] ≠s

∀t.C≠s *since* HP[t] ≠s

∀t.(C≠s since HP[t]≠s)⇒¬(HP[t]=s since C=s)

(Invariant is preserved)

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

grace-based synchronization

- **grace invariant:** $\forall t,s.$ **grace period**$_{t,s}$
  - $\forall t,s.$ t guards s *since* s was shared then s is allocated
  - $\forall t,s.$ $G_{t,s}$ *since* R_s $\Rightarrow$ A_s

➡ *t* access *s*:

$G_{t,s}$	*since*	R_s
HP[t]=s	*since*	C=s

➡ t_1 reclaim *s*:

$\forall t.$	$\neg R_s$	*since*	$\neg G_{t,s}$
$\forall t.$	C≠s	*since*	HP[t]≠s

★tautology: $(\neg R_s \text{ since } \neg G_{t,s}) \Rightarrow \neg(G_{t,s} \text{ since } R_s)$

temporal separation logic

$$R, G, I \vdash \{P\}\ C\ \{Q\}$$

temporal separation logic

$$R, G, I \vdash \{P\}\ C\ \{Q\}$$

$$\sigma = (\boxed{p}, \boxed{s}) \in \Sigma_p \times \Sigma_s$$

$$\omega = (\boxed{p}, \boxed{s}\boxed{s}\boxed{s}) \in \Sigma_p \times \Sigma_s^*$$

temporal separation logic

$$R, G, I \vdash \{P\}\ C\ \{Q\}$$

$$\sigma = (p, s) \in \Sigma_p \times \Sigma_s$$

$$\omega = (p, s\,s\,s) \in \Sigma_p \times \Sigma_s^*$$

$$P, Q = \{(p, s\,s\,s), \ldots\} \subseteq \Sigma_p \times \Sigma_s^*$$

$$I = \{s\,s\,s, \ldots\} \subseteq \Sigma_s^*$$

$$R_t, G_t = \{(s, s), \ldots\} \subseteq \Sigma_s \times \Sigma_s$$

temporal separation logic

temporal logic, separation logic, rely/guarantee, local actions, stability, temporal assertions, spatial resource invariants, local/shared state partitioning, permissions, temporal invariant

$$R, G, I \vdash \{P\}\ C\ \{Q\}$$

$$\sigma = (\boxed{p}, \boxed{s}) \in \Sigma_p \times \Sigma_s$$

$$\omega = (\boxed{p}, \boxed{s}\boxed{s}\boxed{s}) \in \Sigma_p \times \Sigma_s^*$$

$$P, Q = \{(\boxed{p}, \boxed{s}\boxed{s}\boxed{s}), \dots\} \subseteq \Sigma_p \times \Sigma_s^*$$

$$I = \{\boxed{s}\boxed{s}\boxed{s}, \dots\} \subseteq \Sigma_s^*$$

$$R_t, G_t = \{(\boxed{s}, \boxed{s}), \dots\} \subseteq \Sigma_s \times \Sigma_s$$

proof (retire)

1 $\{V \Vdash p \mapsto_m \_ * F_{\mathtt{tid}} \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \wedge \boxed{p \mapsto_e \_ * \mathsf{true}}\}$
2 `void retire(int* p) {`
3 $\quad\{V \Vdash p \mapsto_m \_ * \exists A.\, \mathtt{detached} \mapsto A * D(A) \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \wedge \boxed{p \mapsto_e \_ * \mathsf{true}}\}$
4 `insert(detached,p);`
5 $\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
6 `  if (nondet())`
7 $\quad\quad\{V \Vdash F_{\mathtt{tid}} \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
8 `    return;`
9 `  Set used;`
10 $\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) \wedge \mathit{used} = \emptyset \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
11 `  while (!isEmpty(detached)) {`
12 $\quad\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) \wedge A \neq \emptyset \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
13 `    bool my = true;`
14 `    Node *n = pop(detached);`
15 $\quad\quad\{V \Vdash \mathit{my} \wedge \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) * n \mapsto_m \_ * \boxed{n \mapsto_e \_ * \mathsf{true}} * \boxed{H * \exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
16 `    for (int i = 0; i < N && my; i++) {`
17 $\quad\quad\quad\{V \Vdash \mathit{my} \wedge 0 \leq i < \mathtt{N} \wedge \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) * n \mapsto_m \_ * \boxed{n \mapsto_e \_ * \mathsf{true}} *$
18 $\quad\quad\quad\boxed{H * \exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \wedge \forall 0 \leq j < i.\, \boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}$ since $\boxed{\mathtt{HP}[i] \neq n * \mathsf{true}}\}$
19 `      if (⟨HP[i] == n⟩Id)`
20 `        my = false;`
21 $\quad\quad\quad\{V \Vdash 0 \leq i < \mathtt{N} \wedge \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) * n \mapsto_m \_ * \boxed{n \mapsto_e \_ * \mathsf{true}} *$
22 $\quad\quad\quad\boxed{H * \exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \wedge (\mathit{my} \Rightarrow \forall 0 \leq j \leq i.\, \boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}$ since $\boxed{\mathtt{HP}[j] \neq n * \mathsf{true}})\}$
23 `    }`
24 $\quad\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) * n \mapsto_m \_ * \boxed{n \mapsto_e \_ * \mathsf{true}} \wedge$
25 $\quad\quad\boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}} \wedge (\mathit{my} \Rightarrow \forall 0 \leq j \leq \mathtt{N}.\, \boxed{\exists y.\, y \neq n \wedge \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}$ since $\boxed{\mathtt{HP}[j] \neq n * \mathsf{true}})\}$
26 `    if (my) {`
27 `      ⟨ ; ⟩Take;`
28 $\quad\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) * n \mapsto \_ * \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
29 `      free(n);`
30 `    } else {`
31 `      insert(used, n);`
32 `    }`
33 $\quad\quad\{V \Vdash \exists A.\, \mathtt{detached} \mapsto A * D(A) * D(\mathit{used}) \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
34 `  }`
35 $\quad\{V \Vdash \mathtt{detached} \mapsto \emptyset * D(\mathit{used}) \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
36 `  moveAll(detached, used);`
37 $\quad\{V \Vdash F_{\mathtt{tid}} \wedge \boxed{H * \exists y.\, \mathtt{C} \mapsto y * y \mapsto \_ * \mathsf{true}}\}$
38 `}`

summary

- **Hazard Pointers** [Michael'02]
  - non-blocking reclaim, single reading of hazard array, dynamic allocation of hazard pointers, non-blocking stack, reuse of next-pointer
- **RCU** [McKenney Slingwine'98]
  - first formalization
- **Epoch** [Fraser Haris'03]

★ invariant: $\forall t,s.\ G_{t,s}\ \mathit{since}\ R_s \Rightarrow A_s$

★ tautology: $(\neg R_s\ \mathit{since}\ \neg G_{t,s}) \Rightarrow \neg(G_{t,s}\ \mathit{since}\ R_s)$

related work

- separation logic [Reynolds, LICS'02] [O'Hearn+ POPL'01]
- CSL [O'Hearn, TCS'07]
- Stack + HP in CSL [Parkinson+, POPL'07]
- reductions [Elmas+, POPL'09]
- R/G Separation Logic [Vefiadis, PhD.'08]
- temporal separation logic [Fu+, CONCUR'10]
- interval temporal logic [Tofan+, ICTAC'11]

future work

- **Pass the Buck** [Herlihy+, DISC'02]
- weak memory

conclusions

- *since* operator as abstraction of histories
  - natural specification
  - complicates logic

- programming patterns ⇒ proof patterns
  - simplify proofs using specialized rules

thank you!

“weirdnesses”

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
   skip
 }while(HP[i]== s);

 i = i + 1;
}

free(s);
}
```

C ≠ s

t_i does not guard s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
   v = *s1;
   *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

is s1 shared? (C=s1?)

use s1

unguard s1

“weirdnesses”

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C ≠ s

t_i does not guard s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
    s1 = C;
    HP[tid] = s1;
    s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

s1 might get deallocated

is s1 shared? (C=s1?)

use s1

unguard s1

“weirdnesses”

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C ≠ s

t_i does not guard s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
    s1 = C;
    HP[tid] = s1;
    s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

s1 might get deallocated

is s1 shared? (C=s1?)

use s1

unguard s1

“weirdnesses”

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C ≠ s

t_i does not guard s

t_i might guard s

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
     s1 = C;
     HP[tid] = s1;
     s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1;
 }while(!CAS(&C,s1,n));
 HP[tid] = null;
 reclaim(s1);
 return v;
}
```

guard s1

is s1 shared? (C=s1?)

use s1

unguard s1

s1 might get deallocated

“weirdnesses”

```
int *C;
```

```
int *C;
int *HP[TNUM];
```

```
reclaim(s) {
 i = 0;
 do{
  do{
    skip
  }while(HP[i]== s);

  i = i + 1;
 }

 free(s);
}
```

C ≠ s

t_i does not guard s

t_i might guard s

guard s1

is s1 shared? (C=s1?)

use s1

unguard s1

```
int inc() {
 int v,*s1,*s2,*n;
 n = new int;
 do{
  do{
    s1 = C;
    HP[tid] = s1;
    s2 = C;
  }while(s1 != s2);
  v = *s1;
  *n = v + 1
 }while(!CA
 HP[tid] =
 reclaim(s1);
 return v;
}
```

s1 might get deallocated

s1 reallocated ⇒ “reguarded”