Example Concurrent Program

```
int x = 0
co
  x = x + 1
//
  x = x + 2
oc
print x
```

What are the possible outputs of this program?

Example Concurrent Program (cont.)

- One possible execution order is:
  - Thread 0: R1 := x (R1 == 0)
  - Thread 1: R2 := x (R2 == 0)
  - Thread 1: R2 := R2 + 2 (R2 == 2)
  - Thread 1: x := R2 (x == 2)
  - Thread 0: R1 := R1 + 1 (R1 == 1)
  - Thread 0: x := R1 (x == 1)
- Final value of x is 1 (!!)
- Question: what if Thread 1 also uses R1?

Example Concurrent Program

```
int x = 0
co
  x = x + 1
//
  x = x + 2
oc
print x
```

Possible outputs are 1, 2, and 3

The output **cannot** be 0 because of the oc

More Concurrent Programming: Linked Lists (head is shared)

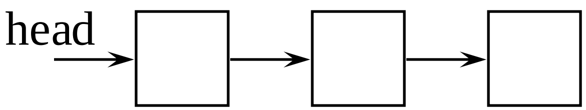
```
Insert(head, elem) {
    elem→next := head;
    head := elem;
}
```

```
Void *Remove(head) {
    Void *t;
    t := head;
    head := head→next;
    return t;
}
```

(Assume one thread calls Insert and one calls Remove, concurrently)

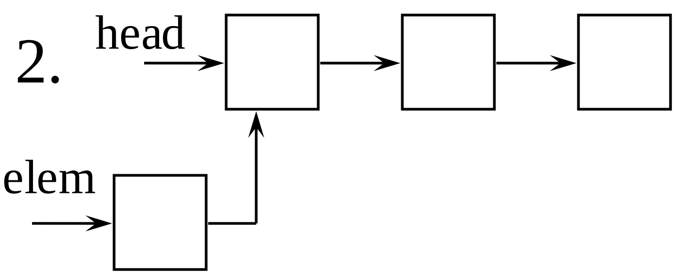
One Possible (Fine) Execution

1.



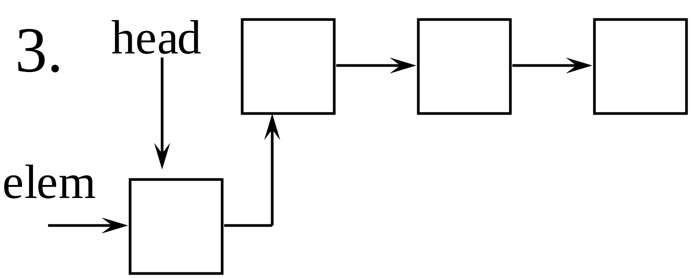
Insert: elem→next := head;

2.



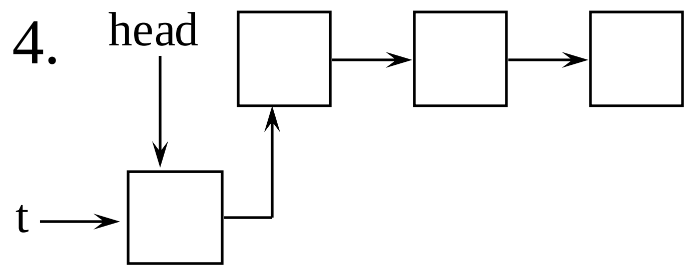
Insert: head := elem;

3.



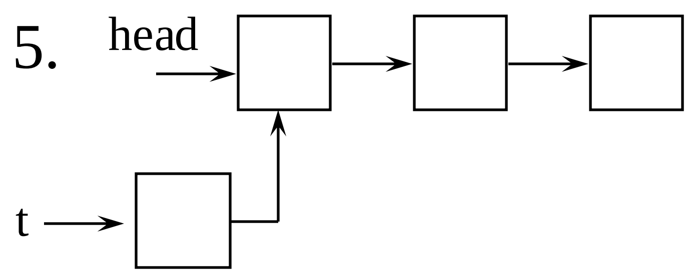
Remove: t := head;

4.



Remove: head := head→next;

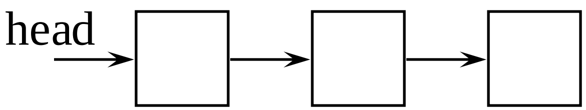
5.



Remove: return t;

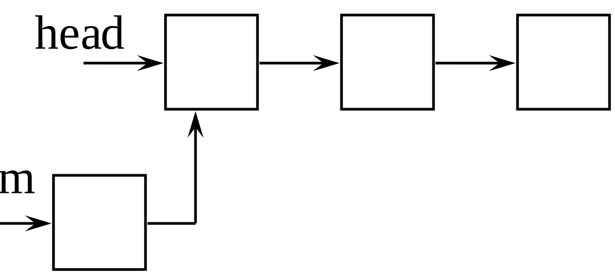
One Possible (Bad!) Execution

1.



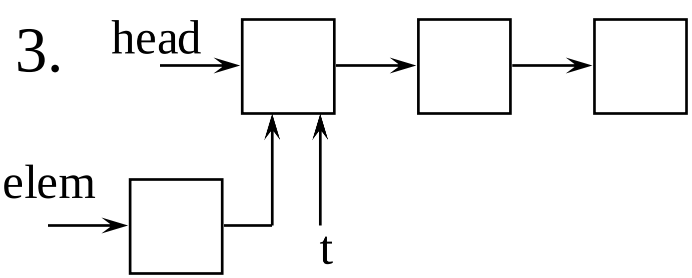
Insert: elem→next := head;

2.



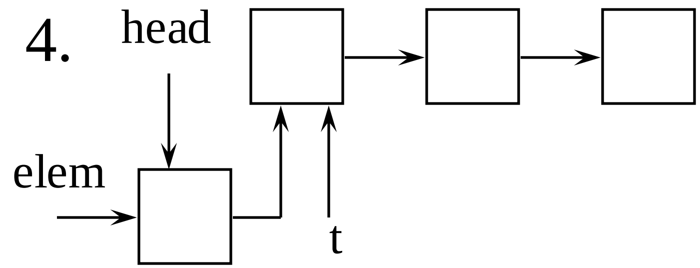
Remove: t := head;

3.



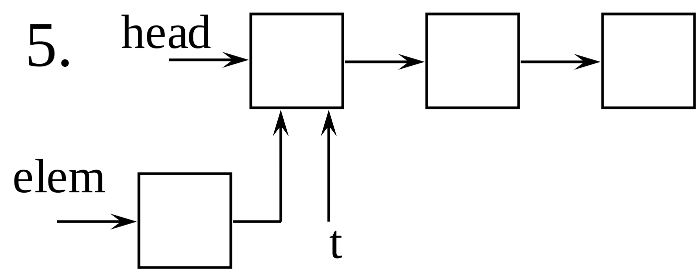
Insert: head := elem;

4.



Remove: head := head→next;

5.



Remove: return t;

Definitions

- Several important terms
  - State
    - The values of all program variables, both implicit and explicit, at a given point in time
  - Atomic action
    - an action that indivisibly examines or changes program state
    - an operation that, once started, runs to completion
      - **more precisely, logically runs to completion**
    - we assume **individual** loads/stores are physically atomic
      - meaning: if thread A stores 1 into variable x and thread B stores 2 into variable x at about the same time, result is either 1 or 2

Definitions, continued

- Additional terms
  - History
    - Linearization (interleaving) of the atomic actions of all threads
      - **Different histories may lead to the same output**
      - **Atomic actions of a particular thread must appear in the linearization in program order**
  - Safety: program never enters a bad state
    - Example: partial correctness
  - Liveness: program eventually enters a good state
    - Example: termination

Definitions, continued

- Additional terms
  - Interference
    - Thread 1 interferes with Thread 2 if:
      - Thread 1 executes an assignment statement that modifies a shared variable that invalidates an assertion in Thread 2

Example of Interference

Assertions are in {…}

int x = 0

co

{x == 0} Assertion: represents state before assignment in thread 1

x = x + 1 Assignment in thread 1

{x == 1} Assertion: represents state after assignment in thread 1

//

{x == 0} Assertion: represents state before assignment in thread 2

Invalidated!

x = x + 2 Assignment in thread 2

{x == 2} Assertion: represents state after assignment in thread 2

oc

Race Condition

- When output depends on ordering of thread execution
- More formally:
  - (1) two or more threads access a shared variable with no synchronization (*or incorrect/insufficient synchronization*), **and**
  - (2) at least one of the threads writes to the variable

Both the addition code and the list code shown previously have race conditions

General Form of Atomic Operation

(Removes Race Conditions)

- <await (B) S> Called a *conditional atomic action*
  - Atomically do (all of) the following:
    - Evaluate B
    - Wait until B is true
    - Execute S (an arbitrary statement list)
  - If the “await (B)” is omitted, S is immediately executed, but still atomically
  - <…> hides intermediate states and reduces number of histories

Example With Await

```
int x = 0
co
  x = x + 1
//
  <(await x == 1) x = x + 2>
oc
print x
```

This program will always output 3.

(It also serializes execution.)

Example with Atomic Operations

```
int x = y = 0, z
co
  <x = 1>; <z = x+y>
//
  <y = 2>; <z = x-y>
oc
```

What are the possible final values of x, y, and z?

How many histories are there?

Example with Atomic Operations

```
int x = y = 0, z
co
  <x = 1>; <z = x+y>
//
  <y = 2>; <z = x-y>
oc
```

Vars x and y must be 1 and 2; z can be -1 or 3

Number of histories is 6

General formula: $(n*m)! / (m!^n)$, where n is number of threads and m is number of atomic actions per thread

Same Example, Removing Explicit Atomicity

```
int x = y = 0, z
co
  x = 1; z = x+y
//
  y = 2; z = x-y
oc
```

What are the possible final values of x, y, and z?

Same Example, Removing Explicit Atomicity

x = k translates to a single Store

z = x + y translates to Load, Load, Add, Store

z = x – y translates to Load, Load, Subtract, Store

int x = y = 0, z

co

  x = 1; z = x+y

//

  y = 2; z = x-y

oc

One key point: Thread 0 can have loaded x and y and Thread 1 can be at any of its instructions. This is not the case in the first scenario, where z = x+y and z = x-y were atomic.

x is 1	R1 is 1	R2 is 0 or 2	R3 is 1 or 3	z is 1 or 3
Thread 0: Store 1, x	Load R1, x	Load R2, y	R3 = R1 + R2	Store R3, z
Thread 1: Store 2, y	Load R4, x	Load R5, y	R6 = R4 – R5	Store R3, z
y is 2	R4 is 0 or 1	R5 is 2	R6 is -2 or -1	z is -2 or -1

Same Example, Removing Explicit Atomicity

```
int x = y = 0, z
co
  x = 1; z = x+y
//
  y = 2; z = x-y
oc
```

As before, x and y must be 1 and 2, but while z can still be -1 or 3 (as before), it can now also be -2 or 1

Note that enumerating all histories here is impractical

Via previous formula: $(10!) / (5!^2) == 252$ histories

(2 threads, 5 atomic actions each)

Scheduling policies for atomic actions

- Unconditional fairness
  - Every unconditional atomic action eventually executes
    - Round robin scheduling satisfies this
- Weak fairness: UC + conditional atomic actions execute if true and seen by the thread
- Strong fairness: UC + conditional atomic actions execute if true infinitely often

Scheduling policies: WF vs. SF

```
continue := true; try := false
co
  while (continue) {try := true ; try := false}
//
  <await (try) continue := false>
oc
```

- With weak fairness, program may never terminate; with strong fairness, it will terminate
  – Practical schedulers, however, are not strongly fair

Finding the max of an array in parallel

Sequential version

```
int max = MINVAL
int a[n]
for i = 0 to n-1 {
   if (a[i] > max)
      max = a[i]
}
```

Finding the max of an array in parallel

Incorrect parallel version

```
int max = MINVAL
int a[n]
co i = 0 to n-1 {
   if (a[i] > max)
      max = a[i]
}
```

Finding the max of an array in parallel

Correct but slow parallel version

```
int max = MINVAL
int a[n]
co i = 0 to n-1 {
   <if (a[i] > max)
      max = a[i]>
}
```

Finding the max of an array in parallel

Another incorrect parallel version

```
int max = MINVAL
int a[n]
co i = 0 to n-1 {
   if (a[i] > max)
      <max = a[i]>
}
```

Finding the max of an array in parallel

Correct, efficient (but complicated) parallel version

```
int max = MINVAL
int a[n]
co i = 0 to n-1 {
  if (a[i] > max) {
    <if (a[i] > max)
      max = a[i]>
  }
}
```

Why do this?