COL380

Introduction to
Parallel & Distributed Programming

OpenMP Threads & Tasks

- Tasks are a software construct
  - ➡ Have execution context: stack, heap, PC..
  - ➡ To be executed by some “device”
- In OpenMP tasks are not directly scheduled on hardware devices
  - ➡ It instead uses the intermediate construct “Thread”
  - ➡ Threads are (can be) scheduled by the OS
- User-space code schedules/assigns tasks to threads

```
#pragma omp parallel
#pragma omp single
#pragma omp task
    processAB(a, b, n);
```

Sharing and Racing

```
void loop(int &a, int n)
{
    for(int i=0; i<n; i += 3)
        a++;
}

..
int a = 0, n = 10;
#pragma omp parallel
    loop(a, n);
```

- Operation are not instantaneous:
- Such an operation can becomes visible to different threads are different times
- apparent order of operations may not be *consistent*
- Even *consistent* operations can be *concurrent* (vary from execution to execution)
- Program must remain correct no matter the order

Data race: Concurrent RW or WW operations

Race condition: If variable order of concurrent operations affects correctness

Share Occasionally

```
void loop(int &a, int n)
{
    for(int i=0; i<n; i += 3)
        a++;
}
..
int a = 0, n = 10;
#pragma omp parallel
    loop(a, n);
```

(Rd a; Add1; Wr a)

```
        movl    (%rdi), %eax
        addl    $1, %eax
        movl    $0, %edx
.L3:    movl    %eax, %ecx
        addl    $3, %edx
        addl    $1, %eax
        cmpl    %edx, %esi
        jg      .L3
        movl    %ecx, (%rdi)
```

Share Occasionally

```
void loop(volatile int &a, int n)
{
    for(int i=0; i<n; i += 3)
        a++;
}
```

(Rd a; Add1; Wr a)

```
..
int a = 0, n = 10;
#pragma omp parallel
    loop(a, n);
```

```
        movl    (%rdi), %eax
        addl    $1, %eax
        movl    $0, %edx
.L3:    movl    %eax, %ecx
        addl    $3, %edx
        addl    $1, %eax
        cmpl    %edx, %esi
        jg      .L3
        movl    %ecx, (%rdi)
```

```
.L3: movl    (%rdi), %eax
     addl    $1, %eax
     movl    %eax, (%rdi)
     addl    $3, %edx
     cmpl    %edx, %esi
     jg      .L3
     ret
```

Share Occasionally

```
void loop(volatile int &a, int n)
{
    for(int i=0; i<n; i += 3)
        a++;
}
..
int a = 0, n = 10;
#pragma omp parallel
    loop(a, n);
```

(Rd a; Add1; Wr a)

```
        movl    (%rdi), %eax
        addl    $1, %eax
        movl    $0, %edx
.L3:    movl    %eax, %ecx
        addl    $3, %edx
        addl    $1, %eax
        cmpl    %edx, %esi
        jg      .L3
        movl    %ecx, (%rdi)
```

```
.L3: movl    (%rdi), %eax
     addl    $1, %eax
     movl    %eax, (%rdi)
     addl    $3, %edx
     cmpl    %edx, %esi
     jg      .L3
     ret
```

var = val need not be "seen"
or, need not be "atomically" seen

Atomic = "Not divisible"
(with respect to a set of operations)

Memory Model

Memory Model

- Register size?
- Limit on the number of readers and writers?
- Asynchronous reads and writes
  - ➡ Global knowledge of ‘time?’

Memory Model

- Register size?
- Limit on the number of readers and writers?
- Asynchronous reads and writes
  - ➡ Global knowledge of ‘time?’

✳ What is the value read? (Correctness)

“Most recent Write”

Memory Model

- Register size?
- Limit on the number of readers and writers?
- Asynchronous reads and writes
  - ➡ Global knowledge of ‘time?’

✳ What is the value read? (Correctness)

- Increasing power ⇒ reducing performance

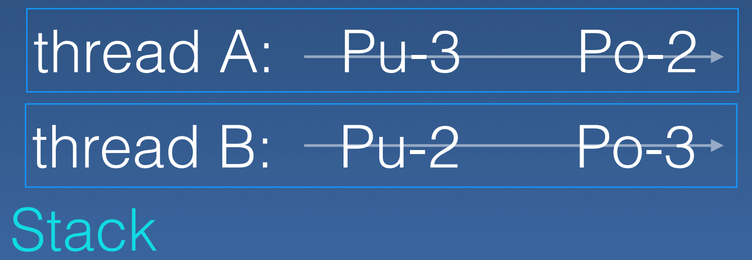
Memory Model

- Register size?
- Limit on the number of readers and writers?
- Asynchronous reads and writes
  - ➡ Global knowledge of ‘time?’

✳ What is the value read? (Correctness)

- Increasing power ⇒ reducing performance

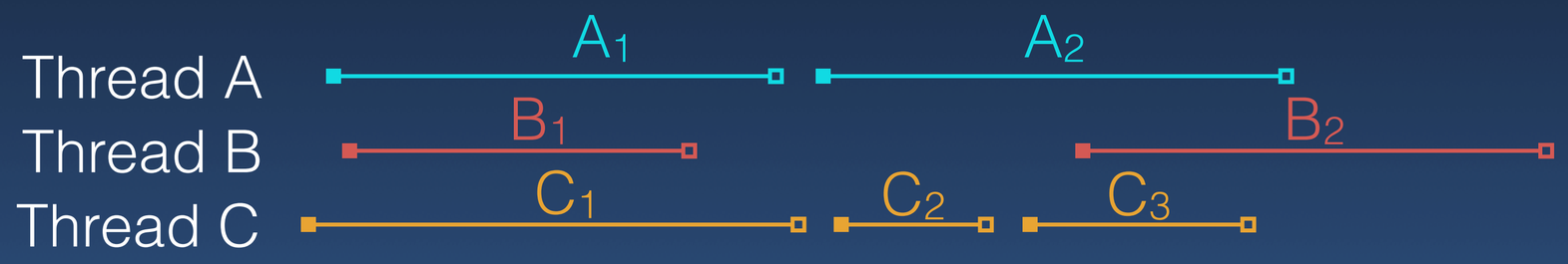
Same ideas apply to higher level data structures



Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

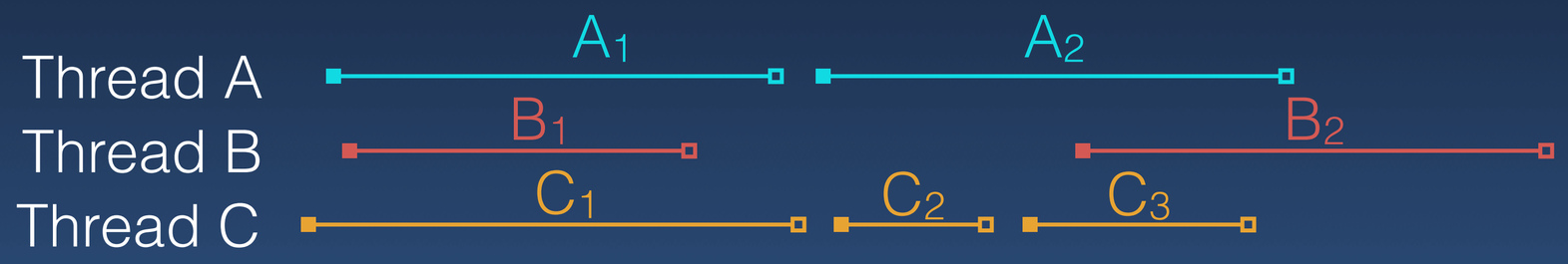
(knowledge of global time)



Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

(knowledge of global time)

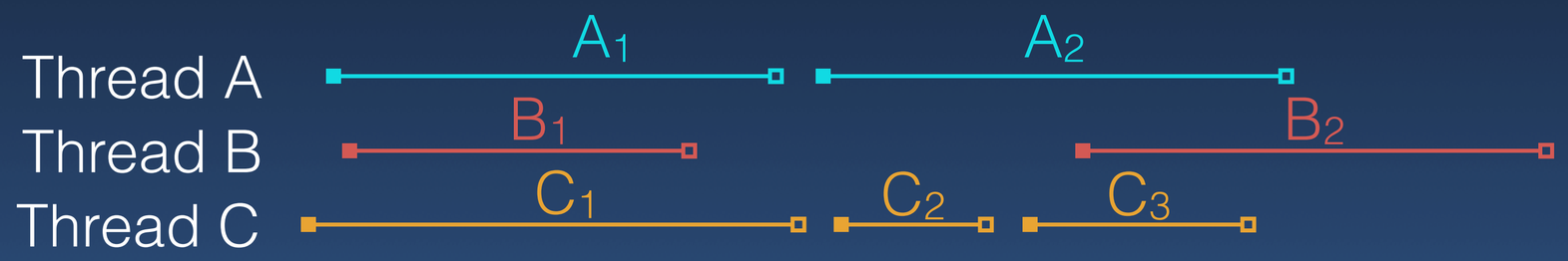


Equivalent sequential history S exists:

Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

(knowledge of global time)



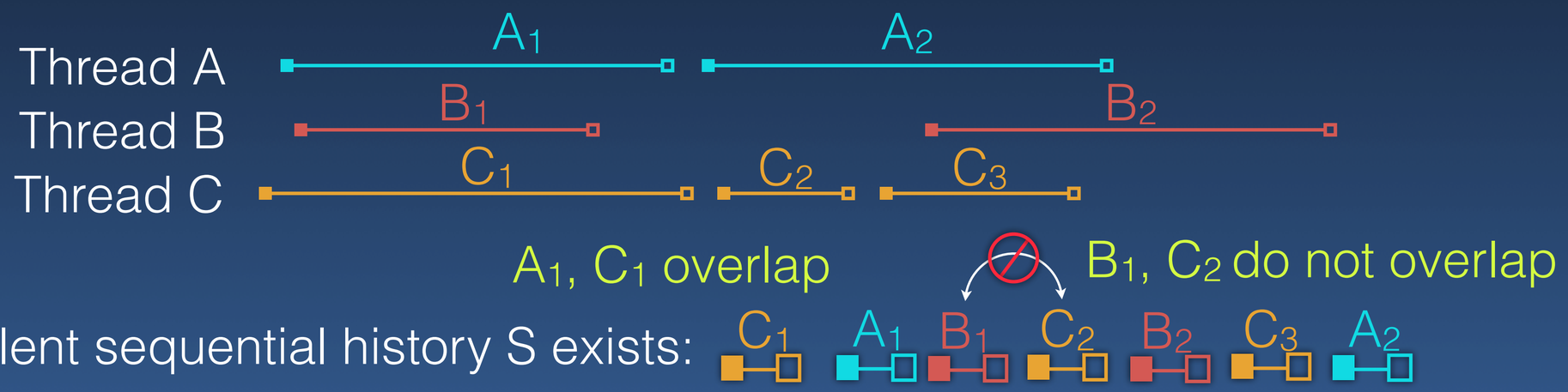
A_1, C_1 overlap

Equivalent sequential history S exists:

Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

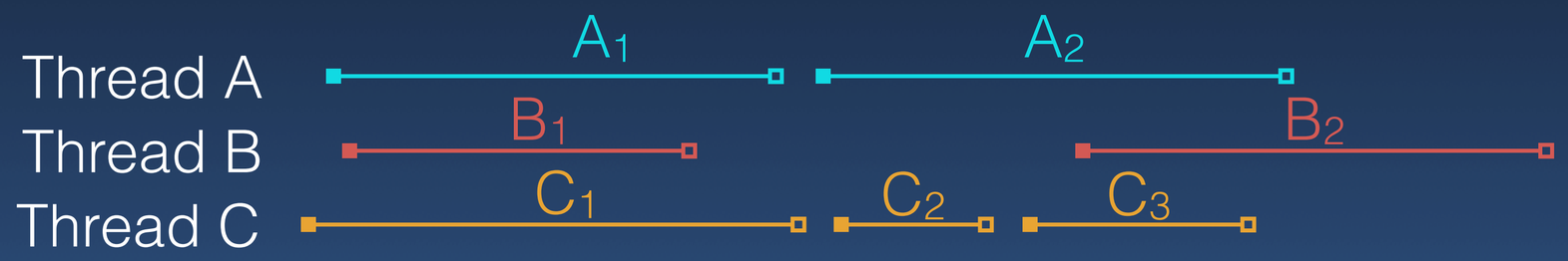
(knowledge of global time)



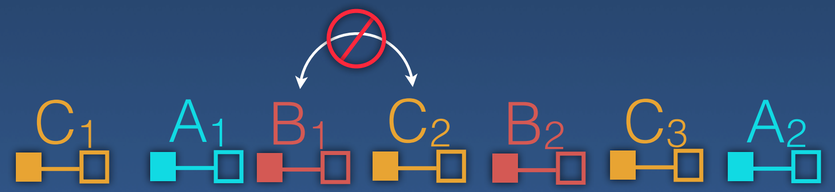
Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

(knowledge of global time)



Equivalent sequential history S exists:

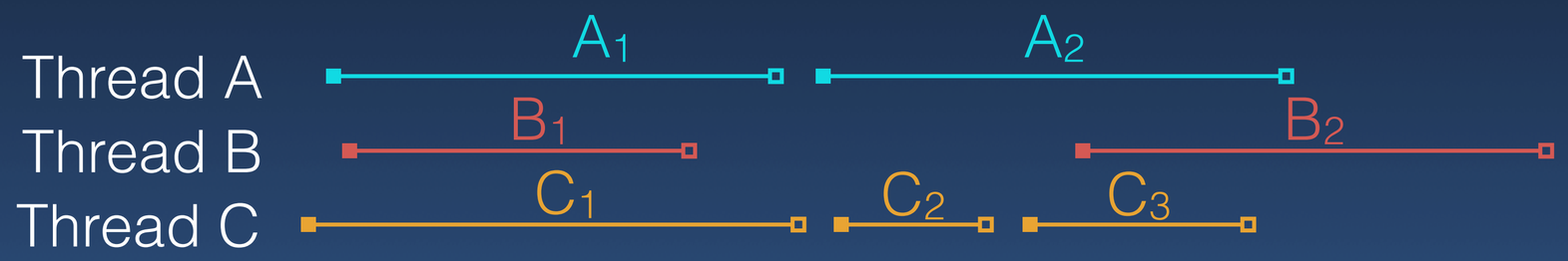


1. Got the result S would get
2. No thread's history is violated in S
3. Non-overlapping operations retain order in S

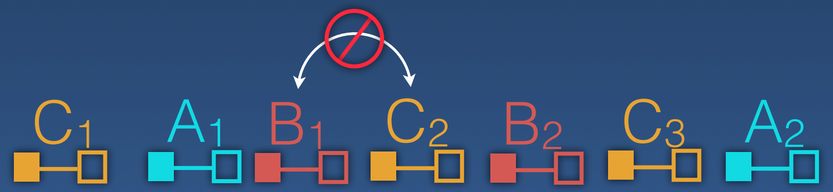
Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

(knowledge of global time)



Equivalent sequential history S exists:



1. Got the result S would get
2. No thread's history is violated in S
3. Non-overlapping operations retain order in S

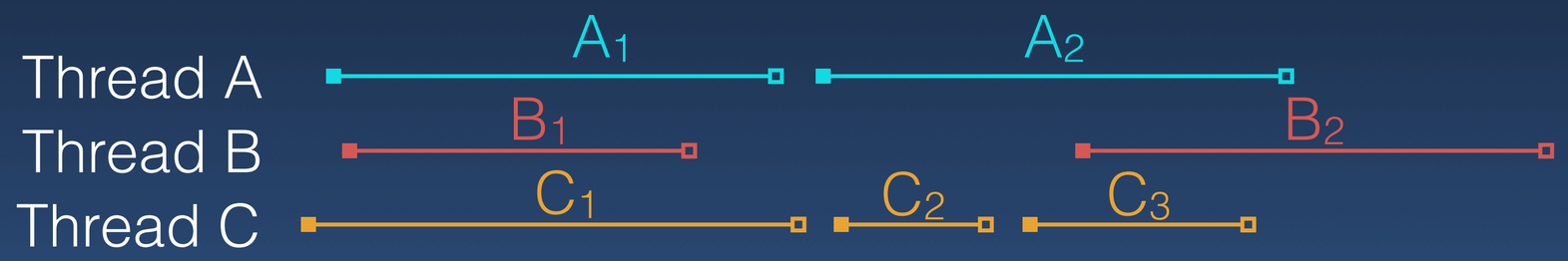
Linearizability is Composable

Subodh Kumar

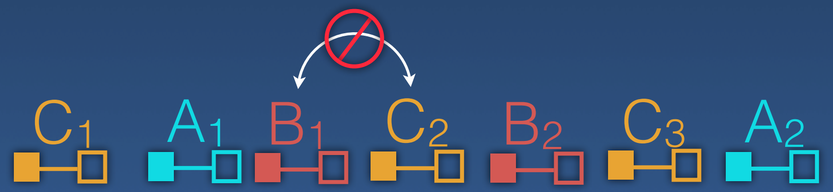
Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined

(knowledge of global time)



Equivalent sequential history S exists:



1. Got the result S would get
2. No thread's history is violated in S
3. Non-overlapping operations retain order in S

Linearizability is Composable

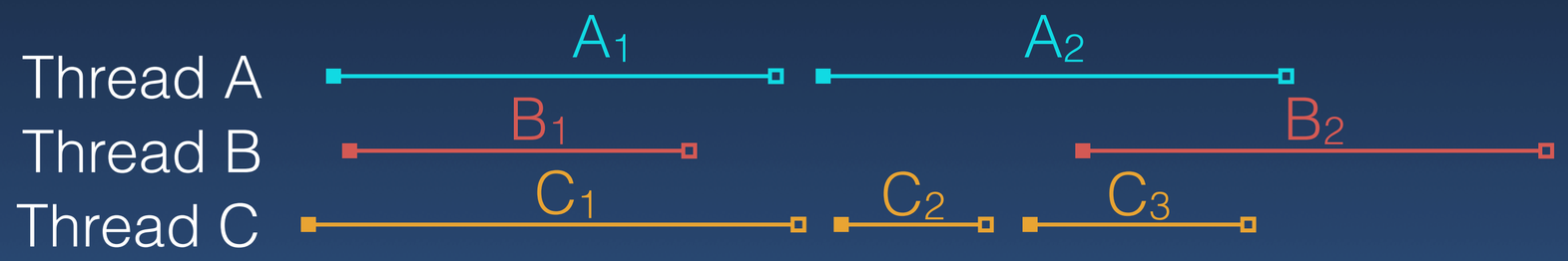
What if we cannot tell what overlaps what

Subodh Kumar

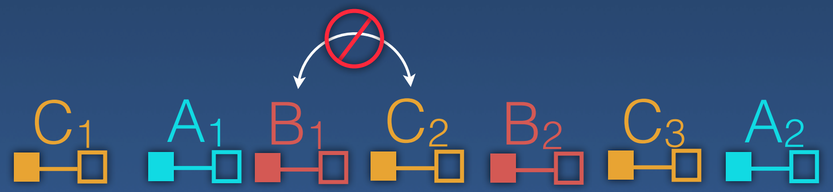
Linearizable

- Operations appear to take global effect at some instant
  - ➡ between their start and end
  - ➡ If there is no overlap, ordering is well defined (knowledge of global time)

Strict Consistency:
Operations are instantaneous
👉 Directly gives sequential order



Equivalent sequential history S exists:



1. Got the result S would get
2. No thread's history is violated in S
3. Non-overlapping operations retain order in S

Linearizability is Composable

What if we cannot tell what overlaps what

Subodh Kumar

Types of Registers

- Linearizable registers
  - ➡ SRSW 1-bit safe register
  - ➡ MRMW n-bit atomic linearizable register
- Sequentially consistent registers
- Causally consistent registers
- FIFO consistent registers
- Weakly consistent registers

Can build shared registers with high guarantee using ones with low guarantee

Can enforce order between read/write using shared registers (general synchronization)