Introduction to Parallel & Distributed Programming

Lec 08—OpenMP

Subodh Sharma | Jan 23, 2026

(RECAP): Work Sharing Constructs: Sections

- Each thread executes the region within the section
- Each section is executed only **ONCE**
- **Good use cases:**
  - Pipelined tasks
  - Overlapping I/O and compute
  - Multimodal streaming tasks
  - Divide-and-conquer

```
#pragma omp parallel
{
  #pragma omp sections
  {
    #pragma omp section
    { compute_A(); }

    #pragma omp section
    { compute_B(); }

    #pragma omp section
    { compute_C(); }
  } // implicit barrier at end of sections (unless nowait)
}
```

Work Sharing Constructs: Task

- Thread creates a task and add to the task pool when `#pragma omp task` is encountered
- Usually created by one thread in the parallel region
- **No implicit barrier** at the end

```
#pragma omp parallel
{
    #pragma omp single
    {
        #pragma omp task
        {
            printf("Task 1 handled by thread %d\n", omp_get_thread_num());
        }

        #pragma omp task
        {
            printf("Task 2 handled by thread %d\n", omp_get_thread_num());
        }
    }
}
```

Examples of Task Parallelism

Linked List Traversal

- Traversal by the `while loop`
- Tasks handle it perfectly
- `firstprivate (p)`
  - Creates private copy of the var for each thread
  - Initialised with the value from the master thread

```
#pragma omp parallel
{
    #pragma omp single
    {
        while (p) {
            #pragma omp task firstprivate(p)
            {
                process_node(p);
            }
            p = p->next;
        }
    }
}
```

Exercise: Parallelism

Fibonacci

- Draw the task to thread assignment map

```
#pragma omp parallel
{
    // Only one thread should
    #pragma omp single
    {
        result = fib(n);
    }
}
```

```
int fib(int n) {
    int x, y;
    if (n < 2) return n;

    #pragma omp task shared(x)
    x = fib(n - 1);

    #pragma omp task shared(y)
    y = fib(n - 2);

    #pragma omp taskwait
    return x + y;
}
```

Memory Consistency Models

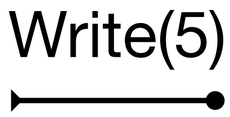
What is a Memory Consistency Model?

- **Set of rules** for how memory operations **on shared variables** become **visible** to different threads in a system.
- The contract is between the H/w, the compiler and the programmer:
  - **The programmer**: “I have developed the code with some intuition so that I don’t get bugs”
  - **The Compiler**: “I will optimise your code, which means reordering some reads and writes differently from the way you programmed it”
  - **The Hardware:** “I will execute instructions as fast as possible, which may mean running them out of order”

Linearisability

Strongest Memory Consistency Model

- An operation — has a start and has an end!
  - It starts with an invocation, and ends with a response

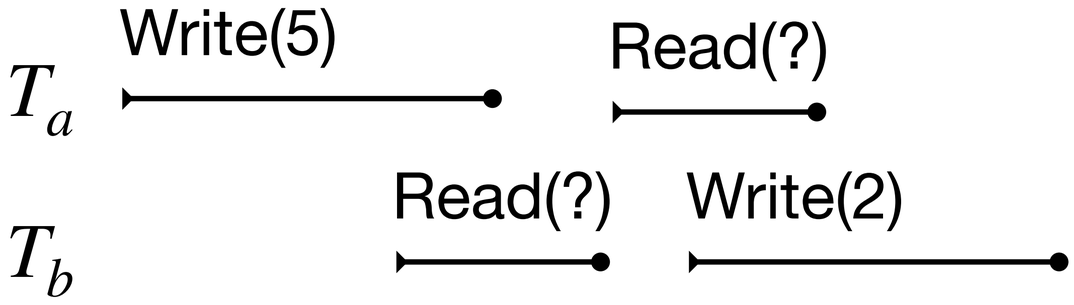


- **Rules of Linearisability:**
  - **Atomicity:** Every operations appears to happen **instantaneously** (between invocation & response)
  - **Order:** If operation A completes before operation B then A must appear to **happen before** operation B in the global history of events
  - **Correctness:** The final outcome must match the **sequential version**
  - **The order is not just logical but also respects real-time**

Linearisability

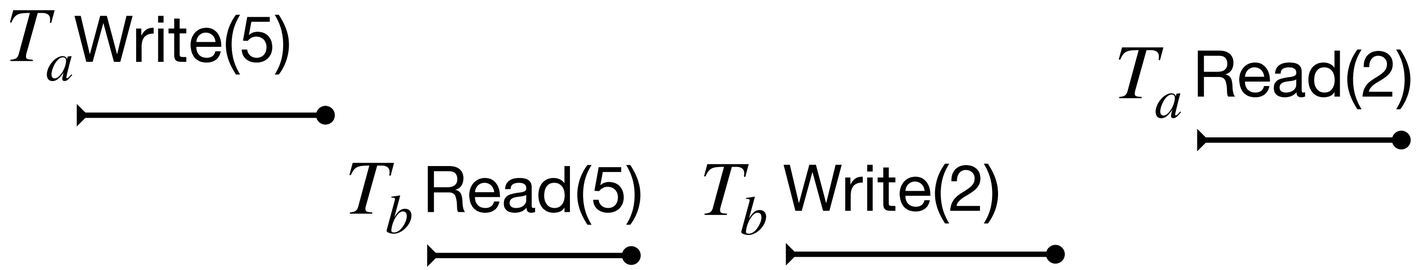
Visualising Linearisability

- Ordering is well-defined; **no overlap**
  - **Imagine a shared register**



- **Equivalent Sequential History**

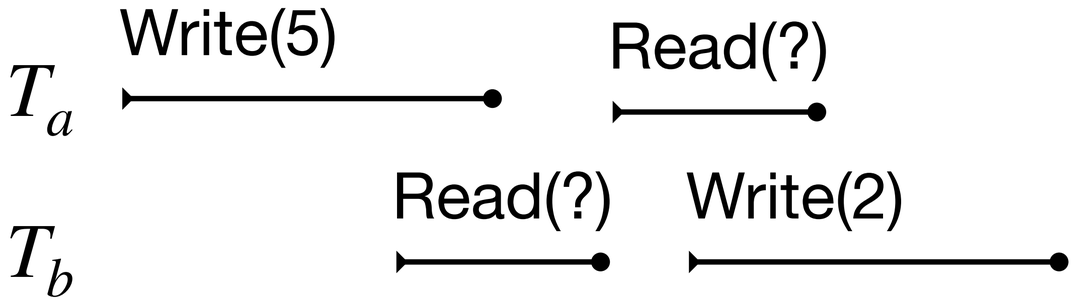
Valid, linearisable history!



Linearisability

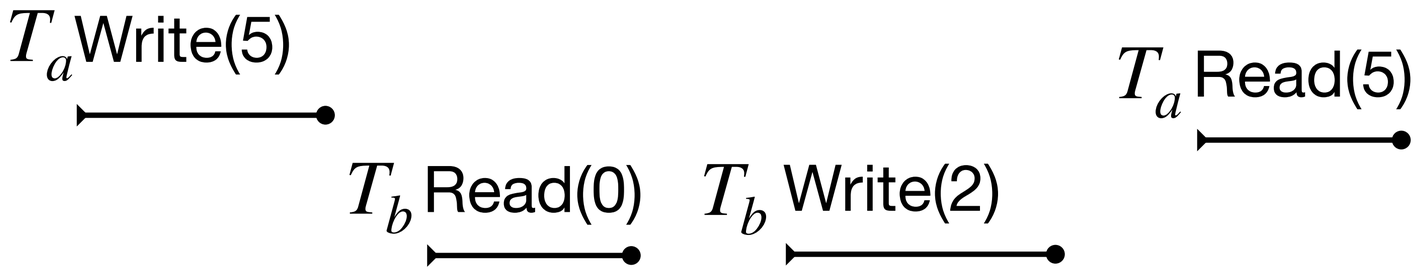
Visualising Linearisability

- Ordering is well-defined; **no overlap**
  - **Imagine a shared register**



- **Equivalent Sequential History**

Invalid, nonlinearisable history!



Linearisability

Realising OR Verifying Linearisability

- CPU cache update latencies may be non-uniform;
  - One would need expensive memory barriers to implement linearisability
- Network Latency: In distributed systems, this may impact visibility
  - Achieving linearisability via **consensus** such as **Paxos** or **Raft**
- **Verification: Hard problem**
  - Identify linearisation points
  - Show that all possible ordering of linearisation points lead to sequential (valid) histories