1 Introduction

In this lecture, we discussed implementing a k-ary atomic multi-reader single-writer from k-ary atomic single-reader single-writers. We argue that reads must also write in order to collaborate and maintain atomicity.

We informally review a multi-reader single-writer scenario in which atomicity is violated, followed by a proof of the theorem.

Finally, we introduce the concept of distributed shared memory and how it can be constructed from a message passing system through a network.

2 k-ary Multi-Reader Single-Writer Atomic Registers from k-ary Single-Reader Single-Writer Atomic Registers

2.1 The Atomic Multi-Reader Problem

We first informally examine the problem of implementing a k-ary atomic multi-reader single-writer from a k-ary atomic single-reader single-writer.

Consider a simple algorithm that does not work for atomic registers.

Assume that the writer’s algorithm is unknown, only assuming that the writer writes to multiple registers in some order. Suppose that the writer writes to two places, $x_1$ and $x_2$. Without loss of generality, assume the writer writes in $x_1$ first before writing to $x_2$, so we linearize the first write and then the second.

We can then construct an execution with two readers, $r_1$ and $r_2$, such that even though a reader reads after a write is completed, it returns a value that violates atomicity. For example, suppose, without loss of generality, that the initial values of $x_1$ and $x_2$ are 0. Now suppose that the writer writes a 1 to $x_1$, which is followed by a read by $r_1$. Before the writer writes a 1 to $x_2$, $r_2$ reads $x_2 = 0$, and so $r_1$ returns 1 and $r_2$ returns 0. This violates the atomicity property.

The essential claim is that there is no legal way to linearize writes (see Figure 1) while preserving temporal ordering, and thus readers need to write in order to communicate with each other to ensure consistency.
2.2 A Solution: Readers Must Also Write

Theorem 1 In a wait-free simulation of a single-writer multi-reader atomic register from single-writer single-reader atomic registers, at least one reader must also write.

Proof: Suppose, for sake of contradiction, that there exists a simulation for register $R$ where readers do not write. Let $P_w$ be the writer and $P_1$ and $P_2$ be the readers, and suppose that the initial value of $R$ is 0.

Note that since physical registers can only be read by a single reader, they can be partitioned into two sets, which we denote $S_1$ and $S_2$, such that $P_1$ reads from $S_1$, and $P_2$ reads from $S_2$.

Consider an execution $\alpha$ with a single write operation, WRITE(1). $P_w$ then writes to a series of physical registers $w_1, w_2, ..., w_k$, where $w_i$ is in either $S_1$ or $S_2$ for all $i$. Note that since the model is wait-free, there are no bounds on how long this process takes, so we can place a read between any two writes.

For each $i = 1, 2$ and each $j \in 0, ..., k$, we define an alternative execution $\alpha^i_j$ obtained from $\alpha$ by interposing a READ by processor $P_i$ after the linearization point of $w_j$ and before the linearization point of $w_{j+1}$. Let $v^i_j$ be the value returned by the READ operation.

Since the simulation guarantees atomicity for each $i = 1, 2$, there exists a $j_i$ between 1 and $k$ such that $v^i_j = 0$ for all $j < j_i$, and $v^i_j = 1$ for all $j \geq j_i$.

If $j_1 \neq j_2$, since $w_{j_1}$ writes in $S_1$ and $w_{j_2}$ writes in $S_2$, and $S_1 \cap S_2 = \emptyset$.

Without loss of generality assume $j_1 < j_2$.

Let $\alpha'$ be an execution where a READ by $P_1$ followed by a READ by $P_2$ is inserted between $w_{j_1}$ and $w_{j_2}$ (see Figure 2). Now $P_1$’s READ operation returns 1, while $P_2$’s READ operations returns 0. This violates the property of atomicity and thus is a contradiction.

Remarks: Informally, the basic idea here is that readers must write to each other, thus creating an ordering between them. For example, before a reader “decides” what value it has and returns it, it corresponds with other readers for verification. Essentially:
3 Distributed Shared Memory

We now turn to Distributed Shared Memory. The basic idea behind our initial constructions is that the shared memory model is an easier model to write algorithms for, but harder to implement while maintaining memory consistency, while message passing is, generally, a simpler model to implement.

Thus, we implement shared memory over a distributed network by constructing the system from message passing primitives. We can then argue about what sort of consistency conditions to allow and explore things such as data race free conditions, synchronization, strong reads and strong writes, and the tradeoffs between how powerful a system is and how many resources are required to implement it.

The system will present an interface like that of shared memory, with invocation and response operations, but the underlying operations with the network will be send and receive.