

CS766: Analysis of concurrent programs 2022

Lecture 2: Traces

Instructor: Ashutosh Gupta

IITB, India

Compile date: 2022-02-18

Topic 2.1

Formal model of execution

Introducing parallel composition

We add parallel composition in our simple programming language.

$$c \parallel c$$

Each c is called a **thread**.

We define interleaved semantics as follows

$$((v, c_1 \parallel c_2), (v', c_1' \parallel c_2)) \in T \quad \text{if} \quad ((v, c_1), (v', c_1')) \in T$$

$$((v, c_1 \parallel c_2), (v', c_1 \parallel c_2')) \in T \quad \text{if} \quad ((v, c_2), (v', c_2')) \in T$$

$$((v, \text{skip} \parallel \text{skip}), (v, \text{skip})) \in T$$

Interleaved semantics

Global variables

Definition 2.1

*We call a variable **global** if two or more threads access the variable. Other variables are called local.*

Topic 2.2

Understanding time in interleaved semantics

Instantaneous events

Events happen in threads

- ▶ are instantaneous
- ▶ No simultaneous events

Global instantaneous events

On global variables

- ▶ Writes
- ▶ Reads
- ▶ Read/Write

Synchronization

- ▶ fences

We write them as follows

$$R/W\langle\text{VariableName}\rangle = \langle\text{Value}\rangle \mid \text{fence}$$

Events on a timeline

```
init: counter := 0
```

```
thread0:          thread1:  
a1:tmp0 := counter || b1:tmp1 := counter  
a2:counter = tmp0 + 1  b2:counter = tmp1 + 1
```

Global event order in a possible execution:

```
thread0: Rcounter=1  
thread1: Rcounter=1  
thread1: Wcounter=2  
thread0: Wcounter=2
```

Time is always **totally ordered**.

Intervals

- ▶ Often things are not instantaneous
- ▶ Takes time to finish such as function calls.

Definition 2.2

An interval

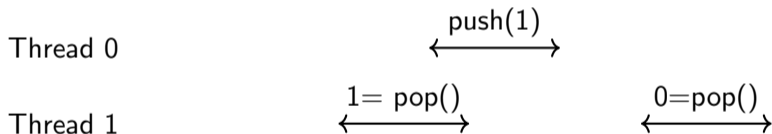
$$A = (e_1, e_2)$$

is time between events e_1 and e_2 .

Intervals may or may not overlap

Example 2.1

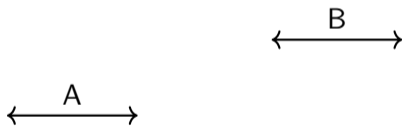
Operations on stack:



Happens before relation (hb)

Definition 2.3

If A ends before A starts, we say A *happens before* B .



Strict partial order

Theorem 2.1

hb relation is a strict partial order.

Proof.

hb has the following properties of strict partial order.

- ▶ Antisymmetry: $hb(A, B) \Rightarrow \neg hb(B, A)$
- ▶ Transitivity: $hb(A, B) \wedge hb(B, C) \Rightarrow hb(A, C)$
- ▶ Irreflexive: $\neg hb(A, A)$



Exercise 2.1

Which condition to add in partial orders conditions such that it becomes total order?

Orders that we know: program order(po)

po : events in a thread are ordered.

$Wx = 0$



$Ry = 0$

Orders that we know: read from(**rf**)

rf : every read reads from exactly one write

$$W_x = 0$$



$$R_x = 1$$

Orders that we know: write serialization(**ws**)

ws : all writes on a global are totally ordered

$Wx = 0$



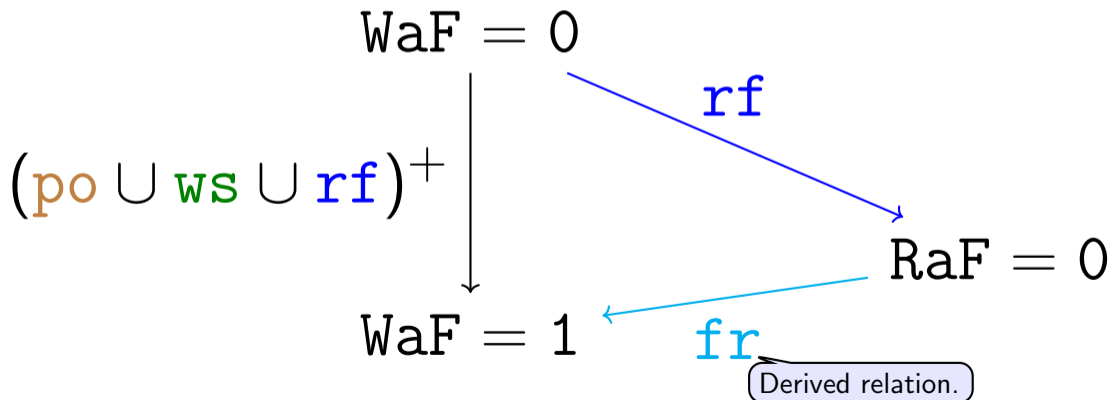
ws

Not directly
observed

$Wx = 1$

Orders that we know: from read(fr)

fr : no write comes between the pair in **rf**



Traces

Definition 2.4

A *trace* consists of the following relations.

- ▶ **po** : events in a thread are ordered.
- ▶ **rf** : every read reads from exactly one write
- ▶ **ws** : all writes on a global are totally ordered
- ▶ **fr** : no other write comes between the read write pairs in rf

Definition 2.5

A trace is valid if $(\text{po} \cup \text{rf} \cup \text{ws} \cup \text{fr})^+$ is strict partial order.

Theorem 2.2

An execution of a program induces only a valid trace.

Analyzing programs using traces

For analyzing a program,

1. We **enumerate** all (possibly infinite) execution paths in each thread.
2. Each path has a set of events and defines **po**.
3. We **enumerate all rf** by mapping each read to some write if values match.
4. We **enumerate all ws** among writes.
5. Using **po**, **rf**, and **ws**, we compute **fr** and check if trace is valid.

If no trace reaches a bad state, then program is safe.

Topic 2.3

Formal properties of mutual exclusion protocols

Structure of mutual exclusion protocol

In a typical mutual exclusion protocol, we have the following structure

- ▶ Global initialization
- ▶ Some code before critical section, called lock
- ▶ critical section
- ▶ Some code after critical section, called unlock

```
init: counter := 0
```

```
while(true){
10: lock()
c0: tmp0 := counter
    counter := tmp0 + 1
u0: unlock()
} ||
while(true){
11: lock()
c1: tmp1 := counter
    counter := tmp1 + 1
u1: unlock()
}
```

Properties: mutual exclusion

```
init: counter := 0
```

```
while(true){
l0: lock()
c0: tmp0 := counter
    counter := tmp0 + 1
u0: unlock()
}

||

while(true){
l1: lock()
c1: tmp1 := counter
    counter := tmp1 + 1
u1: unlock()
}
```

Let i th occurrence of critical section in thread 0 be interval $CS0^i = (c0, u0)$.

For all i and j , $hb(CS0^i, CS1^j) \vee hb(CS1^j, CS0^i)$

Properties: deadlock free

```
init: counter := 0

thread0:          thread1:
  while(true){   while(true){
l0:  lock()      l1:  lock()
c0:  tmp0 := counter    || c1:  tmp1 := counter
    counter := tmp0 + 1    counter := tmp1 + 1
u0:  unlock()      u1:  unlock()
  }                }
```

If thread0 visits l0 and never reaches u0,
then thread1 visits u1 infinitely often.

And also symmetric condition. Assumes fair scheduling.

Properties: starvation free

```
init: counter := 0
```

```
thread0:                                thread1:
  while(true){                            while(true){
l0:  lock()                                l1:  lock()
c0:  tmp0 := counter                       || c1:  tmp1 := counter
     counter := tmp0 + 1                   counter := tmp1 + 1
u0:  unlock()                              u1:  unlock()
  }                                        }
```

If thread0 visits l0, it eventually reaches u0.

And also symmetric condition. Assumes fair scheduling for the Precise statements

Topic 2.4

Analyzing mutual exclusion protocols

Example: Flag

```
init: f0 := false, f1 := false
```

```
thread0:
```

```
  while(true){
```

```
l0:  f0 := true
```

```
    while(f1);
```

```
c0:  // critical section
```

```
u0:  f0 := false
```

```
  }
```

```
||
```

```
thread1:
```

```
  while(true){
```

```
l1:  f1 := true
```

```
    while(f0);
```

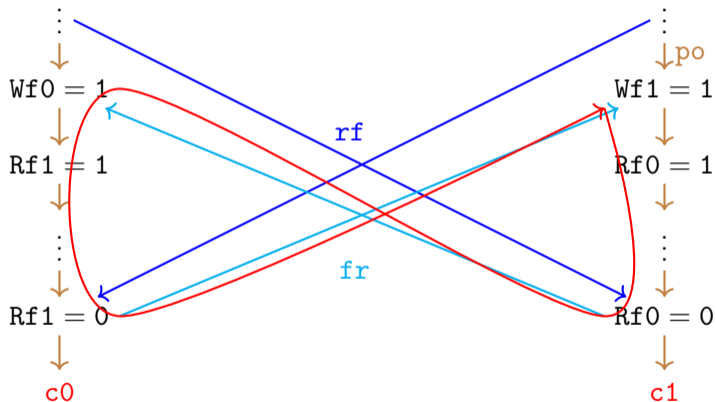
```
c1:  // critical section
```

```
u1:  f1 := false
```

```
  }
```

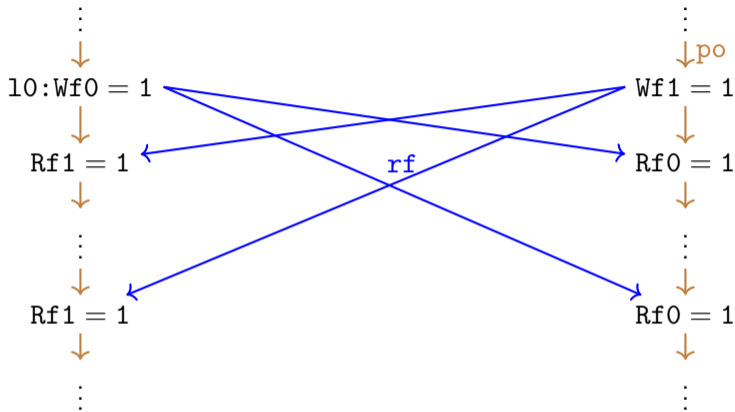
Example: proving mutual exclusion for Flag

Violating execution: Assume threads reached c_0 and c_1 at the same time



The above drawing proves that there will be no violation of mutual exclusion.

Example: Flag can deadlock



Assume never reaches u_0

Does not visit u_1 infinitely often

Trace does not have cycles and does not violate fair scheduling. Therefore, execution possible.

Exercise 2.2 Under our definition, does deadlock \Rightarrow starvation?

Flag: an observation

- ▶ Concurrent execution may deadlock
- ▶ Sequential execution does not

Example: LucknowNawab

```
init: v := random()
```

```
thread0:
```

```
  while(true){
```

```
l0:  v := 0
```

```
  while( v == 0);    ||
```

```
c0:  // critical section
```

```
u0:  skip
```

```
}
```

```
thread1:
```

```
  while(true){
```

```
l1:  v := 1
```

```
  while( v == 1);
```

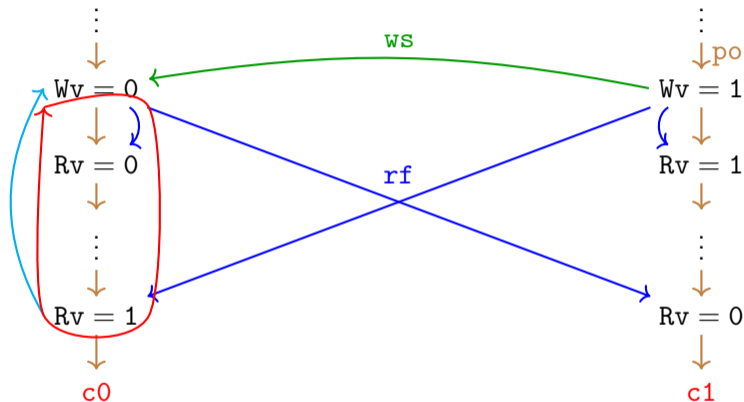
```
c1:  // critical section
```

```
u1:  skip
```

```
}
```

Example: proving mutual exclusion for LucknowNawab

Violating execution: Assume threads reached $c0$ and $c1$ at the same time



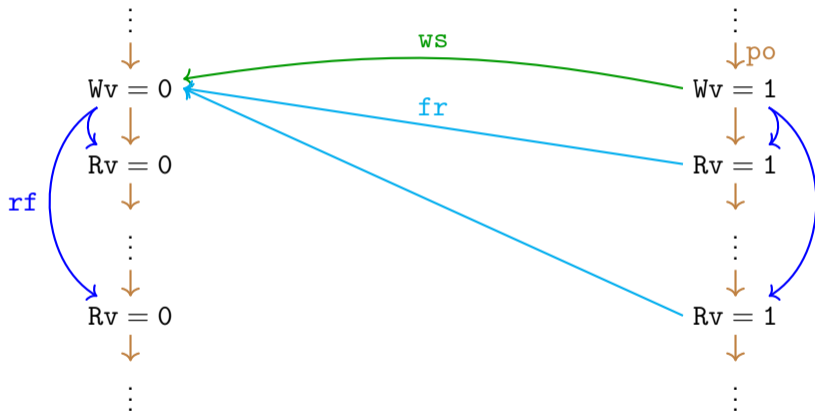
The above drawing proves that there will be no violation of mutual exclusion.

Exercise 2.3

If the last reads, read from some earlier writes, will the drawing continue to work?

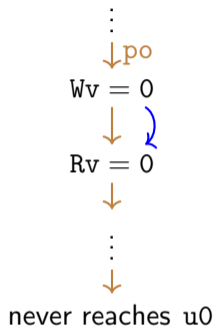
Example: proving deadlock freedom for LucknowNawab

Violating execution: Assume thread 0 and thread 1 are stuck at inner loops.



$Wv = 0$ has to wait for infinite time, which violates fair scheduling assumption. Therefore, we cannot have deadlock.

Example: starvation execution in LucknowNawab



Starvation in sequential execution! Odd!

Exercise 2.4

Does LucknowNawab really starve under the formal definition? Is the fair scheduling assumption real?

Example: Peterson (Nawab meets Flag)

```
init: f0 := false, f1 := false, v := random()
```

```
thread0:                                thread1:
  while(true){                            while(true){
l0:   f0 := true                          l1:   f1 := true
      v := 0                               v := 1
      while(f1 && v==0);                    while(f0 && v==1);
c0:   // critical section                 c1:   // critical section
u0:   f0 := false                          u1:   f1 := false
  }                                        }
```

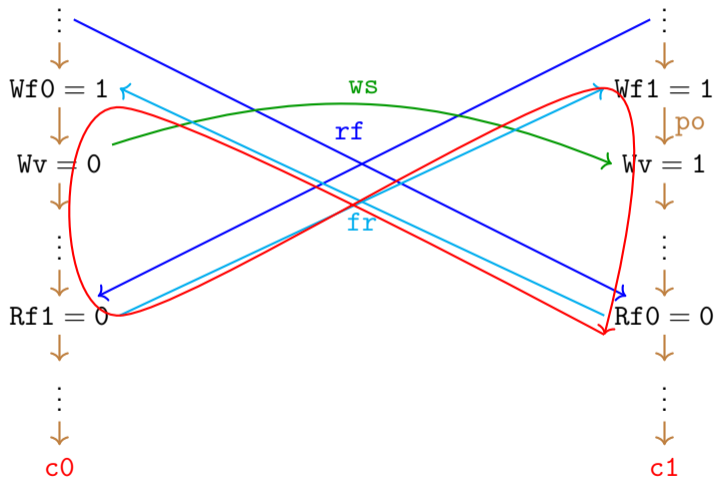
||

The inner loop has two conditions. If any of them fails, the thread enters in the critical section.

There are four (2x2) scenarios that lead to violation.

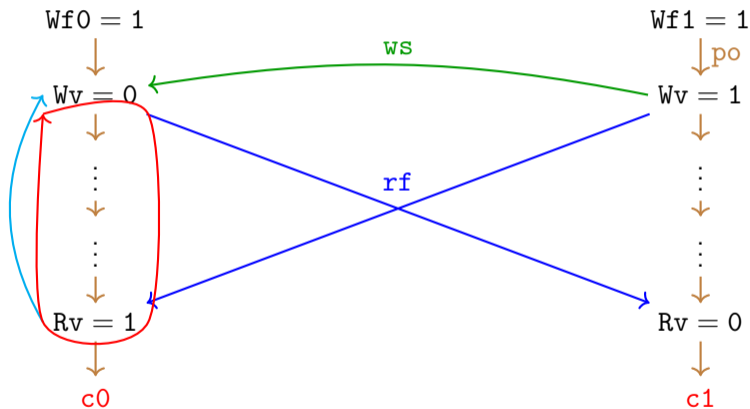
Example: proving mutual exclusion for Peterson (scenario 1: flag proof)

Scenario 1: Assume conditions f_0 and f_1 failed at the same time



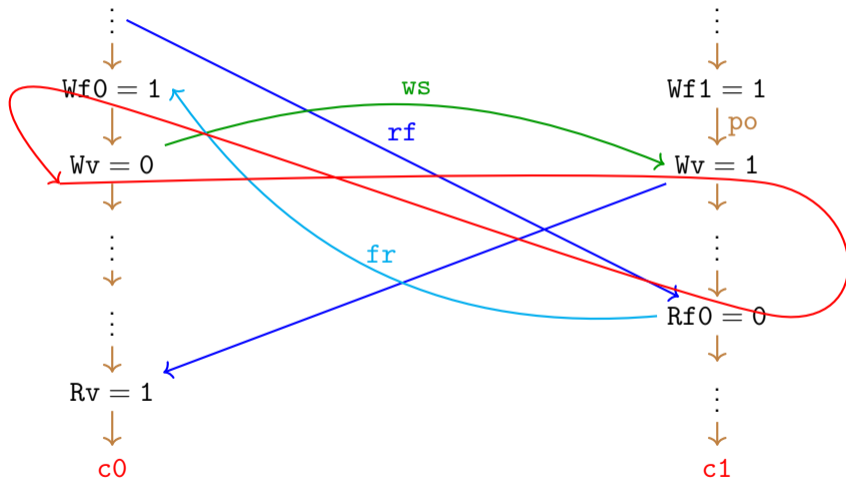
Example: proving mutual exclusion for Peterson (scenario 2: Nawab proof)

Scenario 2: Assume conditions $v == 0$ and $v == 1$ failed at the same time



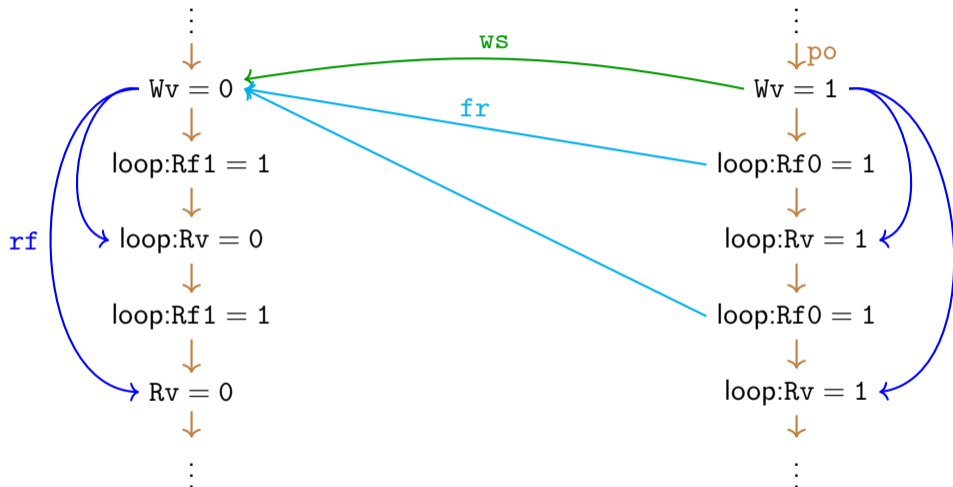
Example: proving mutual exclusion for Peterson (scenario 3/4)

Scenario 3: Assume conditions $v == 0$ and f_0 failed at the same time



The drawings for the four scenarios prove that there will be no violation of mutual exclusion.

Example: proving deadlock freedom for Peterson: Like LucknowNawab



$Wv = 0$ has to wait for infinite time, which violates fair scheduling assumption. Therefore, we cannot have deadlock.

End of Lecture 2