A Study of Software Primitives in the context of Concurrent Data Structures

---

Estudio de Primitivas en Software en el contexto de Estructuras de Datos Concurrentes

UNIVERSIDAD COMPLUTENSE DE MADRID

FACULTAD DE INFORMÁTICA

Trabajo de Fin de Grado

Grado en Ingeniería Informática

Mayo 2023

Autor

Lidia Casado Noguerales

Dirigido por

Samir Genaim

Abstract

A Study of Software Primitives in the context of Concurrent Data Structures

The "free ride" towards faster processor speeds at the pace set by Moore's Law has come to an end. Cramming ever smaller transistors on the same processor has reached a physical limit, and quantum technology is yet too immature to take on the challenge. Multiprocessor architectures have surged to meet the rising demand for computational power that has arisen in recent years. These architectures are capable of outperforming single-core architectures, but they require meticulous and orderly resource management to do so in a cost-efficient manner. This is where Concurrent Data Structures come to play. The new ultimate goal is to provide data structure designs that transparently manage workload balancing through several processors, ensuring correctness as well as versatility in a variety of concurrent settings.

In this context, we approach the subject of providing the building blocks for Concurrent Data Structures: software and hardware synchronisation primitives. These are the operations in charge of the most critical functionalities of concurrent programs, those having to do with shared-resource management. Synchronisation primitives have to satisfy specific constraints related to correctness conditions for concurrent executions, and are therefore delicate matters worthy of unhurried study. In particular, we dive into the algorithmic details concerning the *Non-blocking K-Compare-Single-Swap* (`KCSS`) primitive proposed at (Luchangco et al., 2008), a *non-blocking obstruction-free* software primitive aimed at meeting the challenges posed by Concurrent **Linked** Data Structures. We provide a profuse educational guide through every non-trivial design feature of `KCSS` culminating in the proposal of a fully-functional, efficient and transparent-to-the-user `C++` implementation, as well as usage instructions.

Keywords

Concurrent data structures, blocking and non-blocking synchronisation, shared memory, compare and swap.

Resumen

Estudio de Primitivas en Software en el contexto de Estructuras de Datos Concurrentes

El "viaje gratuito" en el aceleramiento de las velocidades de procesado al ritmo establecido por la Ley de Moore ha llegado a su fin. La integración de transistores cada vez más pequeños en un mismo procesador ha alcanzado un límite físico, y la tecnología cuántica es aún demasiado inmadura para asumir el desafío. Las arquitecturas multiprocesador han surgido para satisfacer la creciente demanda de poder computacional que ha surgido en los últimos años. Estas arquitecturas son capaces de superar a las arquitecturas de un solo núcleo, pero requieren una gestión de recursos meticulosa y ordenada para hacerlo de manera rentable. Aquí es donde entran en juego las Estructuras de Datos Concurrentes. El nuevo objetivo final es proporcionar diseños de estructuras de datos que gestionen de forma transparente el equilibrado de la carga de trabajo entre procesadores, asegurando corrección y versatilidad en distintos escenarios de concurrencia.

En este contexto, abordamos el tema de proporcionar los componentes básicos para la construcción de Estructuras de Datos Concurrentes: las primitivas software y hardware de sincronización. Estas son las operaciones encargadas de las funcionalidades críticas de los programas concurrentes, las que tienen que ver con la gestión de recursos compartidos. Las primitivas de sincronización tienen que satisfacer restricciones específicas relacionadas con las condiciones de corrección para ejecuciones concurrentes y, por lo tanto, son asuntos delicados merecedores de un estudio detallado. En particular, profundizamos en los detalles algorítmicos relacionados con la primitiva *K-Compare-Single-Swap sin bloqueo* (`KCSS`) propuesta en (Luchangco et al., 2008), una primitivo en software *sin bloqueo* (*non-blocking* en inglés) y *libre de obstrucciones* (*obstruction-free*) destinada a afrontar los desafíos planteados por las Estructuras de Datos **Enlazadas** Concurrentes. Aportamos una guía educativa acerca de las decisiones de diseño no triviales de `KCSS`, que culmina con la propuesta de una implementación en `C++` funcional, eficiente y transparente para el usuario, así como sus instrucciones de uso.

Palabras clave

Estructuras de datos concurrentes, sincronización de bloqueo y sin bloqueo, compartición de memoria, comparar e intercambiar.

Contents

Chapter 1

Introduction

1.1. Motivation

The "free ride" towards faster processor speeds at the pace set by Moore's Law has come to an end for the computer industry. 20 years ago we reached the conclusion that transistor technology would soon fail to keep up with the demands of this Law. In 2004 Intel released the first processor containing transistors smaller than 100nanometers, only to make a public statement later that year announcing that they were abandoning some single-core on-going projects in favour of a new line of development: dual-core chips.

Since the possibility of cramming ever smaller transistors on the same processor was reaching a dead-end – the only hope being the yet immature quantum technology – workload balancing through several processors became the new trend. More processors would allow exploiting the power of distributing tasks among them, avoiding altogether the physical complications that had arisen while trying to speed up processors by search of faster hardware. While many research topics still required time to develop, this new solution was set on the table to allow the computer industry to progress in the meantime. Soon, dual-core architectures would turn into quad-core, octa-core, ... Nowadays we refer to them as multi-core architectures.

Multi-core architectures have proven their capability to outperform single-core architectures, but they require meticulous and orderly resource management to do so. The best engineering designs are the result of a balance between efficiency and cost, and in our particular case this translates into a multi-core architecture that cannot afford the cost of having multiple storage units or multiple sets of every resource a processor needs in a computer, but instead has to be designed to be efficient with the sharing of those resources available. In fact, it is true that multi-core architectures do not use extra resources so as to not raise the cost of the end-product, but also for efficiency's sake, since sharing resources implies sharing their state, which implies sharing the information that state holds, which is key to coordinate processors. This explains the proliferation of commercial shared-memory multiprocessors: We are searching for a cost-efficient design that allows resource-

sharing among processors so as to speed up processing speeds by doing workload balancing among them.

Different strategies have been proposed to achieve efficient memory-sharing. Some rely on coarse-grained measures that require locking critical sections of code for correctness, sacrificing performance in exchange for simplicity; while a new trend has emerged in recent years that aims at obtaining more efficient, fine-grained equivalents to locking mechanisms. The former follows the typical "sequential" thinking seen in other forms of programming, but the latter requires deeper knowledge on new concepts introduced through the development of concurrent programming, as well as the challenges that come with it.

1.2. Goals

Throughout this report we will explore the latter of the alternatives just mentioned: fine-grained mechanisms for concurrent programming consisting on **hardware and software primitives** that can be safely executed by several processors accessing shared memory. This primitives constitute the building blocks of concurrent programming, ensuring the correctness of the most critical functionalities of shared-memory multi-core machines.

More specifically, we will dive into the study of a multi-location synchronisation software primitive devised by Victor Luchangco, Mark Moir and Nir Shavit (Luchangco et al., 2008): the `KCSS` Operation. This software primitive answers the need for a synchronisation tool suitable for the particular case of designing **concurrent Linked Data Structures**[1].

These data structures share the trait that their operations consist of several sub-operations to update the pointers that "link" them. Having these many sub-operations is an added challenge to provide a concurrent implementation because the more steps an operation consists of, the harder it is to synchronise threads executing concurrently on it. We will tackle this problem by exhaustively exploring available solutions to the most common synchronisation problems, (e.g. managing concurrent modifications of two locations that are interdependent), and comparing their performance to that of `KCSS`.

The final outcome of this work will be an exhaustive, educational study of well-known concurrency primitives and their impact on concurrent data structures; a further in detail explanation of the intricacies of the `KCSS` operation, shedding light on non-trivial design and implementation details that will lead us to proposing our own `C++` code implementation. This code proposal will serve as a pedagogic tool by exemplifying the materialisation of solutions to the challenges encountered through our study, while being easy to use both for the user having read this work and for the inexperienced user only having a vague knowledge on the functionality provided by `KCSS`.

[1] Linked Data Structures include: linked lists, arrays, list-based sets, hash maps, ...

1.3. Work Plan

To attain the goals described on the previous section, the first step is studying the broad topic of concurrency through synchronisation paradigms, open research areas and targeted software and hardware primitives and the problems tackled by them. After this background examination, the problem posed by Concurrent Linked Data Structures is contextualised and we can go ahead and dive into the research paper through which authors Victor Luchangco, Mark Moir and Nir Shavit (Luchangco et al., 2008) proposed the `KCSS` Operation. After non-trivial, thorough work to reason about its correctness, we are able to justify every design decision and follow the paper to the detail. Having acquired the necessary understanding of the challenge at hand we can go ahead and carry out a profuse and detailed explanation of the `KCSS` primitive with added resources and insights. Finally, the culmination of this work is the proposal of a code solution that is fully functional and easy to use, incorporating suggested improvements.

1.4. Structure

The work carried out is presented throughout this report together with the code which can be found at the following code repository: `https://github.com/Liidiacnog/KCSS_Project`.

The report is organised in the following way:

1. Chapter 2 gives an introduction and contextualisation of key concurrency problems, concepts and tools that will be employed in subsequent chapters.

2. Chapter 3 addresses the challenges posed by Concurrent Linked Data Structures both through well-known and novel concurrency tools, in the context of implementing a Concurrent Linked List.

3. Chapter 4 studies the `KCSS` Operation in detail: its mechanism, components, correctness guarantees, limitations and related work.

4. Chapter 5 studies our code proposal for this operation, how it covers the scenarios described in the previous chapter, and how to use it through a practical example.

5. Chapter 6 presents our conclusions from the work carried out, and proposals of future directions for research and improvement.

Introducción

Motivación

El "viaje gratuito" en el aceleramiento de las velocidades de procesado al ritmo establecido por la Ley de Moore ha llegado a su fin para la industria informática. Hace 20 años llegamos a la conclusión de que los avances en tecnología de transistores pronto dejarían de cumplir con las exigencias de esta Ley. En 2004, Intel lanzó el primer procesador que contenía transistores de menos de 100nanómetros, haciendo una declaración pública ese mismo año en la que anunciaban que abandonaban algunos proyectos de un solo núcleo a favor de una nueva línea de desarrollo: chips de doble núcleo.

Dado que la posibilidad de incrustar transistores cada vez más pequeños en el mismo procesador estaba tocando fondo (la única esperanza siendo la tecnología cuántica, aún inmadura), el equilibrado de la carga de trabajo entre procesadores se ha convertido en la nueva tendencia. Más procesadores permiten explotar el poder de distribuir las tareas entre ellos, evitando así las complicaciones físicas que habían surgido al tratar de acelerar los procesadores desde los transistores. Mientras muchos temas aún estaban en desarrollo, esta nueva solución se planteó para permitir que la industria informática progresase entre tanto. Pronto, las arquitecturas de doble núcleo (dual-core) se convertirían en cuatro núcleos (quad-core), ocho (octa-core),... Hoy en día nos referimos a ellas como arquitecturas multinúcleo (multi-core).

Las arquitecturas multinúcleo han demostrado su capacidad para superar a las de un solo núcleo, pero requieren una gestión de recursos meticulosa para ello. Los mejores diseños en ingeniería son el resultado de un equilibrio entre eficiencia y coste, y en nuestro caso particular esto se traduce en una arquitectura multinúcleo que no puede permitirse el coste de tener múltiples unidades de almacenamiento o múltiples conjuntos de cada recurso que necesita un procesador en un computador, sino que debe diseñarse para que sea eficiente compartiendo los recursos disponibles. De hecho, es cierto que las arquitecturas multinúcleo no utilizan recursos extra para no encarecer el producto final, pero también por motivos de eficiencia ya que compartir recursos implica compartir su estado, lo que implica compartir la información que ese estado contiene, que es clave para coordinar los procesadores. Esto explica la proliferación comercial de multiprocesadores de memoria compartida: estamos

buscando un diseño rentable que permita compartir recursos entre procesadores para acelerar las velocidades de procesamiento equilibrando la carga de trabajo entre ellos.

Múltiples estrategias han sido propuestas para lograr un intercambio de memoria eficiente. Algunas se basan en medidas de granularidad gruesa (*coarse-grained* en inglés) que requieren bloquear secciones críticas de código para que sean correctas, sacrificando rendimiento a cambio de simplicidad; mientras que ha surgido una nueva tendencia en los últimos años que aspira a obtener equivalentes más eficientes que los mecanismos de bloqueo: las medidas de granularidad fina (*fine-grained* en inglés). Mientras las primeras se amoldan al razonamiento "secuencial" común en otros paradigmas de programación, las segundas requiere un conocimiento más profundo de los conceptos que la programación concurrente ha introducido, así como de los desafíos que conlleva.

Objetivos

A lo largo de este informe exploraremos la segunda de las alternativas mencionadas: mecanismos de granularidad fina para programación concurrente que consisten en **primitivas de hardware y software** que pueden ser ejecutadas de manera segura por varios procesadores que acceden a regiones de memoria compartidas. Estas primitivas constituyen los componentes básicos de la programación concurrente, asegurando la corrección de las funcionalidades más críticas de las arquitecturas multinúcleo de memoria compartida.

Concretamente, profundizaremos en el estudio de una primitiva de sincronización en software capaz de actuar sobre múltiples ubicaciones, ideada por Victor Luchangco, Mark Moir y Nir Shavit (Luchangco et al., 2008): la operación `KCSS`. Esta primitiva de software responde a la necesidad de una herramienta de sincronización adecuada para el caso particular de diseñar **Estructuras de Datos Enlazadas Concurrentes**[2] .

Estas estructuras de datos comparten la característica de que sus operaciones consisten en varias suboperaciones para actualizar los punteros que las "enlazan". Tener tantas suboperaciones es un desafío adicional para proporcionar una implementación concurrente porque cuantos más pasos requiere una operación, más difícil es sincronizar subprocesos que la ejecutan simultáneamente. Abordaremos este problema explorando exhaustivamente las soluciones disponibles para los problemas de sincronización más comunes (por ejemplo, administrar modificaciones simultáneas de dos ubicaciones que son interdependientes) y comparando su rendimiento con el de `KCSS`.

El resultado final de este trabajo será un estudio exhaustivo de las primitivas de concurrencia ya conocidas y su impacto en las estructuras de datos concurrentes; una explicación más detallada de las complejidades de la operación `KCSS` en concreto,

[2]Las estructuras de datos enlazadas incluyen: listas enlazadas, arrays, conjuntos basados en listas, mapas hash, ...

arrojando luz sobre sus decisiones de diseño no triviales y los detalles de implementación que nos llevarán a proponer nuestra propia implementación de código C++. Esta propuesta de código servirá como una herramienta pedagógica al ejemplificar la materialización de soluciones a los desafíos encontrados a través de nuestro estudio, siendo fácil de usar tanto para el usuario que ha leído este trabajo como para el usuario inexperto que posee un conocimiento difuso sobre la funcionalidad proporcionada por KCSS.

Plan de trabajo

Para lograr los objetivos descritos en la sección anterior, el primer paso es estudiar el amplio tema de la concurrencia a través de paradigmas de sincronización, áreas de investigación abiertas y primitivas de software y hardware específicas y los problemas que abordan. Después de este análisis de antecedentes, se contextualiza el problema planteado por las Estructuras de Datos Enlazadas Concurrentes y podemos profundizar en el trabajo de investigación a través del cual los autores Victor Luchangco, Mark Moir y Nir Shavit (Luchangco et al., 2008) han propuesto la operación KCSS. Después de un esfuerzo minucioso para razonar sobre su corrección, nuestro trabajo es una guía detallada para justificar cada decisión de diseño y seguir la propuesta de KCSS al detalle. Una vez adquirida la comprensión necesaria del desafío en cuestión, podemos llevar a cabo una explicación detallada de la primitiva KCSS con recursos y sugerencias adicionales. Finalmente, la culminación de este trabajo es la propuesta de una solución de código funcional y fácil de usar, que incorpora las mejoras sugeridas.

Estructura

El trabajo realizado se presenta a lo largo de este informe junto con el código que se puede encontrar en el siguiente repositorio de código: https://github.com/Liidiacnog/KCSS_Project.

El informe está organizado de la siguiente manera:

1. Capítulo 2 proporciona una introducción y contextualización de problemas, conceptos y herramientas clave en concurrencia, que se emplearán en capítulos posteriores.

2. Capítulo 3 aborda los desafíos planteados por las Estructuras de Datos Enlazadas Concurrentes a través de herramientas de concurrencia conocidas y novedosas, en el contexto de la implementación de una lista enlazada concurrente.

3. Capítulo 4 estudia la operación KCSS en detalle: su mecanismo, componentes, garantías de corrección, limitaciones y trabajo relacionado.

4. Capítulo 5 estudia nuestra propuesta de código `C++` para esta operación, cómo solventa los escenarios descritos en el capítulo anterior y cómo usarlo a través de un ejemplo práctico.

5. Capítulo 6 presenta nuestras conclusiones del trabajo realizado y propuestas de futuras direcciones de investigación y mejora.

Chapter 2

Contextualising Concurrent Data Structures

2.1. Definition and Characteristics

This work focuses on **Concurrent Data Structures for Shared-Memory Multi-Processor Systems**, which are data structures intended for these systems where, in contrast to uni-processor systems, multiple threads of execution can execute concurrently.

The fact that they are *multi-processor* systems means their architecture is made up of multiple processors (cores) that can combine their computing power. On the other hand, being *shared-memory* systems implies that there exists a shared address space between all processors, irregardless of whether each one has other local memory units for their own use. We will refer to *processors (cores)* as the hardware devices that constitute the Central Processing Unit (CPU) of a computer. Each processor can spawn *threads*: software constructs that represent the flow of execution of a program. For clarification, different operating systems have different definitions for words *thread* and *process*, since they are both units of flow control. In this report we will refer to them as the same software construct because we are only interested in describing their behaviour as units of flow control that could execute concurrently, but we are not interested in particularities of one or another specific operating system or architecture.

When we say that threads can execute *concurrently* we mean that they are capable of **executing at the same time** (this does not mean they will execute *synchronously*, as we will now explain). Firstly, for the systems under study (*shared-memory*, *multi-processor*) we assume that concurrent execution of more than one thread is possible because each processor can spawn at least one thread of execution at the same time (**concurrently**) as the rest, and we are in the context of a system with several cores. The fact that threads do not necessarily execute synchronously is key to the understanding of the execution contexts we will be analyzing throughout the following chapters: we have said that threads will execute "asynchronously" but

```
1  shared volatile int count;
2
3  fun produce(){
4      while(true){
5          if(count == 0){
6              // Perform some costly operation
7               ...
8              count += 1;
9              print("Produced 1 unit \n");
10         }
11     }
12 }
```

Figure 2.1: *Producer* Program

at the same time. Executing "asynchronously at the same time" means that threads do not necessarily have the same clock frequencies, so they are not synchronous, but they **are running at the same time**, at their own pace. This ultimately means that in some situations the behaviour of threads will greatly vary from the sequential counterpart. For example, in the concurrent setting we cannot expect all threads to halt at the same instant upon the occurrence of a certain event that they can all observe. We will later dive deeper into the kind of *events that all threads can see*. For now we will simply say that every thread can check if specific conditions are observed on the resources it has access to, and most commonly these are related to reads and writes to shared-memory. For our purposes, this means that we will have to be extra careful when reasoning about the behaviour of threads and the correctness of concurrent data structures, which will challenge our intuition.

To make all of the previously mentioned ideas clear, let us see an example of a well-known concurrency problem and how it is solved using the fine-grained mechanisms we have been talking about: **the Producer-Consumer Problem**. Note that this problem is purposefully built so as to study the shortcomings of programs built for uni-processors when used in a concurrent setting, and it can seem somewhat artificially built. Suppose we have a multi-processor system consisting of three cores (which we will refer to as $Core_1$, $Core_2$ and $Core_3$, respectively), and the two programs depicted in figures Figure 2.1 (*Producer*) and Figure 2.2 (*Consumer*).

In this code, we assume that variable `count` is a shared variable, meaning it is located in a memory region belonging to a **shared address space** that all processors can access, and therefore so can any threads they create. `count` is declared as `volatile`, meaning on every access to it or use of it, its value will be fetched directly from memory, and not cached. We also assume that `count` initially holds value 0.

The context of use of programs such as the *Producer* and *Consumer* is one where a costly operation has to be performed every time another particular operation takes place: one would be performed by the `produce()` code, and the other one by the `consume()` code. A plausible example of one such operation could be reading/writing data from/to a buffer. In that case, the *Producer* (Figure 2.1) would be in charge of performing the "costly operation" of reading data from the input source, and the

```
shared volatile int count;

fun consume(){
    while(true){
        if(count != 0){
            // Perform some costly operation
             ...
            count -= 1;
            print("Consumed 1 unit \n");
        }
    }
}
```

Figure 2.2: *Consumer* Program

```
    Produced 1 unit
    Consumed 1 unit
    Produced 1 unit
    Consumed 1 unit
    Produced 1 unit
    Consumed 1 unit
    Produced 1 unit
    ...
```

Figure 2.3: Possible output obtained by letting three threads execute concurrently, two of which run on the *Consumer* Program, one on the *Producer* Program.

Consumer (Figure 2.2) of writing it to the corresponding output destination. Variable `count` is the mechanism that both programs will use to let each other know that the other one has already performed its operation.

Now let us suppose $Core_1$ and $Core_2$ each spawn a thread to execute the *Consumer*, and $Core_3$ another one to execute the *Producer*. We will refer to these threads as $Consumer_1$, $Consumer_2$ and $Producer_1$, respectively. $Consumer_1$ and $Consumer_2$ "want" to set the value of `count` to 0 (because it will mean they have finished their "consumption phase", decrementing `count` by 1 unit), while $Producer_1$ wants to set `count`'s value to 1 (it wants to go through its "production phase", which increments `count` by a unit).

Given that the initial value of `count` is 0, it is reasonable to assume that once $Producer_1$, $Consumer_1$ and $Consumer_2$ start running, the sequence of strings depicted on Figure 2.3 will be printed until we stop the execution of all threads. However, in reality we have no guarantees that this will be the case. This particular set of strings is the result of a specific succession of events consisting of: thread $Producer_1$ accessing `count`'s memory address and seeing a 0 stored in it, then adding 1 to it and printing `Produced 1 unit`; then one of the consumer threads ($Consumer_1$, or $Consumer_2$) accessing `count`'s memory address and seeing a 1 stored in it, subtracting 1 from it and printing `Consumed 1 unit`, which is when the cycle would start again. This is the expected behaviour in a sequential setting. This behaviour can be expected if we picture threads as parallel flows of execution that execute every

```
1     Consumed 1 unit
2     Consumed 1 unit
3     Produced 1 unit
4     Consumed 1 unit
5     Consumed 1 unit
6     Consumed 1 unit
7     Consumed 1 unit
8     Consumed 1 unit
9     ...
```

Figure 2.4: Possible output obtained by letting three threads execute concurrently, two of which run on the *Consumer* Program, one on the *Producer* Program.

code statement in synchronization to a single clock.

In a concurrent setting such as the one we have described (three threads executing at the same time while sharing a common variable `count`), the output shown by Figure 2.3 is one possibility among many, due to the non-deterministic component that the interaction between asynchronous core clocks opens the door to. Another possible output of the Producer-Consumer setup we have described could also be the one depicted by Figure 2.4.

At a first glance, this does not make sense because in order to print string `Consumed 1 unit` value `count` has to be different from 0, and the only way to achieve this is through the execution of one iteration of the *Producer* by thread $Producer_1$. However, each of these would result in the printing of string `Produced 1 unit`, and we do not see this string until the third line of output is reached. Moreover, we do not see any more apparitions of string `Produced 1 unit` at all.

In this concurrent setting, the following can happen (see Figure 2.5 for a visual illustration of the situation):

- $Producer_1$ thread reads a 0 in `count`, and proceeds to increment `count` by 1 unit.
- Before $Producer_1$ has time to print `Produced 1 unit`, both $Consumer_1$ and $Consumer_2$ read a 1 in `count`, which satisfies the condition of their `if-clause`. They both proceed to execute the line which modifies `count`'s value (`count` is currently -1).
- Now, $Consumer_1$ and $Consumer_2$ (each at its own pace, not necessarily at the same time) proceed to printing string `Consumed 1 unit`. Then they restart the loop.
- At this moment, $Producer_1$ manages to finalise its printing instruction, so the console currently contains two `Consumed 1 unit` strings followed by one `Produced 1 unit` string.

The problem that we encounter at this instant is that `count` now holds value -1, and therefore $Producer_1$'s `if-clause` will never be satisfied again, since the only

Consumer Program

```
1  shared volatile int count;
2
3  fun consume(){
4      while(true){
5          if(count != 0){
6              // Perform some costly operation
7               ...
8              count -= 1;
9              print("Consumed 1 unit \n");
10         }
11     }
12 }
```

Producer Program

```
1  shared volatile int count;
2
3  fun produce(){
4      while(true){
5          if(count == 0){
6              // Perform some costly operation
7               ...
8              count += 1;
9              print("Produced 1 unit \n");
10         }
11     }
12 }
```

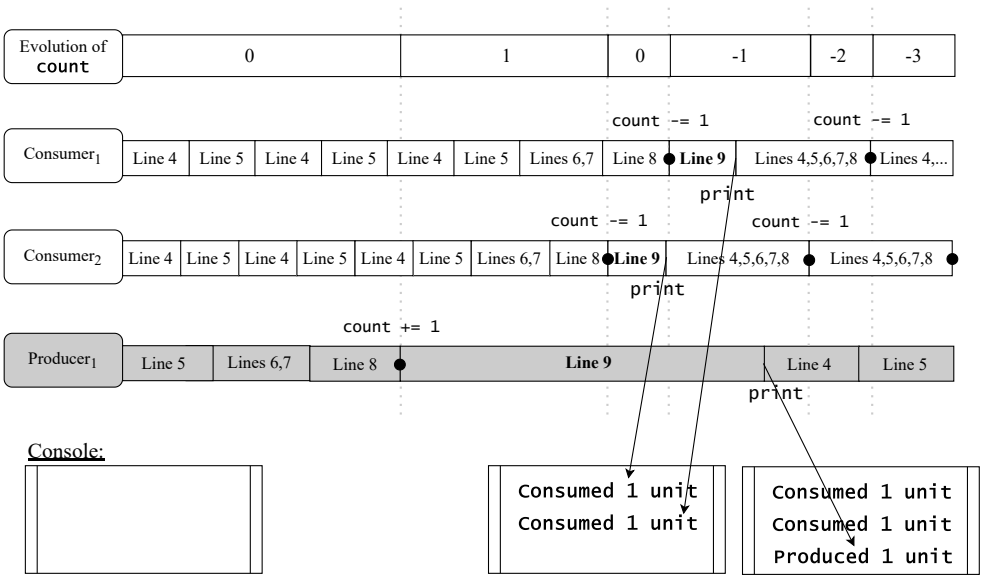


Figure 2.5: Illustration of a possible execution of the *Producer* and *Consumer* Programs in a setting where there are two consumers: $Consumer_1$ and $Consumer_2$, and one producer $Producer_1$. Note that the lines of code referred to by the figure correspond to those at the code that each thread is executing ($Consumer_1$ and $Consumer_2$ execute the *Consumer* Program, whereas $Producer_1$ executes the *Producer* Program).

executing threads are both "consumers" of this value. $Consumer_1$ and $Consumer_2$'s `if-clause` will be satisfied at every iteration, endlessly decreasing `count`'s value until their execution is stopped.

What the *Producer-Consumer Problem* illustrates is the fact that concurrent settings introduce non-determinism into executions derived from their asynchronous execution with respect to each other. Threads are complex software constructs that require a fine balance between **independence**, in order to exploit the power of multi-core architectures, and constant **inter-communication** in order to not impede other threads from progressing or in order not to duplicate work already done by another thread. This is the motivation for doing research on software primitives enabling communication and synchronisation of some sort between threads.

2.2. Principles of Concurrency

Throughout this section we will review key concurrency concepts involved in the reasoning to construct correct Concurrent Data Structures.

Let us think back to the *Producer-Consumer Problem* example. How can we make it execute as Figure 2.3 shows? We want methods `produce()` and `consume()` to execute the code within their `if-clause` as an indivisible unit of code, so that threads can observe the result of the actions carried out by these lines of code as a single outcome occurring at an indivisible moment in time. The concurrency concept summarising this property is *atomicity*.

2.2.1. Atomicity

> *Atomicity* (Moir and Shavit, 2004): A code block is guaranteed to be *atomic* if no thread can observe a state in which the block has been partially executed.

Achieving *atomicity* can be very costly, and therefore it is common practice to aim at concurrent programs that limit their requirements of *atomic* operations as much as possible while ensuring correctness. There exist software and hardware primitives providing *atomic* operations for commonly needed actions.

One such example are *atomic* **read-modify operations**, for which all modern multiprocessors provide one of the following primitives: *load linked/store conditional* (`LL/SC`) or *compare-and-swap* (`CAS`).

- Compare-And-Swap (`CAS`): operation that *atomically* loads a memory location, compares the value read to an expected value, and stores a new one at the location if the comparison succeeds. See Figure 2.6 for the semantics of this instruction in pseudo-code.

```
1   bool CAS(L, E, N) {
2       atomic{
3           if(*L == E) {
4               *L = N;
5               return true;
6           } else {
7               return false;
8           }
9       }
10  }
```

Figure 2.6: The semantics of the `CAS` operation. The *atomic* keyword requires the block it labels to be executed *atomically*. Figure taken from (Moir and Shavit, 2004).

- Load Linked/Store Conditional (`LL/SC`): Information taken from (Barreira, 2023). Together, these two operations provide a means to synchronise memory `load` and `store` operations.

  - Load Linked (`LL`): It reads a memory address α, and stores address α into a special register called the *link register*, local to each processor. This register's content are deleted in the rest of processors whenever another thread performs an `LL` operation **to the same memory location**; or whenever the same thread that performed `LL` calls `SC` and it succeeds.
  - Store Conditional (`SC`): It stores the content of a register `R` in a memory address α, but only if α is the same address as the one stored in the *link register*. Otherwise, no action is performed. If `SC(`α`,R)` was successful it returns `true`, otherwise it returns `false`.

  An `LL` operation on address α, followed by a successful `SC` operation on the same address guarantees that α's content did not change between the call to `LL` and the call to `SC`. This behaviour provides a means to know whether `LL/SC` were executed as if they were *atomic* (without being interrupted by other threads attempting modifications to α), by analyzing the value returned by `SC`. If a thread T_1 calls `LL(`α`)`, and then `SC(`α`,R)`: if `SC` returns `true` we are guaranteed that thread T_1's *link register* contained address α, read by the `LL` operation performed by T_1. This means no other `LL(`α`)` operation was performed in between T_1's `LL/SC` calls, because another thread's call to `LL` with the same address would have deleted T_1's *link register*'s content. If `SC` returns `false`, we know another thread attempted to modify α's content, preventing T_1 from completing its `LL/SC` operation.

2.2.2. Linearizability

Another key concept in concurrency is *linearizability* of operations. As we have seen, in concurrent settings it is not always straightforward to point out the exact moment when a specific action has taken place. Actions carried out by some threads might be obscured by other concurrently-running threads' own actions, making their

outcome invisible or confusing for a spectator unaware of the inner workings of the code. To reason about concurrent programs' correctness, we define *linearizability* to precisely refer to this moment in time when we can talk about the consequences of the execution of an operation/a set of operations on the system.

> (Herlihy and Shavit, 2008) An execution is called *linearizable* if each operation appears to take effect instantaneously at some point between its invocation and its response. This point in time is called its *linearization point*.

2.2.3. Blocking and Non-Blocking Constructs

Concurrency's main goal is to exploit the parallel computation capabilities of multi-core systems. Ideally, every task being executed would have a disjoint set of independent sub-tasks on which the initial task could be subdivided so that a one-to-one assignment from tasks to cores could be drawn. However, it is often the case that this is not possible, and inter-dependencies between sub-tasks impose limits on the progress that threads can make when they are running concurrently and carrying out related sub-tasks.

There exists a classification for the "amount of progress" that threads can make in the situation described: *Blocking* and *Non-Blocking Constructs*. Information taken from (Moir and Shavit, 2004).

- *Blocking Constructs*: Constructs belonging to this class do not guarantee any progress to be made by threads executing them. Threads could block forever, there are no guarantees that the execution will terminate. One example are constructs employing locks.
- *Non-Blocking Constructs*: Constructs belonging to this class guarantee that threads will progress if specific conditions are met. These conditions are called *Progress Conditions*, and they require that the failure or indefinite delay of a specific thread does not prevent other threads from making progress. Depending on the necessary *Progress Conditions*, several sub-classes are defined:
  - *Wait-Freedom*: A *wait-free* construct guarantees it will finalise after **a thread executing it takes a finite number of its own steps**, regardless of the timing behaviour of other threads.
  - *Lock-Freedom*: A *lock-free* construct guarantees that after a thread executing this construct takes a finite number of its own steps, **some thread's execution will finalise** (it could either be this one or another thread executing the same construct).
  - *Obstruction-Freedom*: An *obstruction-free* construct guarantees it will finalise if **a thread executing it manages to not encounter interference from other threads for a finite number of steps**.

The subject of this study: the `KCSS` operation, is *non-blocking*, and in particular *obstruction-free*, as we will later see.

In the concurrent setting, what defines a software construct's progress guarantees are the synchronization primitives it employs. These are the building blocks in charge of the key actions that can compromise the correctness of the final end-product. Synchronization primitives are nothing more than operations carrying out specific tasks that are straightforward in the sequential setting but require a redefinition in the concurrent setting because their sequential counterpart does not support concurrent uses.

There exist **software** and **hardware** synchronisation primitives. The latter are the optimal choice in general, but not all systems provide them due to their dependence on architecture-specific features. One example is the *Compare And Swap* (`CAS`) operation, which is nowadays supported by most modern architectures. Software primitives, on the other hand, are malleable and customisable to our needs, in exchange of being higher-level, less efficient operations. There are many examples of software primitives, e.g. there exist variations of the `CAS` operation providing *Double-Location Compare And Swap* (`CAS2` or `DCAS`) (Greenwald, 1999), or *N-Location Compare And Swap* (`CASN`) (Harris et al., 2002).

In the following chapters, we will explore the limits and capabilities of different synchronization primitives, and compare them to the `KCSS` primitive that is the main subject of this work.

Chapter 3

Linked Data Structures: Implementing a Concurrent Linked List

The design of linked concurrent data structures (e.g., linked lists, arrays, list-based sets, hash maps, etc) serves as an interesting study ground for concurrency tools. These data structures share the trait that their corresponding operations consist of several sub-operations, e.g., to update the pointers that "link" them. Having these many sub-operations is an added challenge to provide a concurrent implementation because the more steps an operation consists of, the harder it is for a thread to make them appear *atomic* to the rest of threads. In the next sections we discuss solutions to solve this problem.

3.1. The Locking Mechanism

Suppose we want to implement a linked list with support for concurrency, and we want to use locks to do it. As we know, "locking" consists of applying a set of directives to some components of the data structure with the purpose that at most one thread can have access to those parts. Depending on the number of components affected by the locking directives, we can talk about coarse-grained locking, or fine-grained locking.

3.1.1. Coarse-Grained Locking

We can apply coarse-grained locking to a linked list by employing a lock that encompasses the data structure as a whole. The linked list code can be the same as the sequential one, except that no access is granted to it unless the executing thread acquires the list's lock beforehand.

Suppose we have such a list, implemented having a coarse lock over the full list, initially containing two elements as depicted in Figure 3.1. In Figure 3.2 we can see an example where two threads (T_1 and T_2) respectively insert and delete an element

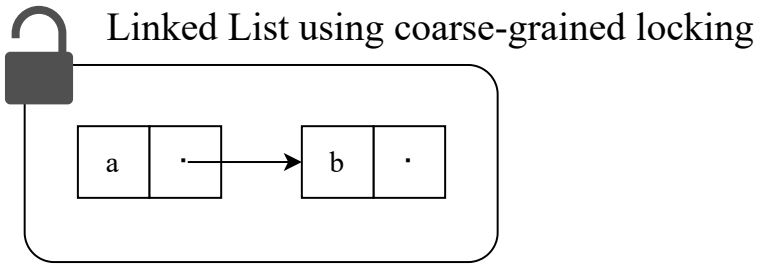


Figure 3.1: Initial state of the linked list implemented using a coarse lock.

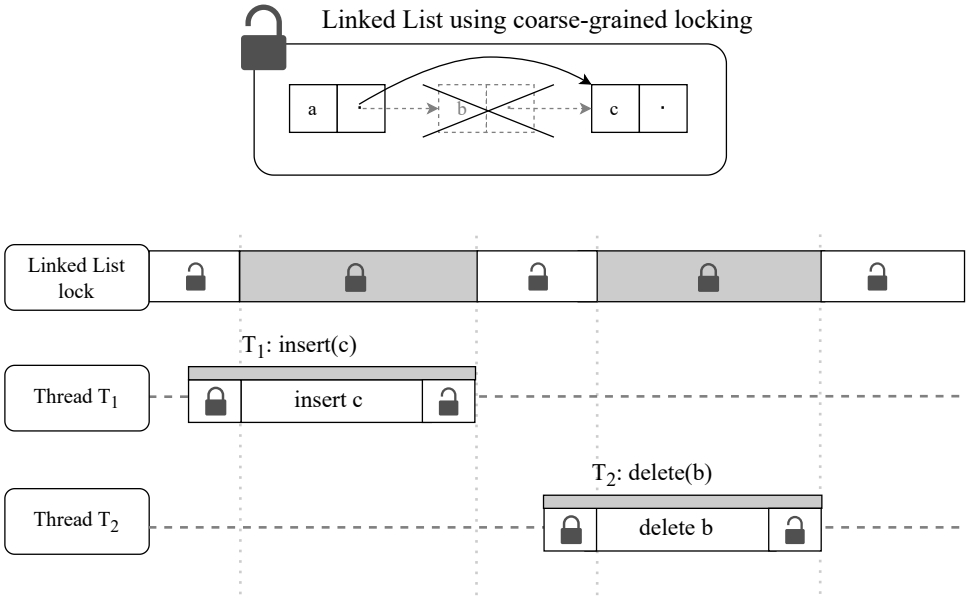


Figure 3.2: Illustration of a linked list implemented using a coarse lock that encompasses the whole data structure. Thread T_1 performs an insertion operation, T_2 a deletion.

using this linked list. T_1 and T_2 need to go through a locking phase before executing their respective operations, where they request the lock and only proceed succeed to obtain it. After the execution, they have to perform a lock-releasing stage, where they announce that they have finished modifying the list by releasing the lock so that another thread can acquire it. In this particular example, T_1 inserts element c while no other thread is requesting access to the list, which means it encounters no contention upon requesting the lock. The same happens for T_2 when it executes the deletion of element b. If this was not the case and both threads attempted to execute any operation at the same time, then the lock would guarantee that only one of them obtain it, while the other one would fail and have to be restarted to attempt to hold it once again. While no operation is performed over the list, it remains unlocked, available to any thread capable of acquiring its lock.

The behaviour we have just described is that of a sequential list. This mechanism is called coarse-grained because the granularity of the lock is coarse. Concur-

```
LinkedList_Node {
  Element elem;
  Node* next_pointer;
}
```

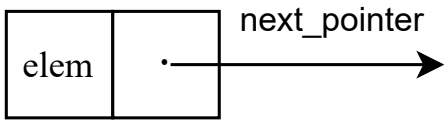


Figure 3.3: Illustration of a Node structure in the Linked List implementation.

rent accesses are "managed" by disallowing them. Only one thread among several concurrently requesting the lock will acquire it, and therefore only one operation at a time can be performed. The lock acts as a referee that provides access to the data structure.

Because of this, the list's performance is no different from the sequential setting. The implementation can run on a multiprocessor architecture but it will not benefit from its multiple cores because it handles concurrent accesses by disallowing them.

3.1.2. Fine-Grained Locking

Let us now see a possible use of fine-grained locking on a linked list. Instead of having a lock on the complete list, suppose we add a lock on every node. Operations wanting to modify the list, like deletion or insertion of nodes, have to acquire the locks to all nodes participating in the operation before executing. Figure 3.3 illustrates a Node structure with this scheme.

The only issue preventing this implementation from working are deadlocks. Suppose we have a list initially containing elements a and c, as depicted in Figure 3.4. In Figure 3.5 we can see an example of two threads concurrently executing an insertion and a deletion on this linked list, that leads to deadlock. T_1 wants to insert element b, while T_2 wants to delete element c. In order to do so, both threads have to first lock the nodes they are going to require for their respective operations. Insertions first lock the node that will be predecessor to the one being inserted, so T_1 locks a. Deletions first lock the element that is going to be subjected to the elimination, so T_2 locks c. Now the insertion requires locking the node that will be successor to the one being inserted. This is element c, which is already locked by T_2. However, T_2 is blocked because in order to fulfill its deletion operation it has to lock the node that precedes the one it is about to delete, which is node a, locked by T_1. As we can see, no thread can make progress because it is waiting for the other one to release a lock, while each has acquired precisely the lock of the node that the other thread needs in order to proceed. This is a deadlock situation.

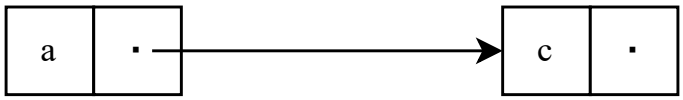


Figure 3.4: Initial state of the linked list containing elements a and c, implemented using a fine-grained locking mechanism.

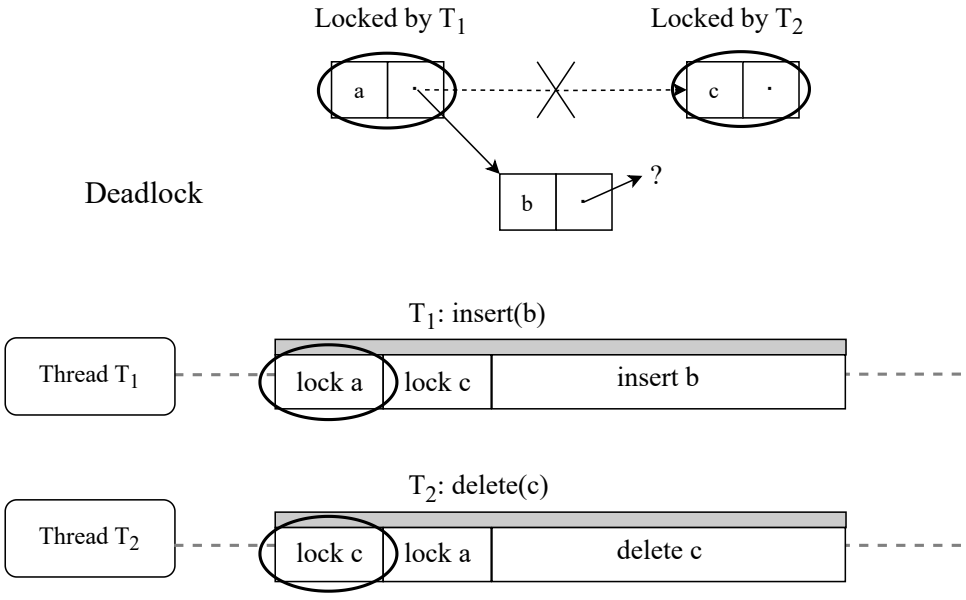


Figure 3.5: Illustration of a deadlock situation derived from the use of locking mechanisms.

Deadlocks can be solved by using constructs equivalent to C++'s `try_lock()`, which attempts to hold the lock, but fails and returns immediately if it is already held by another thread, instead of blocking until it is achieved like what `lock()` does. On failure, we could retry[1] to apply the operation, and eventually one of the threads would make progress and release the node(s) that the other one(s) need.

Another solution to deadlocks is imposing a total order between nodes of the list, to be used as the criterion to order node-locking requests. In our example of Figure 3.5, if the total ordering consisted in the node ordering of elements stored at the nodes: both T_1 and T_2 would initially request a's lock, which would result in only one of their operations succeeding, while the other one would fail and have to be "retried"[1] later due to unavailability of access to one or more of the nodes involved in it.

With the aforementioned considerations, we could provide a correct linked list implementation. We would just need to add a "locking phase" at the beginning of every operation defined in the sequential implementation of this data structure, which would request the lock of every node involved in that operation. We would be working with code that takes the same steps as the sequential linked list, plus an initial locking phase transparent to the user, so it would be straightforward to reason with it.

Compared to the coarse-grained lock employed in the previous section, this fine-grained version has much better performance thanks to enabling disjoint access parallelism. As a reminder, a resource is said to allow disjoint access parallelism if it can be shared at the same time by several threads, so long as they require access to different parts of it. In our case, the linked list would enable disjoint access parallelism because two threads wanting to perform operations on different sets of nodes would not interfere with each other, exploiting the benefits of concurrency. The worst case scenario for performance would happen when two threads performed calls to operations sharing a node, at the same time. In this case, as we saw, one of the calls would fail and have to retry, so we would be forcing operations to occur sequentially. This is precisely the performance of the coarse-grained scenario we described earlier.

All in all, locking can provide solutions whose computational complexity is excellent, because they enable us to exploit concurrency thanks to allowing disjoint access parallelism. This means unused parts of the shared resource can be utilized while idle, without interfering with the modification of other parts at the same time. This is where real performance boosts can occur. On the other hand, in the worst-case-scenario these solutions lead to a refereeing of operations that lets threads know whether they have succeeded or they have to retry, in order to force a sequential ordering of operations.

[1]The word "retrying" used here needs to be taken with a grain of salt. When we say threads have to retry an operation, the reader should not simply interpret it as a need for a call to the same operation. Instead, we are talking about the need to carry out a new set of steps that involve reattempting the operation that failed. But these steps might require multiple sub-steps of their own to achieve their goal. For example, an operation that failed might require starting the whole operation from scratch, and not only the step that failed to obtain the lock.

```
1   bool CAS(L, E, N) {
2       atomic{
3           if(*L == E) {
4               *L = N;
5               return true;
6           } else {
7               return false;
8           }
9       }
10  }
```

Figure 3.6: The semantics of the `CAS` operation. The *atomic* keyword requires the block it labels to be executed *atomically*. Figure taken from (Moir and Shavit, 2004).

Nonetheless, locking is a costly mechanism in terms of memory in some programming languages, like `C++`. In others like Java, the overhead is not so high because every object has an associated Monitor that can be used to lock access to the object, irregardless of whether it is used or not. In any case, our linked list requires one lock per node, and locks are software-supported data structures. We now aim at a more lightweight solution that can run closer to hardware for even better performance.

3.2. The `CAS` Primitive

The first hardware-supported primitive we know of is `CAS` (Compare And Swap) as described in Figure 3.6. `CAS(L, E, N)` *atomically* checks if the content of location `L` matches the expected value `E`, modifying `L` to contain the new value `N` if it is the case, and otherwise not performing any operation. We could use the `CAS` primitive in order to achive a lightweight linked list implementation, because this primitive can be directly implemented as a hardware primitive, in contrast to software-supported locking. It is such a useful tool in concurrency that all modern architectures provide hardware support for it.

In contrast to locking, where we were guaranteed freedom of modification in any way we wanted so long as we were holding the lock. `CAS` introduces a new way of thinking because it forces us to reduce all operations to be performed on a data structure into swaps of memory content. As we will soon see, this is the reason why sometimes `CAS` needs to be complemented with other synchronisation tools, in order to provide a correct implementation of a concurrent data structure.

As an example, imagine we want to delete an element in a linked list like the one in Figure 3.7, initially consisting of three elements: a, b and d. Let us suppose we want to delete b. One might think that we can simply use `CAS` to modify a's `next_pointer` field to store d's memory address, so b is no longer part of the list. The same goes for insertions. If we wanted to insert an element after node b, however, we could simply modify b's `next_pointer` field to point to the new element, and the new element to point to d.

But what would happen if both calls occurred concurrently? This is what is

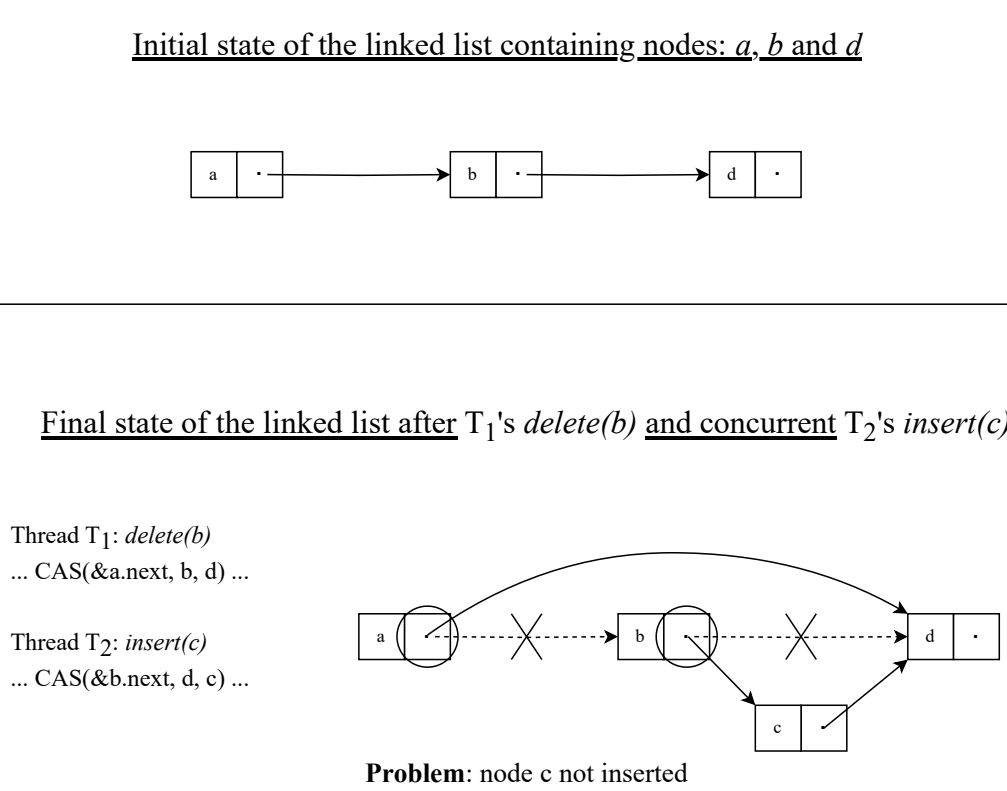


Figure 3.7: Illustration of failed concurrent operations performed by two threads: T_1 and T_2 on a linked list initially containing three nodes: a, b and d, that uses as only synchronisation tool single-location `CAS` operations. Figure inspired by those provided in (Luchangco et al., 2008).

depicted in Figure 3.7. `CAS` guarantees atomicity on the check to see if the expected value is still in the provided location, and on the modification towards the new value, if the check was successful. But two calls to `CAS` on different locations can proceed independently, atomically, having an undesired outcome because they conceptually depend on each other, but `CAS` is oblivious to it since it just works with single locations. In Figure 3.7 we can see that T_1's deletion of b has caused `a->next_pointer` to point to d, but at the same time T_2 has performed an insertion of c between b and d, resulting in `b->next_pointer` pointing to node c. And so we get a final result that does not correspond to applying either operation to the initial list, nor to applying both to it. The result does not even qualify as a proper linked list.

The problems we have encountered are due to not having made sure that the predecessor and successor nodes of the protagonist node in the operation remained immutable while the element was being deleted or inserted. This is the shortfall of `CAS`: it acts upon a single location. Atomicity is costly because it requires imposing access restrictions to shared resources, so we have to aim towards obtaining an efficient primitive providing some atomicity guarantees but not being limited in applicability. For further illustration of our point, we can take a look at another example over a linked list. Figure 3.8 depicts the failure of two concurrent deletions over a list initially containing four nodes: a, b, c and d.

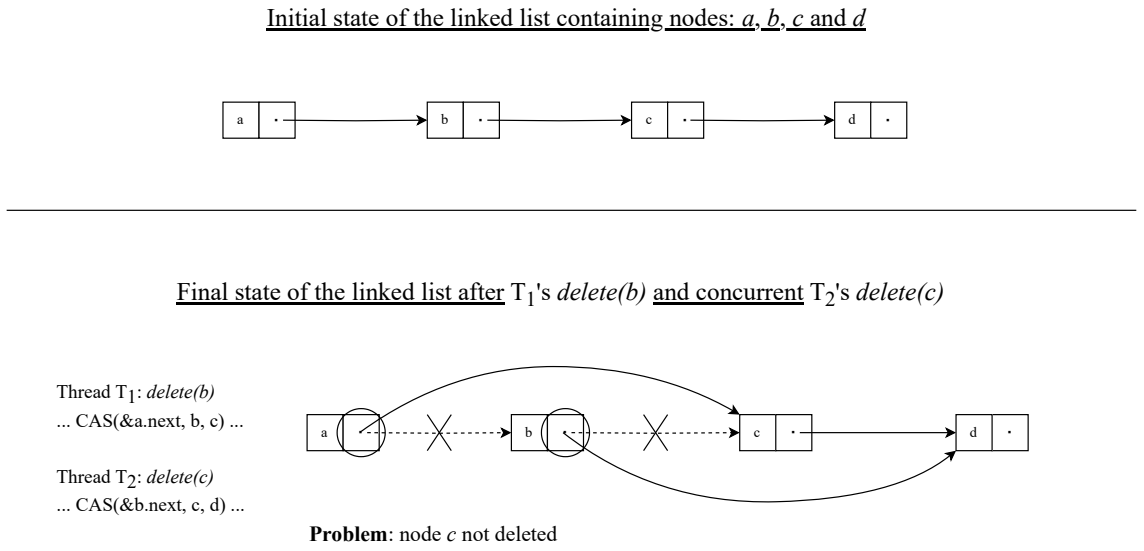


Figure 3.8: Illustration of failed concurrent deletions performed by threads T_1 and T_2, over a linked list initially containing four nodes: a, b, c and d. Figure inspired by those provided in (Luchangco et al., 2008).

We have just seen that a naïve use of single-location `CAS` primitives leaves ground for mistakes when designing linked concurrent data structures. We will now see how a smarter combination of synchronisation tools (including `CAS`), altogether with efficient memory management techniques, can provide a primitive with all the benefits of `CAS`'s memory-efficiency, while allowing disjoint access parallelism.

3.3. Non-blocking K-Compare-Single-Swap (`KCSS`) Operation

Victor Luchangco, Mark Moir and Nir Shavit (Luchangco et al., 2008) have devised an *atomic* and *obstruction-free* software synchronization operation called *K-location-compare single-location-swap* (`KCSS`) that precisely overcomes the difficulties of pointer manipulation in linked data structures with support for concurrency. `KCSS` allows us to modify a single location in memory while ensuring that K other locations remain unchanged. As we will later explain, `KCSS` is a software-supported operation that has the advantage of being more lightweight than general purpose transactional-memory based solutions, it provides disjoint access parallelism, and it is user-friendly because it can be used transparently, like any other software operation call. This contrasts with the opacity of code solutions that utilize complex synchronisation primitives interspersed in the code and therefore require careful concurrent reasoning to ensure correctness. Finally, `KCSS` is an interesting operation because it only requires a small constant memory overhead per word involved in the operation (in contrast to other software-supported operations).

We will now look at a practical use of `KCSS` to implement a linked data structure.

```
Node {
 Element elem;
 Count count;
 Node* next_pointer;
}
```

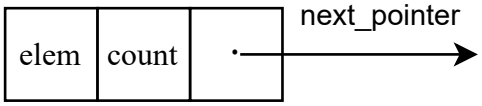


Figure 3.9: Illustration of a Node structure in the multiset implementation.

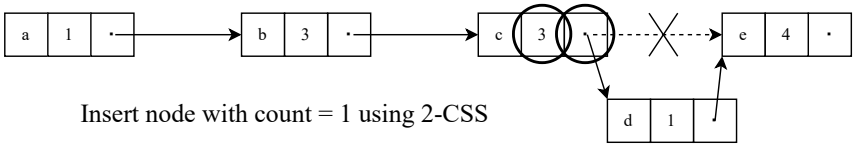


Figure 3.10: Illustration of an insertion over a KCSS based multiset implementation, when the inserted element was not a member of the multiset before.

To demonstrate its potential upon designing concurrent data structures, we will describe its behaviour when applied to a concurrent multiset[2] implementation based on a linked list. This particular implementation choice will evidence why we have reduced the study of "linked data structures" to general linked lists: because we are implementing a variation of a basic linked list.

In this particular case, we design a multiset made up of a linked list of nodes (following the format that can be seen on Figure 3.9), where every node represents one distinct element on the multiset, storing its non-zero multiplicity together with the element value, and a pointer to the next element on the linked list. Because of this, we can already point out the similarities of this multiset to the linked lists we talked about in previous sections: we will encounter similar problems of consistency upon manipulation of the linked list of nodes. The multiset, however, poses the extra challenge of having to manage more fields per node (i.e., the multiplicity field).

In particular, to implement a multiset we need 2-CSS (KCSS instance where K = 2) for insertions, and 4-CSS (KCSS instance where K = 4) for deletions, and search operations. The behaviour of these constructs is as follows (see Figures 3.10, 3.11, 3.12 for illustration):

`search(Element x):` This method is an auxiliary procedure that searches whether the multiset contains any appearance of element x. To do this, it has to check whether any node on the linked list making up the multiset stores a key with value x, and non-zero multiplicity. In particular, `search(x)` returns two adjacent nodes (N1, N2) (i.e. in the list of nodes, the `next_pointer` field of node N1 points to N2) such that N1's key is smaller than x, and N2's key is greater or equal to x. The correctness of this method comes from it guaranteeing to never return a node whose multiplicity is 0. This can be achieved by deleting all nodes found while

[2]As a clarification, when we talk about "multiset" we refer to the mathematical concept: *A generalization of the notion of mathematical set, which allows duplicate values, unlike sets.*

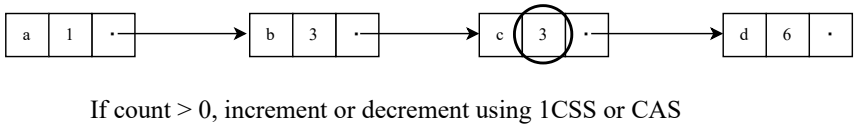


Figure 3.11: Illustration of an insertion over a `KCSS` based multiset implementation, if there was already an appearance of the inserted element on the multiset ($count > 1$).

traversing the list whose multiplicity is 0, making use of 4-CSS. To avoid getting into implementation details at this early stage of introducing the reader to the `KCSS` software operation, we will not explain how exactly 4-CSS is used in the `search` operation.

`insert(Element x):` This method inserts a node whose element is `x`. The first action `insert` will have to do is to search `x` in the multiset. In case `x` was already present on the multiset, we can simply use 1-CSS or `CAS` (their behaviour is equivalent, as we will later see) to increment the `count` field of `x`'s node on the multiset. On the other hand, in the case where we are adding `x` to the multiset for the first time, (with multiplicity 1), we have to carefully manipulate the node that will be predecessor to the new node containing element `x` (we will call this node `node_x`, and its predecessor `predecessor`). To insert `node_x` we have to modify `predecessor`'s pointer to point to `node_x`, while ensuring that no other thread is attempting to change the `predecessor->next_pointer`, nor its `multiplicity`. This is because if another method attempts to decrement `predecessor`'s `multiplicity` to 0, while we are performing `insert(x)`, the `predecessor` node would be deleted during this call to `insert`, which is making use of it. Hence the need for 2-CSS on these 2 locations: `predecessor->next_pointer`, `predecessor->multiplicity`. 2-CSS will modify the content of `predecessor->next_pointer` to set it to `node_x`'s address, guaranteeing in the process that both locations maintain the values they held when the call to 2-CSS was made.

`delete(Element x):` This method decrease the multiplicity of `x`, if there is a node containing it. `delete` will initially search for nodes containing element `x` in the multiset. If one such node is found, and its multiplicity is greater than 1, we can again use 1-CSS or `CAS` to decrement the `count` field of `x`'s node. On the contrary, if a node (let's call it `node_x`) is found to contain `x` and its multiplicity is 1, `node_x` has to be removed from the multiset (we do not allow non-zero multiplicities). The search operation will have provided us with `node_x`'s predecessor (`predecessor`) and successor (`successor`). The `delete(x)` operation has to modify `predecessor->next_pointer`, to point to `successor`. In the meantime, the following 4 locations have to remain unchanged:

- `predecessor->next_pointer`: modifying it is the main purpose of `delete(x)`, and if another thread tries to do so at the same time would cause incorrect behaviour.

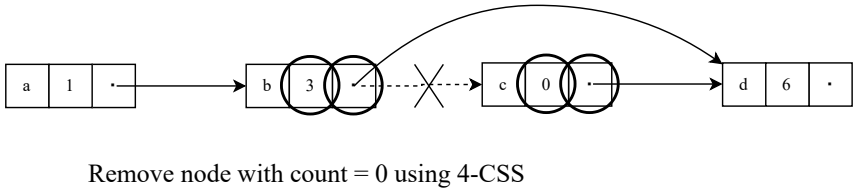


Figure 3.12: Illustration of a deletion over a KCSS based multiset implementation. This operation occurs when the element's count field becomes 0.

- predecessor->multiplicity: it has to remain immutable for the same reason as in insert. Because a concurrent attempt to change it could decrement it to 0, triggering its deletion while this delete(x) operation is using it.
- node_x->next_pointer and node_x->multiplicity: node_x cannot be modified at all concurrently to this operation, because it is being removed. It should appear unreachable to other threads, and so should its fields.

Consequently, we apply 4-CSS on these 4 locations to set the content of of location predecessor->next_pointer to be the pointer to successor – that is, deleting node_x – ensuring all of the above described locations are safe to do so.

As we have seen, KCSS precisely overcomes the shortfalls encountered when using locks, because it allows disjoint access parallelism thanks to being a minimal-effect operation acting on very specific memory areas every time it is used. It also generalizes the applicability of single-location CAS, by providing a similar behaviour upon more than one location. Further on, we will also show how it achieves memory-efficiency, which makes it usable on platforms with different capabilities.

Chapter 4

Non-blocking K-Compare-Single-Swap (KCSS)

Now that we have understood the applicability of the KCSS operation, we will present the tools and requirements necessary to implement it, following what is suggested in (Luchangco et al., 2008). At the end of this section, the reader will have fully understood: the scenarios where KCSS can be used, as well as those where it is most advantageous; the semantics of KCSS and how it can be properly employed.

4.1. System Model

We assume a machine architecture with the following characteristics:

- A 64bit word architecture.
- Hardware support for the CAS operation on memory words.
- Some assumptions on the memory model, that we will discuss later on Section 4.7 (they have to do with the effect of storing and reading values on memory, and we prefer to delay this discussion for simplifying the presentation).

4.2. Formal Specification

KCSS (with $K > 1$, a natural number) is a software operation that takes as parameters:

- The addresses of K different memory locations $a_1, ..., a_k$.
- The expected values of those K locations $e_1, ..., e_k$.

- A new value n_1 that we want to place into the location a_1.

Therefore, a KCSS call has the following form:

$$\texttt{KCSS}([a_1, ..., a_k],\ [e_1, ..., e_k],\ n_1)$$

and its semantics is as follow: it checks that every location a_i contains its expected value e_i, for $1 \leq i \leq K$. If this is the case, then it also updates location a_1 with the new value n_1, and returns *true*. Otherwise, it does not modify any memory location and it returns *false*. We say KCSS either *succeeds* or *fails* in each of these cases.

KCSS is *obstruction-free*. As we have already seen in Section 2.2, this means that threads running concurrently are only guaranteed to make progress if they are left to run on their own for enough time.

4.3. Memory Locations and their Values

KCSS's performance largely depends on the format restrictions it imposes on the memory locations involved in it, and the way it manages the values that these locations can contain. In this section, we will not discuss implementation details of the data structures used by KCSS, but rather give only those necessary to explain the big picture of the mechanism behind it.

For KCSS, a memory location is a 64bit word in memory, just like it is for the system in which it runs. However, KCSS' implementation very much relies on identifying the last thread that has accessed a location. In order to do so, we use a mechanism consisting on leaving a timestamp of the thread that last accessed the location, every time it is accessed with the intention of writing to it (note that *accessing a location with the intention of writing to it* in this case means reading the location's value, performing some operation with it, and then updating the the location). This timestamp is the thread's identifier (which we will refer to as $thread_{ID}$ from now on), and it is stored in the location itself. This location could have contained a normal value such as the value types we know (integer, floating point, pointer, ...), or it could have already contained the timestamp ($thread_{ID}$) of the previous thread that had attempted to start writing to the location and had not yet finished. These two possibilities allow us to differentiate whether another thread had already accessed this location or not, and it is key to our implementation. In addition, we have a mechanism to restore a location to its original value if a writing operation to it fails before completion.

If a thread wants to modify a location α, it first replaces α's current value with its own $thread_{ID}$ (plus other metadata that we will explain later). Then, it saves the value that it had found on α into a special area of shared memory (an array called SAVED_VAL), so that other threads can refer to it in the future if they find that α contains the timestamp of a thread that did not manage to finish writing to

it, i.e., to be able to reverts α's content to the original value. Let us now see this mechanism in detail.

We distinguish two types of values that memory locations can contain: program values (integer, floating point, pointer, ...), and temporal values (the ones we have been referring to as "timestamps"). For a visual illustration and an example of how locations and their values are organised, see Figure 4.1.

- *Program values*: Locations with this value type contain an actual program value (such as an integer, floating point value, a pointer, ...) that has been adapted to fit in 63 bits[1], plus a single bit (the least significant bit) **set to 0** which is reserved to distinguish them from temporal values. Reserving this bit, in practice means that we have to sacrifice 1 bit of precision for normal primitive types. For pointers, however, since we cannot sacrifice precision without changing the pointer's value, we simply use aligned memory positions on even positions for all of the program's pointers. This might seem restrictive but in reality, the vast majority of modern systems already use word-aligned memory positions anyway.
- *Temporal values*: Locations with this value type contain a pair

  $$\langle thread_{ID}, tag_{number} \rangle$$

  where $thread_{ID}$ occupies 15bits, and tag_{number} 48bits[2]. This amounts to a total of 63bits, because once again, the least-significant bit is required to distinguish temporal values from program values. In this case, it will be **set to 1**. Temporal values represent the information related to the last access to this location: $thread_{ID}$ represents the thread that last accessed it, and tag_{number} will be used to distinguish different accesses by the same thread to the same location.

As we have seen, a memory location α's *program value* is temporarily stored in a special location in shared memory that is not the address of α. This special location is `SAVED_VAL`), an array of *program values* that stores the last *program value* that α has contained. `SAVED_VAL` is indexed by thread identifiers, which are unique identifiers given to every thread in the system.

Now that we have understood some key data structures used by `KCSS` to represent information, we will move on to explaining the first and most important components

[1]For now, we will not get into the details of how a "cast" from any program value to a type containing 63bits (instead of 64bits) is possible. The implementation details will be explained in Section 5.1.

[2]The bit choice for fields $thread_{ID}$ and tag_{number} is merely a design choice. In this case, 15bits have been allocated for $thread_{ID}$s because we assume this is enough to ensure unique IDs for all virtual threads that could exist. 48bits have been allocated for tag_{number} because we assume this provides tags in a range big enough to guarantee no wraparound.

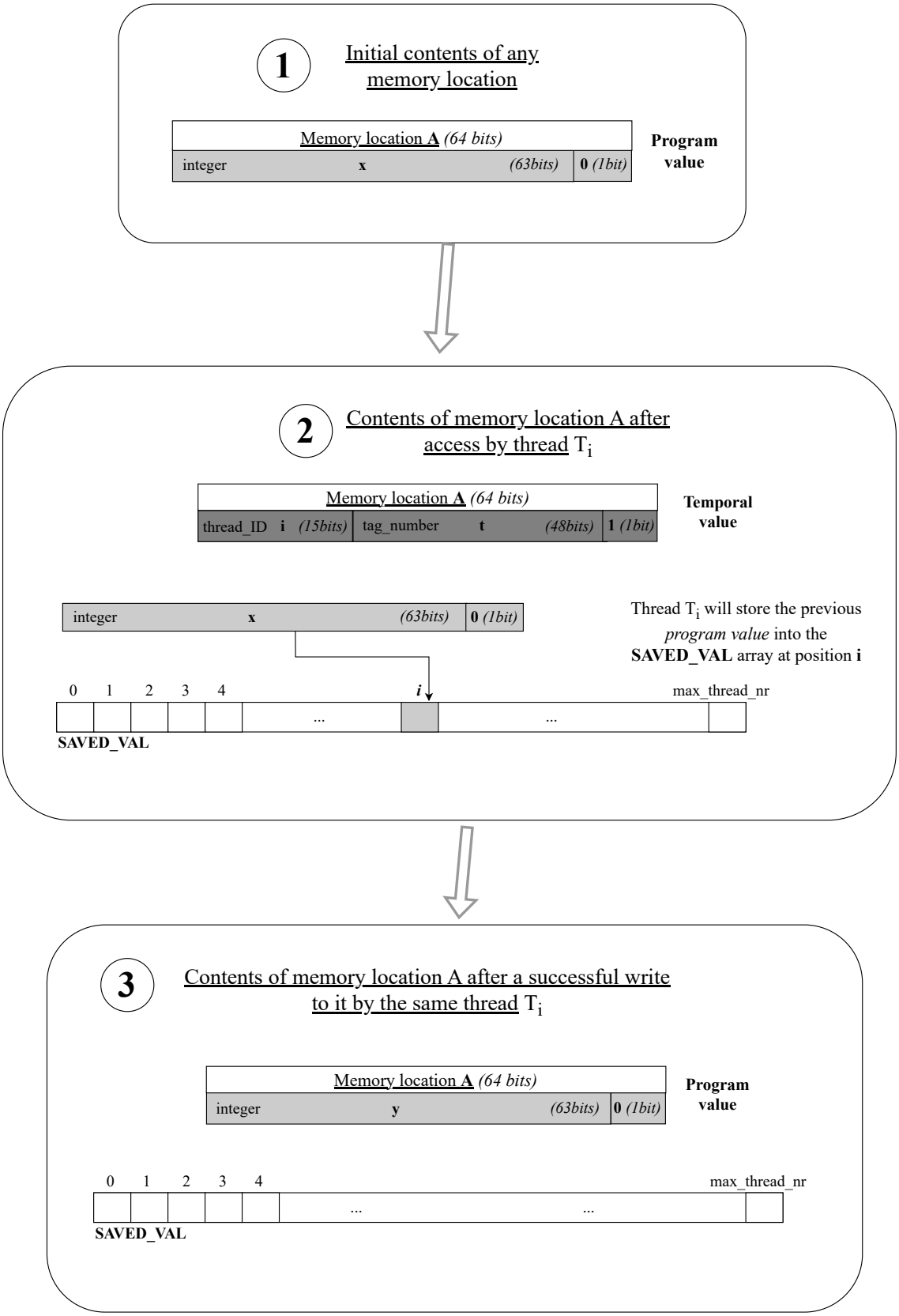


Figure 4.1: Illustration of the bit-wise organisation of *program values* and *temporal values*, in the context of the modification of a location "A" by thread T_i.

```
1 struct loc_struct {
2   uint64_t snapshot_ts; // used for SNAPSHOT
3   uint64_t value;
4 };
```

Figure 4.2: Pseudo-code of the `loc_struct` structure.

of KCSS that make use of them: LL, SC and READ operations. In the following section we will introduce some pseudo-code to explain the aforementioned operations.

Let us define a structure called `loc_struct`, which is depicted on Figure 4.2 (the type `uint64_t` is used to represent a word). The memory location layout that we have just explained, referring to *temporal* and *program values*, will use the field `value`. The structure has another field `snapshot_ts`, which we ignore for now, and explain later when needed for an operation called SNAPSHOT.

4.4. Load Linked (LL) and Store Conditional (SC) Operations

The need to distinguish *program values* from *temporal values* comes from the specific implementation requirements of the Load Linked and Store Conditional operations (LL and SC, respectively[3]) required by KCSS. As a reminder (see Section 2.2), these primitives enable us to: **atomically** read the value of a location α through LL, perform some action using the value, and then use SC to commit a new value only if no-one else has modified α between this thread's LL and SC operation calls.

Recall that the hardware implementation of LL/SC, that we have seen in Section 2.2, required using "link registers" to ensure that no other LL call has been performed between a thread's calls to LL and SC. KCSS is a software primitive, and because of this, it uses software (or virtual) threads, instead of hardware-equivalent constructs. This means, LL and SC now have to achieve synchronisation among a possibly unbounded or not fixed number of threads (varies according to the system's execution capabilities at different points in time), which is is why we cannot make use of hardware implementations of LL and SC. We can no longer use the "link register" as an auxiliary register for communication among threads, because it is not feasible to assign a link register to every virtual thread both in terms of hardware resources (we are not guaranteed to have enough available registers for this purpose if the number of threads is undetermined), and in terms of performance (an unbounded number of threads can generate an unbounded number of collisions when performing LL/SC, which would cause an unbounded number of resetting operations on each other's link register).

To implement KCSS, we have to provide the same functionality as that of conventional LL and SC, but supporting synchronisation among a possibly unbounded number of **virtual** threads. The way KCSS does this is by using *program values* and

[3]From now on, we will simply refer to *Load Linked* and *Store Conditional* as LL and SC.

temporal values for the same purpose: it places a *temporal value* into a location α to indicate that a thread has performed an LL operation on it and has yet to perform SC to commit its changes (after this happens, α will again contain a *program value*). Therefore, the algorithm that provides KCSS's functionality requires a software-supported novel implementation of LL and SC. Next we explain how this is achieved.

Firstly, the mechanism employed by the new LL and SC software versions is the following:

1. Recall that LL and SC work as a pair. An SC(α, ν) call by a thread T (which intends to modify location α's contents to put value ν into it) will only be successful if T has previously executed an LL(α) on the same location α. We will say an LL(α) operation invoked by T is *outstanding* until T invokes SC(α, ν) on the same location α.

2. There is also a restriction on the success of SC: If a thread T makes a call to LL(α), followed by a call by T to SC(α, ν), the SC operation only succeeds if no other LL or SC operations take place on location α between the invocation of T's LL, until the completion of its call to SC ("no other LL or SC operations" means either performed by thread T or others). All of this still holds in our novel implementation.

The following example walks through a possible run of LL/SC.

Example 4.4.1 (A possible run of the LL/SC Operation) *Consider the illustration depicted in Figure 4.3. Threads T_1, T_2 and T_3 attempt various LL and SC operations on two locations **A** and **B**. Initially, T_1 succeeds on performing a call to LL and another one to SC on **A**, because both threads T_2 and T_3 do not attempt any operation **on that location** at the same time. In contrast, when T_2 performs an LL(B), followed by an SC(B, ...) call, this one fails because T_3 has performed an LL operation to **B** while T_2's LL was outstanding. The same thing happens for T_3's SC: it fails because T_2 performs an LL(B) operation between T_3's invocation to LL and SC.*

When a thread T performs an LL operation on a location α, LL reads the content of α, saves it in the SAVED_VAL array, and then attempts to place a *temporal value* containing T's $thread_{ID}$ and corresponding tag_{number} into α . This symbolizes that thread T has an *outstanding* LL operation on this location without requiring a linked register like in the well-known LL/SC implementation of Section 2.2. See Figure 4.4 to follow the explanation together with the pseudo-code for this operation. The READ operation reads the content of a location α, and returns the *program value* of α if α contains a value of this type. Otherwise, if α contains a *temporal value*, READ resets α's content, placing back into α the previous *program value* it contained (which can be found in the SAVED_VAL array we saw earlier). All of this will be explained later in more detail.

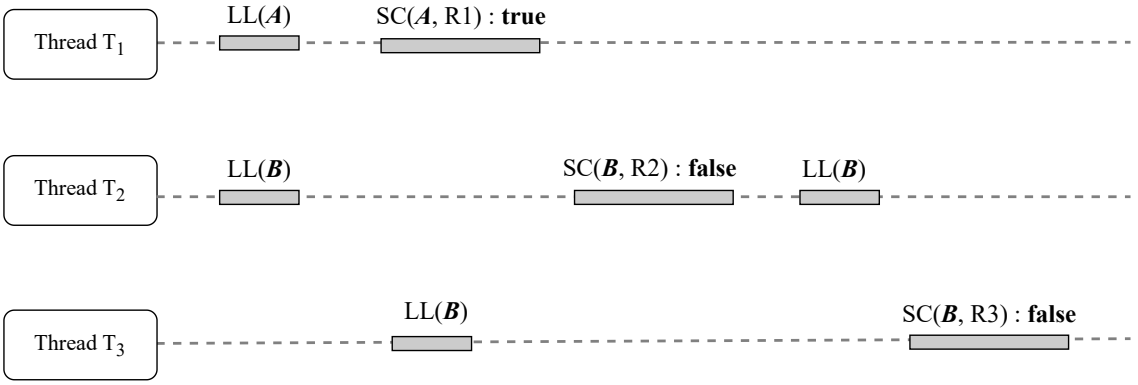


Figure 4.3: Time diagram illustrating the use of KCSS's LL/SC operations by three threads: T_1, T_2 and T_3. Notice T_1's call to LL followed by SC on location **A** succeeds because Threads T_2 and T_3 do not attempt any operation on that location. T_2's SC call fails because T_3 has performed an LL operation to the same location (**B**) while T_2's LL was *outstanding*. The same thing happens for T_3's SC: it fails because T_2 performs an LL(B) operation between T_3's invocation to LL and SC.

```
1  uint64_t LL(loc_struct* a) {
2    while (true) {
3      my_tag_number++;
4      uint64_t oldValue = READ(a);
5      SAVED_VAL[my_thread_ID] = oldValue;
6      uint64_t temporal_val =
7        make_TemporalValue(my_thread_ID, my_tag_number);
8      if (CAS(&a->value, oldValue, temporal_val)) {
9        a->snapshot_ts = temporal_val; // needed for SNAPSHOT
10       return oldValue;
11     }
12   }
13 }
```

Figure 4.4: Pseudo-code of the LL operation.

Further on, T performs an SC(α, ν) which will attempt to write ν to location α, with the condition that α still holds the *temporal value* made up of T's $thread_{ID}$ and corresponding tag_{number} that T had placed in it. If this is not the case, which means that some thread performed an LL on the same location α, SC will fail and perform no operation. See Figure 4.5 to follow the explanation together with the pseudo-code.

```
1 bool SC(loc_struct* a, uint64_t newValue) {
2   uint64_t new_program_val = make_ProgramValue(newValue);
3   uint64_t temporal_val =
4     make_TemporalValue(my_thread_ID, my_tag_number);
5   return CAS(&a->value, temporal_val, new_program_val);
6 }
```

Figure 4.5: Pseudo-code of the SC operation.

We have just described in a broad way how LL/SC work, to help the reader understand the big picture. We will now give an in-depth explanation. Let us first see an example, which we will use to illustrate the behaviour of LL/SC.

Example 4.4.2 (Illustrating the behaviour of the LL/SC) *Consider the illustrations depicted in Figure 4.6. Suppose the initial content of some location, α is a program value $\boldsymbol{x}$. Thread T, with ID $\boldsymbol{i}$, wants to perform an LL/SC operation that will place value $\boldsymbol{y}$ into α. For that, it will first perform an LL operation on α, whose outcome corresponds to the one depicted in step 2 of Figure 4.6. Array SAVED_VAL will now contain an entry at index $\boldsymbol{i}$ storing α's previous program value. α's content will have been replaced by a temporal value generated by thread T using its ID and tag_{number}, $\boldsymbol{t}$. Following LL's actions, T will call SC(α, y), which will finally check whether α still holds the temporal value it had placed there during LL, and substitute the content of α by program value $\boldsymbol{y}$. This concludes a successful run of LL/SC.*

Let us now see the implementation details of this process.

General environment notes: each thread has two thread local variables: my_thread_ID is an integer value unique to each thread that never changes (a thread's $thread_{ID}$); and my_tag_number is an integer value that is incremented by one unit every time a thread performs an LL operation (see Line 3 in Figure 4.4), in order to store as content a new *temporal value* containing a fresh tag_{number} every time.

The need to have unique thread identifiers serves the purpose of identifying which thread was the last one to attempt an LL on a specific location. Since LL replaces a location's content with a *temporal value* containing the thread's $thread_{ID}$, this new information allows the rest of threads to know that some other thread has an *outstanding* LL operation on this location. Furthermore, when a thread reaches step 3 (see Figure 4.6) in the LL/SC sequence of operations, that is, the SC phase, by checking the $thread_{ID}$ it will be able to know whether some other thread has performed an LL on that same location in between its own LL and SC (and therefore its SC has to fail) or whether the SC can be successful because its own ID still holds. The need for tag_{number}s, however, is to ensure that the **same** thread can distinguish different versions of its own LL operations on the same location. We will illustrate the problem through an example.

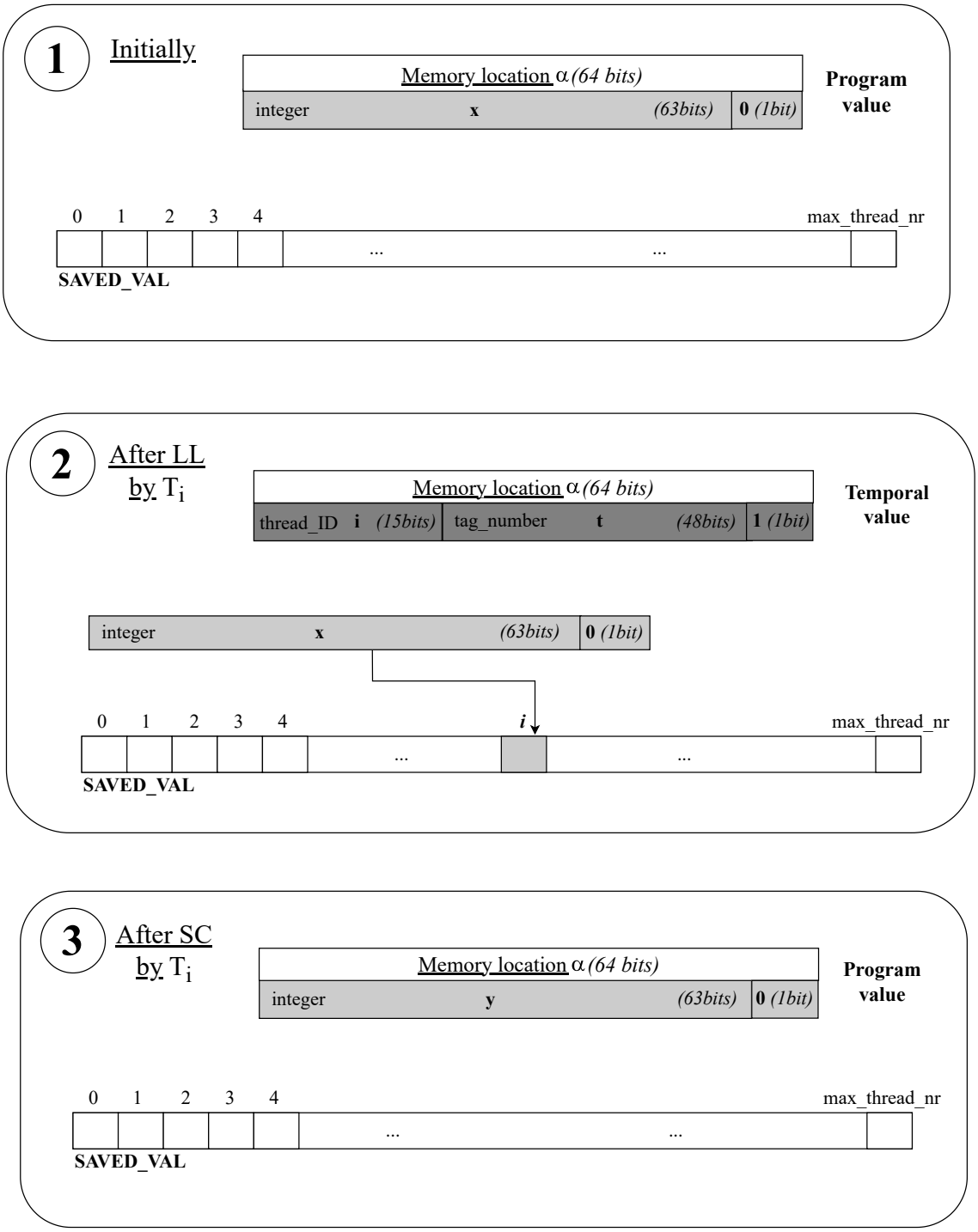


Figure 4.6: Illustration of the 3 steps in an LL/SC operation by thread with ID **i**: T_i, wanting to place value **y** in memory location α.

Example 4.4.3 (Motivating the need for tags: without tags) *In this example we illustrate the problem that we run into when using only thread identifiers as temporal values, i.e., without the tags. Suppose we have the situation depicted in Figure 4.7, where two threads are attempting* `LL/SC` *operations on the same location* $\boldsymbol{B}$*, and thread* T_1 *is alternating between the execution of two tasks (one executes for some time, then it is interrupted by the other one, which runs in the meantime, and so on). Location* $\boldsymbol{B}$ *initially contains a program value* ***oldValue****.* T_1*'s* $Task_1$ *is running, and performs an* `LL` *first, placing into* $\boldsymbol{B}$ *its* $thread_{ID}$ T_1*. Then comes* T_2 *and does the same, so* $\boldsymbol{B}$ *will contain a temporal value made up of* T_2*'s* $thread_{ID}$ *after the first two* `LL` *operations. Immediately after its* `LL`*,* T_2 *performs an* ***SC(B, oldValue + y)****, which succeeds because no other operation to* $\boldsymbol{B}$ *took place between this call and* T_2*'s* `LL` *call. Therefore, the current value of location* $\boldsymbol{B}$ *at this stage is a program value: integer* ***oldValue + y****.*

Until now everything has worked fine. After some time, T_1*'s* $Task_2$ *is allowed to run (interrupting* $Task_1$*).* T_1*'s* $Task_2$*, oblivious to* $Task_1$*'s actions, performs a new* `LL` *on location* $\boldsymbol{B}$*, placing in it a new temporal value with* $thread_{ID}$ T_1*. As we can see in the figure, this new temporal value is indistinguishable from the one put into* $\boldsymbol{B}$ *by* T_1 *at the very beginning. Thanks to having put a fresh tag number into* $\boldsymbol{B}$ *through* `LL`*,* $Task_2$ *will be aware that the program value of* $\boldsymbol{B}$ *has changed to* ***oldValue + y****, but we cannot say the same for* $Task_1$*.* $Task_1$ *wanted to use the last value of* $\boldsymbol{B}$ *to add a quantity* $\boldsymbol{x}$ *to it. Because* $Task_1$*'s* `LL` *happened before Thread* T_2 *changed* $\boldsymbol{B}$*'s value to* ***oldValue + y****,* $Task_1$*'s execution context thinks that the value in* $\boldsymbol{B}$ *is still* ***oldValue****. Not only this, but when* $Task_2$ *gives way to* $Task_1$ *again,* $Task_1$ *will call* `SC` *to commit its operation of modification of* ***oldValue*** *to be* ***oldValue + x****. Because* $Task_2$ *placed Thread* T_1*'s* $thread_{ID}$ *into* $\boldsymbol{B}$*,* $Task_1$ *thinks no-one else has accessed* $\boldsymbol{B}$ *since its* `LL`*, and therefore it thinks that* $\boldsymbol{B}$ *has contained value* ***oldValue*** *all along, between* $Task_1$*'s* `LL` *and* $Task_1$*'s* `SC`*. Therefore,* $Task_1$*'s* `SC` *will succeed, placing an incorrect value on* $\boldsymbol{B}$ *(incorrect because it ignores Thread* T_2*'s changes to* $\boldsymbol{B}$*).*

Example 4.4.4 (Motivating the need for tags: with tags.) *Consider the illustration depicted in Figure 4.8. The same sequence of actions takes place, except that temporal values contain unique tags every time an* `LL` *is performed. The use of tags allows Thread* T_1*'s* $Task_1$ *to know that* $\boldsymbol{B}$*'s content correspond to a version of a temporal value that was placed by a call by* T_1 *to* `LL` *that does not correspond to the one* $Task_1$ *had used to read the value of* $\boldsymbol{B}$ *and operate with it. Therefore, when* $Task_1$ *attempts to perform an* `SC`*, it fails. This is the correct behaviour. Probably,* $Task_1$ *will have to retry* `LL/SC` *again in order to see the latest value of* $\boldsymbol{B}$ *placed by Thread* T_2*.*

The problem we have just described is a well known concurrency issue known as the ABA problem. This is its formal definition:

> (Luchangco et al., 2008) The ABA problem arises when a thread reads a value A in a location, and later [...] attempts to change the location from A to a new value, with the intention that if any other thread writes to the

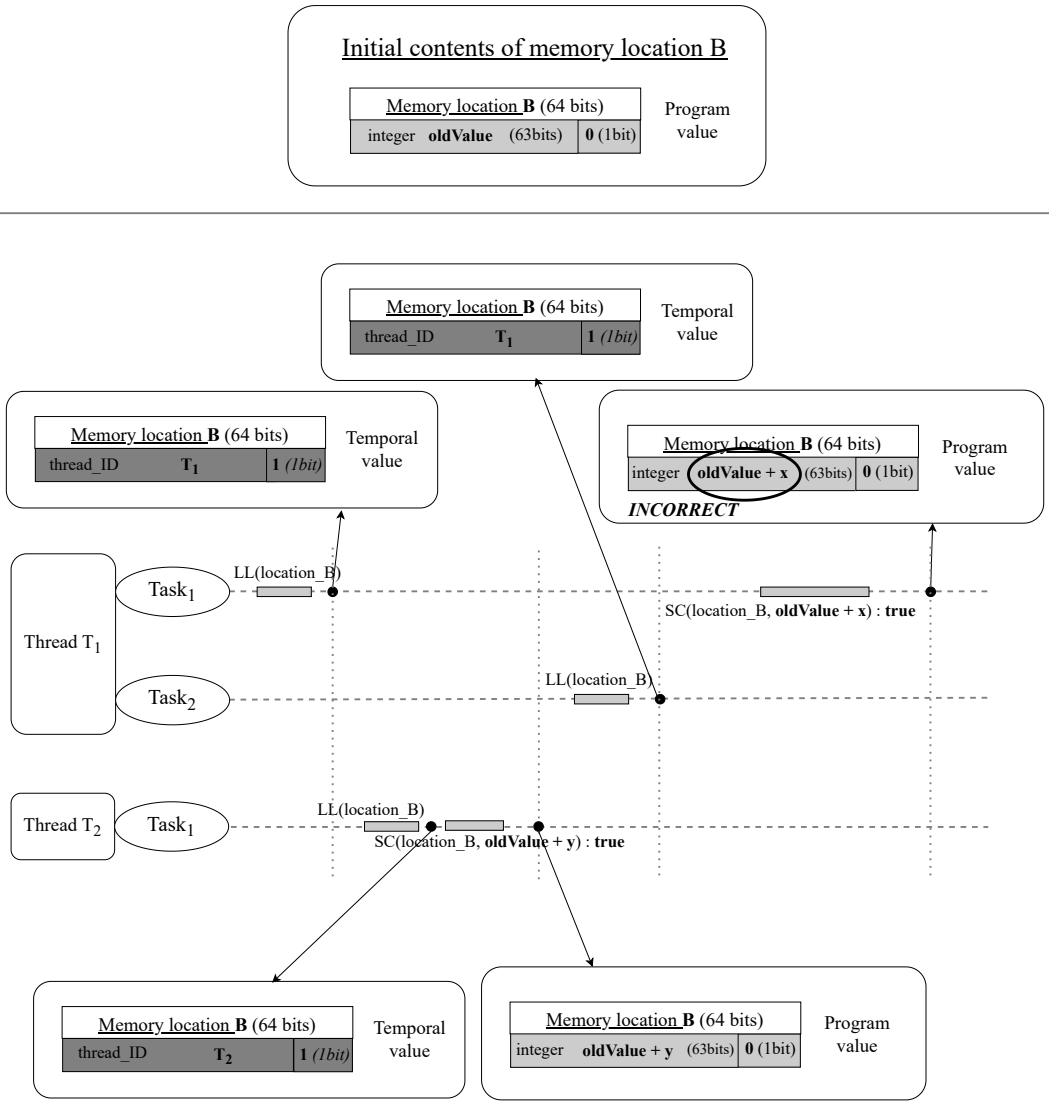
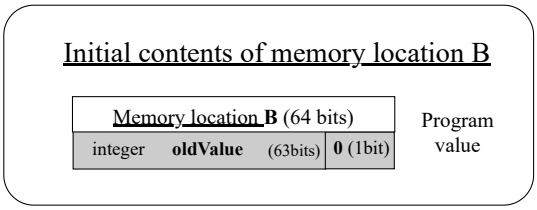


Figure 4.7: Time diagram illustrating the problem derived from an LL/SC implementation that does not rely on tags to check for different versions of LL operations performed by the same thread. In this case, T_1 is oblivious to T_2's modifications to a location **B** because T_1 is alternating between executing two tasks, $Task_1$ and $Task_2$, so T_1 performs an SC operation on **B** that succeeds even though it should not. This SC updates the value of **B** to be one that has ignored T_2's changes to **B** while T_1's $Task_1$ was not running. If, for example, **B** had stored the value of a counter, there would be some increments performed by T_2 that would have gotten lost in the process.



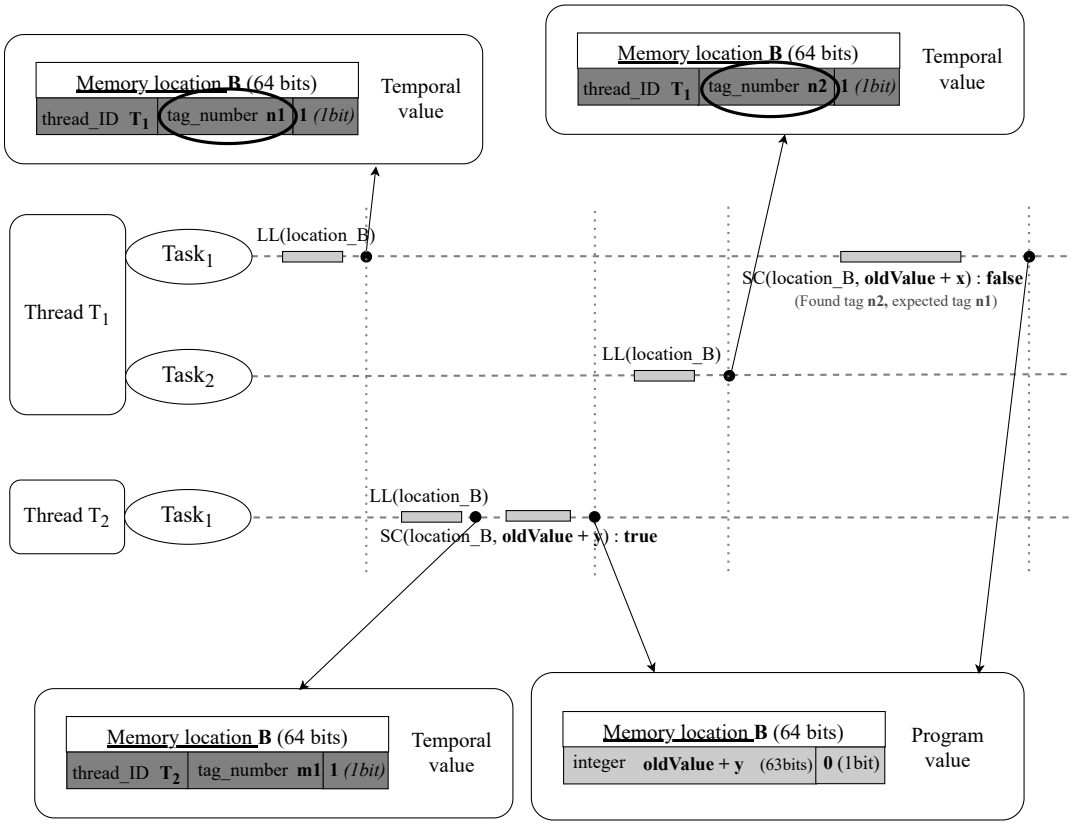


Figure 4.8: Time diagram illustrating how using tag numbers solves the problem depicted by Figure 4.7. In this case, T_1 has a means of checking that a new `LL` operation has happened between T_1 $Task_1$'s `LL` and its `SC`. Therefore, its `SC` fails, which is the expected behaviour.

> location between the read and the modification attempt, the modification will fail, leaving the value unchanged. However, in that interval, the value may change from A to some value B and then back to A again, in which case the modification operation will succeed. By using tags which are incremented every time the location is written, threads can avoid the ABA problem, provided the tags have enough bits to avoid wraparound in practice.

Thanks to storing simultaneously the $thread_{ID}$ and the tag_{number} when we perform LL operations, both the same thread and others can exactly know that, if a location's content is a *temporal value*, some thread must have an *outstanding* LL operation on it, and that thread is precisely identified by the $thread_{ID}$ at that location. Not only this, but a thread who finds a location's content storing a *temporal value* with its own $thread_{ID}$ can check whether its current tag_{number} matches the one found in that location. If it does match, no ABA problem occurred and it is the only thread with an *outstanding* LL operation on that location, otherwise, the ABA situation occurred and this thread can be aware and act accordingly. This mechanism for dealing with the ABA problem is a novelty, and it has been proposed by the authors at (Luchangco et al., 2008) for the first time.

Let us now go deeper into the details of the READ operation that we mentioned earlier. This operation (see Figure 4.9 for the pseudo-code) allows threads to determine the *program value* of a location. It works by repeatedly checking whether a location contains a *program value* or a *temporal value*, until a *program value* is found. If a *program value* is found, it will return it. If a *temporal value* is found, READ calls an auxiliary operation RESET that will replace the *temporal value* found with that location's previous *program value*, which can be retrieved from array SAVED_VAL (see in lines 1 to 6 for RESET's pseudo-code, in Figure 4.9). READ then restarts the loop, which will end if no other thread has performed an LL on the same location in between READ's start of the new iteration and its check of the location's content. This is the reason why we cannot simply return the content of the location upon performing READ. The call to RESET is necessary to prevents a thread's *outstanding* LL operation on it from committing through the corresponding SC, and this will be needed in order to support the SNAPSHOT and KCSS operations that will be presented later.

Now that we have described all the pieces that conform the LL/SC mechanism, let us go back to the pseudo-code of LL and SC and explain it in more detail.

LL pseudo-code in detail (Figure 4.4): An LL(α) operation is expected to retry until it is capable of returning the *program value* in α, which is why the first line of code in LL is a `while(true)` loop (Line 2). LL's loop repeatedly does the following: it first reads the content of α using READ. READ will return a *program value*, irregardless of whether α contained a *program value*, or it found a *temporal value* and RESET had to be called on α to restore it to its previous *program value*. LL will go on to save the returned *program value* into its allocated position in array

```
1  void RESET(loc_struct* a) {
2    uint64_t oldValue = a->value;
3    if (is_temporal_value(oldValue)) {
4      CAS(&a->value, oldValue, SAVED_VAL[thread_id(oldValue)]);
5    }
6  }
7
8  uint64_t READ(loc_struct* a) {
9    while (true) {
10     uint64_t val = a->value;
11     if (!is_temporal_value(val))
12       return val;
13     RESET(a);
14   }
15 }
```

Figure 4.9: Pseudo-code of the READ and RESET operations.

SAVED_VAL[4] so that if this LL succeeds, itself and other threads know what the last *program value* of α was. Then, it will generate the timestamp necessary for the creation of the *temporal value* that it will attempt to place in α to finalise its LL operation, using the currently executing thread's attributes $thread_{ID}$ and tag_{number} (recently incremented to ensure its uniqueness at Line 3). The placement attempt is done using CAS, to ensure it is an atomic operation. A successful CAS will mean the LL finished correctly and is now an *outstanding* LL for location α, and so it will return the *program value* found upon reading α. A failed CAS will mean some other thread managed to change the content of α somewhere in between the call to READ at Line 4 and Line 7. This is why LL has a `while(true)` loop. Just like when we saw locking mechanisms that required `try-lock()` to ensure a failure did not prevent them from progressing, this loop serves the same purpose.

SC pseudo-code in detail (Figure 4.5): A call to SC(α, newValue) by a thread T is a one-time shot at replacing the content of α with value newValue. SC consists of a CAS operation (Line 4) that checks whether α still holds the *temporal value* that T had placed there during the last LL that thread T performed, and replaces the content of α by a new *program value* newValue if that is the case. Otherwise, it simply fails and return. It does not retry because one failure makes it impossible to succeed in the future, unless the tag_{number} is incremented again (which can only happen if a new LL operation is performed by T).

[4]The reason why we can use array SAVED_VAL to store all of the possible previous *program values* replaced by threads' *temporal values*, is because we can have at most as many *outstanding* LL operations as threads there are, and because a single thread can have at most one *outstanding* LL operation in total.

4.5. SNAPSHOT Operation

Let us recall the purpose of KCSS: we want to check that K locations contain their expected values, and if this is the case update the contents of one of them with a new value. Until now, we have focused on the mechanism that enables us to read the content of a location, use it in some intermediate calculations, and then update the location with a new value if it has not been accessed in the meanwhile. All of this is possible using only LL/SC, and READ operations. However, in order to implement KCSS, the first step involved is clear: we have to retrieve the content of K locations, and then proceed to check if the values match the expected ones, and to update the corresponding location if possible. Therefore, we initially have to carry out K READ operations. Unfortunately, this step cannot be done by a simple a loop that reads all locations by calling READ(location_X), because every READ operation takes a non-negligible amount of time. This allows for time gaps between one READ(location_X) and another READ(location_Y) where other threads are concurrently executing and can potentially change the contents of location_X by the time READ(location_Y) finishes. This is a problem because what we really wants is to capture the state of K memory locations at a single point in time, and the values returned are only correct if we can guarantee that, indeed, they have all coexisted in their corresponding memory locations at some point in time. We illustrate this in the following example.

Example 4.5.1 (Motivating the need for the SNAPSHOT operation) *Consider the illustration depicted in Figure 4.10. Suppose we have two threads: T_1 and T_2, and two locations **A** and **B**. T_1 wants to collect the content of both locations, while T_2 is concurrently executing. T_1 goes ahead and performs a READ(A), retrieving value old_A from **A**. Now T_2 starts executing and performs a successful LL/SC call on **A**, changing its contents to new_A. While T_1 is still busy with something else, T_2 keeps executing and manages to also successfully change the contents of **B** to new_B. Now, T_1 is ready to perform its second READ operation: READ(B).*

*T_1 has now captured a value for both **A** and **B**, which we can consider as the "picture" of memory that it will use for future calculations. However, it has captured the contents of **A** and **B** in a way such that the "picture" they form has never occurred (i.e., both values existed at the same time). We can see that T_1 thinks that memory location **A** contains its old value (old_A) and that **B** contains new_B. But if we look at the memory content evolution throughout the execution of both threads, locations **A** and **B** have never concurrently contained these two values. At some point, **A** has contained old_A while **B** contained old_B; and at some other point in time **B** contained new_B, while **A** contained its new value (new_A).*

*In practice, this means that any computation T_1 performs with the captured values will be incorrect, simply because there is no point in time where the two values for **A** and **B**, old_A and new_B, have coexisted. Suppose **A** and **B**'s contents corresponded to two counters. T_1 has captured their count at different points in time, and therefore these values do not represent a valid state of the counters.*

Once again, this example illustrates the difficulties of correctly performing an

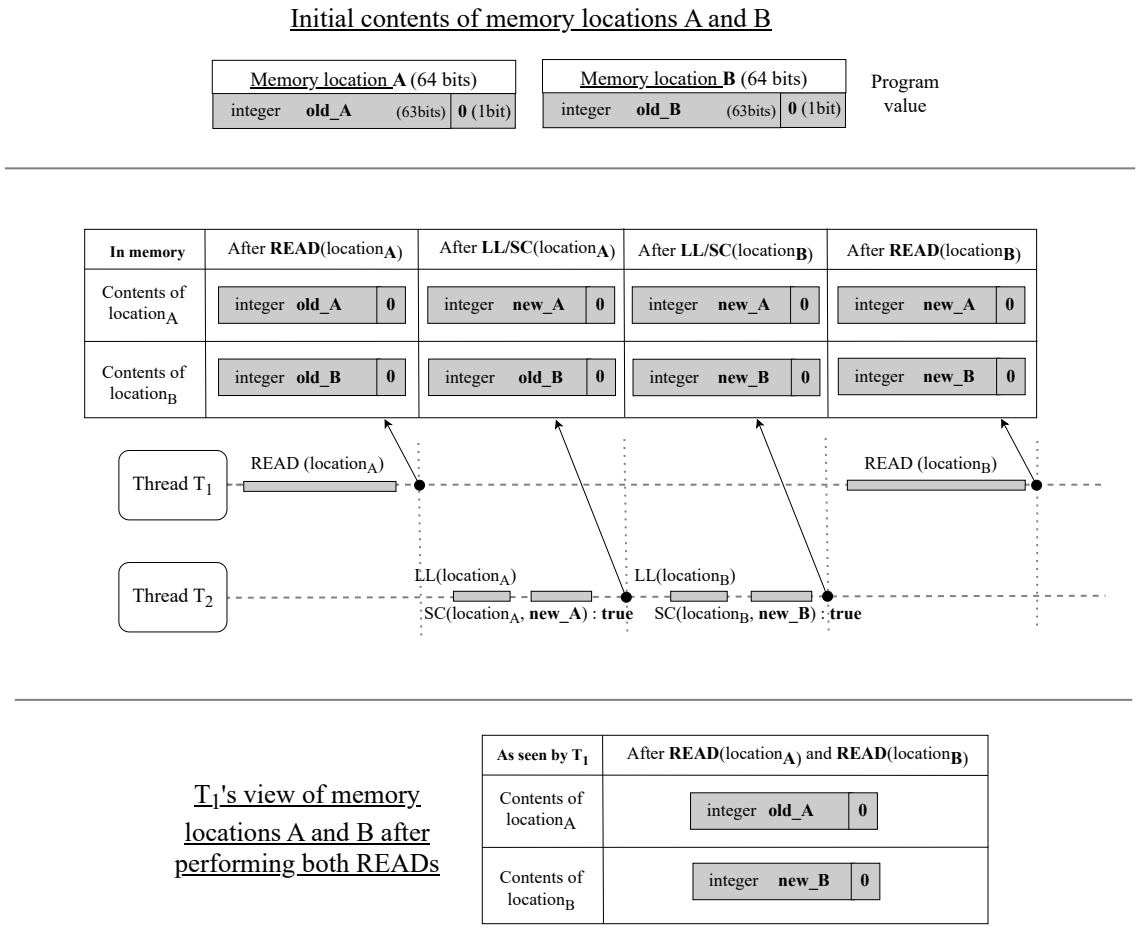


Figure 4.10: Time diagram illustrating the need for a SNAPSHOT operation, through an example where two threads T_1 and T_2 execute concurrent operations over two locations **A** and **B**. T_1 fails to read both locations in such a way that it retrieves a valid state of memory. It captures a memory state that corresponds to different moments in time per location, and this is invalid.

operation in a concurrent setting when this operation consists on several sub-steps, that have to be coordinated in order to provide an accurate picture of the state of memory. This motivates the introduction of a new operation that precisely answers to this requirement: providing a "picture" (which we will call SNAPSHOT) of K memory locations at once, with guarantees that it accurately represents the state of memory at a specific point in time between the invocation of the operation and its finalisation. The pseudo-code for this operation can be found in Figure 4.11.

```
1  uint64_t[1..k] COLLECT_VALUES(uint64_t k, loc_struct*[1..k] A) {
2    uint64_t[1..k] V;
3    for (uint64_t i = 1; i <= k; i++) {
4      V[i] = READ(A[i]);
5    }
6    return V;
7  }
8
9  uint64_t[1..k] COLLECT_SNAPSHOT_TS(uint64_t k, loc_struct* [1..k] A
       ) {
10   uint64_t[1..k] T;
11   for (uint64_t i = 1; i <= k; i++) {
12     T[i] = A[i]->snapshot_ts;
13   }
14   return T;
15 }
16
17 uint64_t[1..k] SNAPSHOT(uint64_t k, loc_struct* [1..k] A) {
18   uint64_t[1..k] Timestamps_1, Timestamps_2;
19   uint64_t[1..k] Values_1, Values_2;
20   while (true) {
21     Timestamps_1 = COLLECT_SNAPSHOT_TS(k, A);
22     Values_1 = COLLECT_VALUES(k, A);
23     Values_2 = COLLECT_VALUES(k, A);
24     Timestamps_2 = COLLECT_SNAPSHOT_TS(k, A);
25     if (for all i, (Timestamps_1[i] == Timestamps_2[i])
26                       && (Values_1[i] == Values_2[i]))
27     {
28       return Values_1;
29     }
30   }
31 }
```

Figure 4.11: Pseudo-code of the SNAPSHOT operation.

The SNAPSHOT operation is a well-known *non-blocking* technique capable of obtaining the content of a number of memory locations in a single point in time. In a few words, it works by repeatedly collecting the content of the locations in a sequential manner (through several READ operations called one after the other by the same thread), and then comparing the retrieved content to the ones read in its previous "collect". Once two "collects" have returned exactly the same values for every one of the locations, we can return these as the SNAPSHOT of memory.

The "collection" of values is done by an auxiliary operation COLLECT_VALUES

(see lines 1 to 7 in Figure 4.11), which we will explain later in depth. After two calls to COLLECT_VALUES return the same set of values, we are guaranteed that the collected locations have maintained the same content at a specific moment in time, which us the moment in time is the one the SNAPSHOT represents.

The only remaining question is how to determine whether a value has indeed not changed between both COLLECT_VALUES operations. We are already familiar with the **ABA problem** (See Example 4.4.3), so we can hint at the possibility of it occurring in the course of a SNAPSHOT operation: If a thread T_1 performs COLLECT_VALUES on a set of locations A[1..K] and retrieves values Values_1[1..K], but then comes another thread T_2 who changes the contents of A[1..K] to different values Values_2[1..K] and then back again to Values_1[1..K], a subsequent COLLECT_VALUES call by T_1 would think no one else has changed A[1..K]'s contents in the meantime, and this is not true.

To solve the **ABA problem** once again we apply the timestamp mechanism that LL/SC used. These timestamps, consisting of a tuple $\langle thread_{ID}, tag_{number} \rangle$ that will be stored together with the values that represent memory locations in loc_struct, as we can see on Figure 4.2, field snapshot_ts. Threads will update snapshot_ts upon every modification of a location structure loc_struct (See Line 9 at Figure 4.4), enabling others and itself to know who has last modified a location (thanks to snapshot_ts's field $thread_{ID}$), and to differentiate several own modifications (thanks to snapshot_ts's field tag_{number}). Timestamps will be collected together with locations' content, thanks to being stored as loc_struct structures, by operation COLLECT_SNAPSHOT_TS. SNAPSHOT will then require that both the result of COLLECT_VALUES and COLLECT_SNAPSHOT_TS match from a collection to the other one, in order to provide a correct "picture" of memory.

SNAPSHOT pseudo-code in detail (Figure 4.11): A call to SNAPSHOT(K, A[1..K]) by a thread T consists of a while(true) loop that repeatedly performs the following:

1. Calls COLLECT_SNAPSHOT_TS(K, A[1..K]) and stores the returned array of timestamps into Timestamps_1.
2. Calla COLLECT_VALUES(K, A[1..K]) to retrieve the values in locations A[1..K], and store the returned array into Values_1.
3. Then it repeats steps 1 and 2, but storing the results into array Timestamps_2 and Values_2.
4. Finally it compares values in arrays Values_1 and Values_2, and timestamps Timestamps_1 and Timestamps_2. If no unmatched value or timestamp is found, SNAPSHOT returns Values_1 as the result. Otherwise, it restarts on Step 1.

As we can see at Lines 1 to 15 of Figure 4.11, auxiliary operations COLLECT_VALUES and COLLECT_SNAPSHOT_TS simply loop over location structures and retrieve the content in which they are interested. Note the use of operation READ by COLLECT_VALUES,

which is necessary in order to guarantee that COLLECT_VALUES only returns *program values*.

4.6. KCSS Operation

The final step in order to build KCSS, is to put all pieces that we have seen so far together. As we discussed in Section 4.2, a KCSS call:

```
KCSS (Locations[A_1..A_k], ExpectedVals[e_1..e_k], newValue_1)
```

checks that every location A_i contains its expected value e_i. If this is the case, then it updates A_1 with the new value newValue_1. Otherwise, it does not modify any memory location.

We now know how to find out a location's *program value* even on locations undergoing changes by other threads; we have seen how to successfully modify the content of a location while ensuring that no other thread has accessed it; and we know how to correctly retrieve the value of K locations at a specific point in time. These are all the pieces we have to put together.

```
1  bool KCSS(uint64_t k, loc_struct* [1..k] A, uint64_t[1..k]
       expectedValues, uint64_t newValue) {
2    uint64_t[1..k] oldValues;
3    while (true) {
4      oldValues[1] = LL(A[1]);
5      oldValues[2..k] = SNAPSHOT(k - 1, A[2..k]);
6      if (for some i, oldValues[i] != expectedValues[i]) {
7        SC(A[1], oldValues[1]); // revert A[1] to oldValues[1]
8        return false; // KCSS was unsuccessful
9      }
10     //else try to finalise:
11     if (SC(A[1], newValue)) {
12       return true; // KCSS was successful
13     }
14   }
15 }
```

Figure 4.12: Pseudo-code of the KCSS operation.

A thread T executing a call to KCSS simply does the following (the pseudo-code for this operation can be found on Figure 4.12):

1. At Line 4, T initiates an LL/SC on location A_1 by performing LL(A_1), declaring its intentions of modifying this location.

2. At Line 5, before proceeding with the modification of A_1, T has to make sure that the K locations it is considering will **maintain their content from now until T's action of "committing" its modifications to A_1**.

To do this, it will perform a SNAPSHOT of those locations, obtaining a "picture" of memory that it will be able to contrast with the expected values of Locations[A_1..A_k]. Note that SNAPSHOT is called with K - 1 locations (Locations[A_2..A_k]) because the LL/SC currently ongoing on A_1 already takes care of maintaining the value of this location.

3. Now that T has retrieved the content of Locations[A_1..A_k] (the contents of Locations[A_2..A_k] are captured by SNAPSHOT and those of A_1 are returned by LL(A_1) on step 1), it can proceed to check that every location A_i currently stores the expected value e_i (Line 6). Any mismatch on the way will result in aborting KCSS, reverting any changes made earlier by returning A_1 to its initial value if possible[5] (see Line 7). On the contrary, a perfect match of all locations to their expected values paves the way for a successful KCSS. T will be able to move on to try "committing" the newValue onto A_1 (see lines 10 to 13).

This process is attempted repeatedly, until either the KCSS fails, or it succeeds. This concludes the detailed explanation of the algorithmic intricacies of KCSS. Now that we have built all the necessary foundations and understanding of this powerful operation, next we consider its possible limitations.

4.7. Limitations of the KCSS Operation

KCSS provides an answer to the need for a software primitive solving the challenges that *Concurrent Linked Data Structures* pose. However, there are some considerations to take into account when using it.

Firstly, as we mentioned in Section 4.2, throughout all our work we have assumed that the system on which KCSS runs provides *linearizable* Load, Store and CAS operations. This requirement cannot be relaxed due to the strict conditions under which memory models are consistent. In our case, without *linearizable* Load, Store[6] and CAS operations, we would not have any guarantees when reasoning with the ordering of operations that access memory.

Secondly, the only functional drawback of KCSS, which we are willing to pay in exchange for the rest of its assets, is that it is *non-blocking*. This progress condition is a relaxation of lock-freedom. *Non-blocking* constructs depend on the contention of

[5] Actually, in this case, the SC(A_1, oldValues[A_1]) is there to ease the work of other threads. If T left it containing the *temporary value* it had placed after performing LL(A_1), another thread accessing A_1 would have to first go through the process of resetting A_1 no matter the operation it wanted to perform over A_1. On the other hand, if T already leaves A_1 in the state it found it when it started executing KCSS, T is easing the access to A_1 for other threads and itself in the future.

[6] Note that here Load and Store refer to memory load and memory store operations, (they have nothing to do with the higher-level LL or SC operations, which in fact make use of Load and Store when they access memory locations). The *linearizability* of memory Load and Store operations is given by the memory model upon which the system is implemented.

the system, so as long as several threads are running, they do not guarantee progress to be made. The only guarantee they provide is that threads will keep retrying so that, as soon as they get to run alone, they will make progress.

It is important to keep this in mind because as users of a *non-blocking* construct such as `KCSS` we could encounter counter-intuitive behaviours whose correctness we need to be able to reason about. More specifically, in the case of `KCSS`, threads retrying an operation and finally managing to run on their own can find that the environment on which they were designed to run has changed (for example, if we are using `KCSS` on a list where a thread T_1 was going to delete an element, but this element has already been deleted by another thread T_2 before T_1 got the chance to run on its own). This would **correctly** trigger the failure of that/those thread(s), but it is not trivial to reason why this happened, due to the concurrent nature of the execution.

Lastly, the `KCSS` algorithm requires one bit in memory locations to distinguish *program values* from *temporal values*, and therefore it cannot be used in contexts in which *program values* cannot spare this bit. Nevertheless, for the interested reader, in (Doherty et al., 2004), a novel implementation of the `LL` and `SC` operations that overcomes this weakness is provided.

4.8. Related Work

The `KCSS` operation is one among many initiatives to develop multi-location synchronization software primitives. The most relevant of which are variations of the idea of generalizing `CAS` (**Single-Location** Compare and Swap operations) to support more than one location.

`DCAS` (**Double-Location** Compare and Swap) was the first of those initiatives to see the light, being introduced into the Motorola MC68030 chip as a hardware-supported primitive. `DCAS` initiatives soon derived into attempts to achieve a generalist K-Location Compare and Swap primitive. In (Barnes, 1993) Barnes introduced for the first time a software implementation for such an operation. His implementation was based on the *cooperative helping* synchronisation technique: threads "help" each other complete an on-going operation before attempting their own. This mechanism also has shortfalls, like the extra overhead incurred when carrying unnecessary "helping" actions, (repeating actions already carried out by the thread being helped due to dependencies existing between actions, and threads not being aware of them).

Along the K-Location Read-Modify-Write line of research, Herlihy and Moss first suggested (Herlihy et al.) *transactional memories*: an approach to synchronise actions to shared resources through a mechanism mimicking that of a transactional system where transactions represent actions proposed and waiting to be committed until their correctness is verified. This approach has seen great development in recent years, with Shavit and Toitou coming up with *Software Transactional Memories* (STM) in (Shavit and Touitou, 1997), a *lock-free* implementation that reduced redundant "helping" between threads and therefore outperformed most known *lock-*

free algorithms at the time. Their proposal, however, required knowing in advance the memory locations subject to accesses by the STM. Herlihy, et al. would later introduce a *Dynamic* STM (Herlihy et al., 2003) capable of handling memory locations not known in advance, at the cost of being *obstruction-free* instead of *lock-free*. The limitation of STMs is their lack of scalability due to the overhead of transaction formatting and processing of actions as such. The `KCSS` operation itself could be implemented using a *Dynamic* STM, but the result would be a complex software application requiring fresh memory to be allocated per executed transaction or a specialized built-in memory management scheme to be introduced. It would only increase time complexity slightly, but have twice the memory overhead of our `KCSS` implementation, which is lightweight and can be used transparently.

Finally, in (Harris et al., 2002) Harris, Fraser and Pratt propose a *non-blocking* implementation of a K-Location `CAS` operation (which they call `CASN`) that also requires a constant amount of memory per word involved, but is capable of modifying the contents of all K locations, as opposed to `KCSS`. It does so by imposing more severe memory model requirements for consistency.

Chapter 5

K-CSS C++ Implementation

In this chapter we will present an original C++ implementation of the KCSS operation we discussed on Chapter 4. The reason for choosing C++ is its efficiency and support for concurrency (since version C++11, it includes support for threads, atomic operations, mutual exclusion, and more).

The KCSS code has been designed with the goal of making its usage transparent and straightforward, and without imposing any restriction on the use of primitive type (in C++) that the user already familiar with. In practice, this means several things. Firstly, we have made KCSS's internal representation of data transparent to the user, who does not have to manage complex data structures defined by KCSS. Secondly, we have designed KCSS so that the logic behind its mechanism's is hidden from the user. As we have often mentioned in this work, the main goal for creating KCSS is providing a self-contained version of CAS that works for K locations. We call it "self-contained" because KCSS is expected to "encase" all the actions necessary for its application within its own code, so that it can be used via a single call like other sequential operations. This contrasts with other software primitives often used in concurrency whose logic cannot be decoupled from that of the running program, making coding error-prone and less maintainable in the long run.

5.1. Data Structures and Defined Types

KCSS is formally represented as a class (see Figure 5.1 for an overview of the class structure), whose main attributes are: the SAVED_VAL array (see Section 4.3 to review the use for this array), array TAG_NUMBERS (will provide tag numbers to threads, see Example 4.4.3 to review the need for them), and a _thread_id variable that will be used to generate unique thread IDs (how to do this will be explained on Section 5.2).

The final key component of the KCSS class is the CAS operation (lines 13 to 15 on Figure 5.1), which makes use of the following GCC function:

```
bool __sync_bool_compare_and_swap (T *ptr, T oldVal, T newVal)
```

whose implementation is based on the use of the assembly instruction `CMPXCHG`. It performs an *atomic Compare And Swap* operation: If the current value of `*ptr` is `oldVal`, then it writes `newVal` into `*ptr` and returns `true`, otherwise no action is performed and it returns `false`.

```
1  class KCSS {
2  private:
3
4    enum {
5      MAX_THREAD_ID = 100
6    };
7
8    uint64_t SAVED_VAL[MAX_THREAD_ID];
9    uint16_t TAG_NUMBERS[MAX_THREAD_ID];
10
11   std::atomic<uint16_t> _thread_id;
12
13   inline bool cas(uint64_t* loc_pointer, uint64_t exp, uint64_t
      new_val) noexcept {
14     return __sync_bool_compare_and_swap(loc_pointer, exp, new_val);
15   }
16
17 public:
18
19   KCSS() noexcept :
20     SAVED_VAL(), TAG_NUMBERS(), _thread_id() {
21   }
22
23   // This part will be explained shortly after
24   //  [...]
25
26 };
```

Figure 5.1: The `KCSS` class code definition.

Recall that `KCSS` acts upon memory locations using a special data structure we called `loc_struct` (see Section 4.3). In `C++`, this structure is a `struct` type we have called `loc_struct_base` as depicted in Figure 5.2, which contains the fields we discussed in Section 4.4: `snapshot_ts` and `value`. Note that type `uint64_t` represents a 64-bit unsigned integer in `C++`, and that we use it to represent words of 64bit in memory.

The `loc_struct_base` structure is also declared as private in the `KCSS` class, since the user will not need to manipulate it directly, but rather throw other public methods and template data structures. In fact, the `KCSS` (from the use prespective) operates on derived `struct loc_struct_t<T>` that are parameterized by using `C++` template parameter `T`, and use template specialization in order to provide different definitions for the different values of `T`, because different types have to comply with different format restrictions (i.e., 64bits integer and floating-point types has to lose 1bit precision, while thos of 32bit integer do not).

On Figure 5.3 we can find the `loc_struct_t<T>` declaration for the `C++` types

```
1  struct loc_struct_base {
2    loc_struct_base() noexcept :
3        snapshot_ts(0), value(0) {
4    }
5    loc_struct_base(uint64_t v) noexcept :
6        snapshot_ts(0), value(v) {
7    }
8    uint64_t snapshot_ts;
9    uint64_t value;
10 };
```

Figure 5.2: `loc_struct_base` C++ code

`uint64_t` and `int64_t`. This definition is used by the compiler if `T` sastified the following constraint:

```
std::enable_if_t< std::is_same_v<T, uint64_t> ||
                  std::is_same_v<T, int64_t»
```

This particular `loc_struct_t<T>` declaration defines an internal `struct t63` that will be used to properly adapt user-provided values of type T (which are `uint64_t` and `int64_t` in this case) to the `loc_struct_t<T>` representation. The way this is done is by using the C++ **bit field** capability to compact a 64bit user-value `T` into the available 63bits in KCSS's value representation format. The **bit field** capability allows to selectively assign the content of every bit or group of bits in a C++ type through the usual assignment operation "=".

Remember that in the case of `loc_struct_t<T>` representations where `T` is either `uint64_t` or `int64_t`, we have to reduce the initial 64 available bits for representing integer values to 63 bits. As we know, this reduction from 64 to 63 bits is necessary due to two factors:

1. The existence of a **1bit** `location_type` field in memory locations that allows us to differentiate locations storing *temporal values* from those holding *program values*, (as seen on Section 4.3); and

2. The need for all `loc_struct_t` structures to occupy at most 64bits so that they are valid arguments for the CAS operation (which take 64bit words as arguments).

Therefore, `t63` is a structure that is used to adapt 64bit user-provided values `T` to the fine-grained **bit-wise representation** imposed by KCSS, consisting of fields `t63->location_type` and `t63->content` (will store a 63bit version of the initial user-value). The `t63` structure is merely an auxiliary tool to adapt values from user-format to KCSS-format, easing access to each field separately, but it is only used at punctual moments when a transition from one of these formats to the other one has to be done i.e. when a user provides a new value to be used within KCSS code, or when the KCSS algorithm returns a result that has to be user-readable. All

along, the KCSS code manages uint64_t values which it treats as memory words; and auxiliary methods are employed at times to preserve the format characteristics that this representation carries, transparently to methods like LL, SC, SNAPSHOT, etc.

```
1  // loc_struct_t for uint64_t and int64_t
2  template<typename T>
3  struct loc_struct_t<T, std::enable_if_t< std::is_same_v<T,
    uint64_t> || std::is_same_v<T, int64_t>>> : loc_struct_base {
4
5    struct t63 {
6      T location_type :1;
7      T content :63;
8    };
9
10   loc_struct_t() noexcept :
11     loc_struct_base() {
12   }
13
14   loc_struct_t(T v) noexcept :
15     loc_struct_base(to_value_t(v)) {
16   }
17
18   constexpr inline uint64_t to_value_t(T v) const noexcept {
19     union {
20       t63 x;
21       uint64_t _x;
22     };
23     x.location_type = 0;
24     x.content = v;
25
26     return _x;
27   }
28
29   constexpr inline T from_value_t(uint64_t v) const noexcept {
30     union {
31       t63 x;
32       uint64_t _x;
33     };
34     _x = v;
35
36     return x.content;
37   }
38 };
```

Figure 5.3: loc_struct_t<T> C++ code, when T is either uint64_t or int64_t.

Let us now look at two key methods in charge of format-compliance, provided by all loc_struct_t<T> declarations (and in particular by loc_struct_t<uint64_t> and loc_struct_t<int64_t> structures): to_value_t and from_value_t (lines 18 to 37 on Figure 5.3 contain the corresponding C++ code).

Method to_value_t(T v) is used to convert a user-provided value v into its adequate representation as a uint64_t value (a memory word), just like the pseudo-code method make_ProgramValue did (see Figure 4.5 for the usage of this method). It is

an auxiliary method to "encode" a user-provided value into its KCSS representation, all the while occupying a 64bit-word). This is done employing the C++ **union** capability, which allows more than one variable, possibly with different type, to be stored the same memory. This means that the value in that memory can be interpreted as one type or another, depending which variable we use to refer to it. In this case, `to_value_t` exploits this capability to re-interpret the user-defined value `v` (which is ultimately a 64bit word `uint64_t`) as a `t63` structure. This is done so that we can take advantage of the ease of bit manipulation through **bit field** assignment to do the following: A `t63` structure `x` is declared in the scope of a **union** construct, altogether with a `uint64_t` value `_x` (Lines 19 to 22 on Figure 5.3), which means that the assignments on Lines 23 and 24 in Figure 5.3 do not only modify `x`'s bits, but also `_x`'s. In particular, the last bit (`t63->location_type`) of both `x` and `_x` will contain a `0` to indicate that the value contained is a *program value*; and the remaining 63bits (`t63->content`) will be filled using the 64bit user-provided value `v`, which will be automatically truncated to fit into 63bits (this is the real power of **bit field** assignment).

Method `from_value_t(uint64_t v)` does the oposite of `to_value_t`. It extracts the content of a KCSS *program value* `v` in order to return its user-format-complying value. Once again, this is done employing the C++ **union** capability to re-interpret the `uint64_t` value `v` as a `t63` structure so that we can easily access the `t63` structure's **bit fields**. More specifically, we are interested in the `content` bits of `v`, which are the 63 least significant ones. Therefore this method returns `t63->content`, which C++'s **union** will automatically fit into a `uint64_t` value `_x` that will be then re-interpreted as a `T` type upon return.

Note that the processing we have described for values of type `uint64_t` or `int64_t` varies for other types. Since KCSS has an exhaustive set of templates covering all possible primitive data types, the compiler will choose the appropriate one depending on the data type provided by the user. Pointers are by default aligned on memory addresses that are even that they can be represented without the need to remove 1 precision bit as it happens for integers (because the least significant bit is always zero). This is also the case for types occupying less than 64bits (such as `int8_t`, `int16_t`, `int32_t` and single-precision floating-point numbers): they can be represented without modifying the original value because they fit into 63bits, which is the maximum allocation of bits that values can use. Their corresponding `loc_struct_t` structure just needs to implement the `to_value_t` and `from_value_t` methods so that all `loc_struct_t` structures share a common interface, and all necessary type conversions can be performed.

The C++ code corresponding to the `loc_struct_t<T>` for the aforementioned cases can be found at Figure 5.4 (code for numeric types occupying less than 64bits) and Figure 5.5 (code for the pointer case).

On the other hand, double-precision floating-point numbers (`double`) pose a new challenge because their bit-wise representation consists of several parts[1]: sign (1 bit), exponent (11 bits), and mantissa or fraction (52 bits); as we can see on

[1] `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`

```
// loc_struct_t for any numeric type of size smaller than 8bytes
template<typename T>
struct loc_struct_t<T,
    std::enable_if_t<
        (std::is_integral_v<T> || std::is_floating_point_v<T>)
            && (sizeof(T) < 8)>> : loc_struct_base {

  loc_struct_t() noexcept :
      loc_struct_base() {
  }

  loc_struct_t(T v) noexcept :
      loc_struct_base(to_value_t(v)) {
  }

  constexpr inline uint64_t to_value_t(T v) const noexcept {
    union {
      struct {
        uint8_t location_type;
        T a2;
      } t;
      uint64_t a1;
    };
    a1 = 0;
    t.a2 = v;

    return a1;
  }

  constexpr inline T from_value_t(uint64_t v) const noexcept {
    union {
      struct {
        uint8_t location_type;
        T a2;
      } t;
      uint64_t a1;
    };
    a1 = v;

    return t.a2;
  }
};
```

Figure 5.4: `loc_struct_t<T>` C++ code, when `T` is any any numeric type of size smaller than 8bytes (64bits).

```
1  // loc_struct_t for pointers
2  template<typename T>
3  struct loc_struct_t<T*> : loc_struct_base {
4
5    loc_struct_t() noexcept :
6        loc_struct_base() {
7    }
8
9    explicit loc_struct_t(T *v) noexcept :
10       loc_struct_base(to_value_t(v)) {
11   }
12
13   constexpr inline uint64_t to_value_t(T *v) const noexcept {
14     static_assert( (alignof(T) & 1) == 0 );
15     union {
16       T *v_aux;
17       uint64_t a;
18     };
19     v_aux = v;
20     assert((a & 1) == 0);//pointer must be aligned to even
    address
21
22     return a;
23   }
24   constexpr inline T* from_value_t(uint64_t v) const noexcept {
25     union {
26       T *a;
27       uint64_t v_aux;
28     };
29     v_aux = v;
30
31     return a;
32   }
33 };
```

Figure 5.5: `loc_struct_t<T>` C++ code, when `T` is a pointer.

Figure 5.6. In this case, "stealing 1 precision bit" is still the mechanism we will use to represent them as `loc_struct_t` structures, setting to `0` the least significant bit of the mantissa. In practice, just like in the case of integer numbers, this means we reduce the range of possible values, but in this case the loss of precision is even more negligible: The maximum relative rounding error will be exactly of 2^{-53}. This is acceptable because 64bit values cover a vast amount of possible numbers, almost inextinguishable for a common-use program. Nevertheless, the user will have to be warned about this limitation, for the sake of correctness.

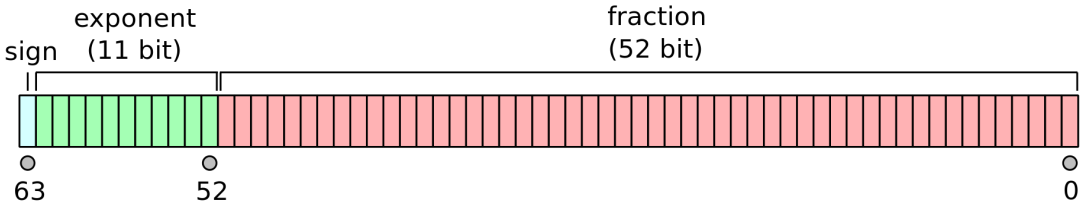


Figure 5.6: Components of a double-precision floating-point number. Figure taken from `https://en.wikipedia.org/wiki/Double-precision_floating-point_format`.

Figure 5.7 contains the `C++` code corresponding to the `loc_struct_t<T>` structure, when `T` is a double-precision floating-point number (`double`). We can see on Line 18 that the way method `to_value_t` works in this case is slightly different to the `uint64_t` and `int64_t` case. Instead of using an auxiliary structure (like `struct t63`) and **bit field** assignment, `double` values are better manipulated using bit-wise operations. So `to_value_t` uses the **union** construct to apply bit-wise operation `&` (logical `AND`) to user-provided value `double v` (that has been assigned to value `double a1` and therefore to `uint64_t a2` on Line 17 of Figure 5.7), "turning" the least significant bit of the mantissa to `0`, which will represent the `location_type` field in `KCSS` value format (being `0` indicates it is a *program value*).

All in all, we have provided proof that there are mechanisms in place so that users do not have to meddle with the internal representation of memory in `KCSS`. They will simply call `KCSS`'s methods using wrappers that `KCSS` provides so that they can declare which values are going to be susceptible of experiencing a `KCSS` operation, but without knowing what actions are being carried out at the `KCSS` class to enable this. When users declare a value of type `KCSS::loc_struct_t<T>` (`T` being some type definition of their choice), the `C++` compiler will automatically assign it to its corresponding template, whose definition is such that the rest of the methods in `KCSS` (LL, SC, SNAPSHOT, ...) can use it like any other `KCSS::loc_struct_t` piece of data.

```
template<>
struct loc_struct_t<double> : loc_struct_base {

  loc_struct_t() noexcept :
      loc_struct_base() {
  }

  loc_struct_t(double v) noexcept :
      loc_struct_base(to_value_t(v)) {
  }

  constexpr inline uint64_t to_value_t(double v) const noexcept {
    union {
      double a1;
      uint64_t a2;
    };
    a1 = v;
    a2 = a2 & 0xfffffffffffffffe; // turn the least significant
 bit of the mantissa to 0

    return a2;
  }

  constexpr inline double from_value_t(uint64_t v) const noexcept
  {
    union {
      uint64_t a2;
      double a1;
    };
    a2 = v;

    return a1;
  }
};
```

Figure 5.7: `loc_struct_t<T>` C++ code, when `T` is a double-precision floating-point number.

5.2. Threads and their Identifiers

Another method worth mentioning, related to compliance with KCSS's representation format of values, is method (depicted in Figure 5.8):

```
make_TemporalValue(uint16_t thread_ID, uint64_t tag_number)
```

As the name suggests, this method takes as parameters the components necessary to build a *temporal value* (`thread_ID` and `tag_number`) and returns the resulting 64bit-compliant representation (for a reminder of *temporal values*, refer to Section 4.3). *Temporal values* are represented via struct `TemporalValue`, which uses C++'s **bit fields** to manage the bit-wise manipulation of its three components: `location_type` (1 bit), `thread_ID` (15 bits) and `tag_number` (48 bits), which make up a single

64bit memory word (`uint64_t` value). `make_TemporalValue` simply fills these fields given the parameters it receives, setting `location_type` to be 1 (which is *temporal values*' identifying trait). This last bit is precisely the one checked by auxiliary method `is_TemporalValue(uint64_t v)`, which returns `true` if the given value `v` is a *temporal value*, and `false` otherwise.

```
1  struct TemporalValue {
2    uint64_t location_type  :1;
3    uint64_t thread_ID    :15;
4    uint64_t tag_number   :48;
5  };
6  static_assert( sizeof(TemporalValue) == 8 );
7
8  constexpr inline bool is_TemporalValue(uint64_t v) const noexcept
     {
9    return (v & 0x0000000000000001) == 1;
10 }
11
12 inline constexpr uint64_t make_TemporalValue(uint16_t thread_ID,
13   uint64_t tag_number) const noexcept {
14   union {
15     TemporalValue vt;
16     uint64_t a;
17   };
18
19   vt.thread_ID = thread_ID;
20   vt.tag_number = tag_number;
21   vt.location_type = 1;
22
23   return a;
24 }
```

Figure 5.8: `make_TemporalValue` C++ code, together with some auxiliary methods.

The way to obtain the currently running thread's identifier, is through method `my_thread_id()`. This method (see Figure 5.9) makes use of a `thread_local` variable `my_thread_id` that is also `static`, which stores the increment by one unit of private variable `_thread_id`. The combination of being `thread_local` and `static` means that this variable has a local value for every thread (it is not shared) and that it is only assigned once (the first time the method is called). Therefore, every time method `my_thread_id()` is called, it will return the result of having incremented private variable `_thread_id` the first time that it was called.

Method `thread_id(uint64_t v)` (see lines 6 to 13 on Figure 5.9) is an auxiliary method that extracts field `thread_ID` from a *temporal value* variable `v`. It is employed by the `RESET` operation in order to index the `SAVED_VAL` array (see Section 4.4 for a reminder of the `RESET` operation).

```
uint16_t my_thread_id() noexcept {
  thread_local static uint16_t my_thread_id = _thread_id++;
  return my_thread_id;
}

constexpr inline uint16_t thread_id(uint64_t v) const noexcept {
  union {
    TemporalValue tv;
    uint64_t a;
  };
  a = v;

  return tv.thread_ID;
}
```

Figure 5.9: `my_thread_id` and `thread_id` C++ code.

5.3. Implementation of methods LL and SC

The C++ code for LL and SC operations exactly follows the pseudo-code we explained in Section 4.4. If we retrace our steps back to the pseudo-code for LL (see Figure 4.4), we can compare both figures statement by statement and see they are identical. In the case of SC operations, however, there is a minor difference: our pseudo-code implementation (see Figure 4.5) includes a call to `make_ProgramValue(a, new_program_val)`, which is not present on the C++ version of the code. This is because in this case it will actually be called with a program value (i.e., a value whose first bit is 0, and already adapted to fit in 63 bits – see the discussion in the previous section). This is the result of our careful and transparent management of data formats using `loc_struct_t<T>` structures and their `to_value_t` and `from_value_t` methods. All of KCSS's code manages `uint64_t` values that already comply to the KCSS format for *program* and *temporal* values, so the call to `make_ProgramValue` (which in C++ has an equivalent method named `to_value_t`) will have already been done before calling SC. We can see an example of this data format compliance before calling LL and SC on the KCSS operation code, that will be explained in Section 5.5.

As for auxiliary operations RESET and READ (their C++ implementation can be found on Figure 5.12), once again their pseudo-code (see Figure 4.9) and their C++ implementation exactly match if we compare every code statement one by one.

5.4. Implementation of SNAPSHOT method

The C++ code for the SNAPSHOT operation can be found on Figure 5.13, altogether with the C++ code for its auxiliary operations COLLECT_SNAPSHOT_TS (Figure 5.14) and COLLECT_VALUES (Figure 5.15). The reader is encouraged to look back into Section 4.5 and verify that the pseudo-code provided for these operations (see Figure 4.11) exactly matches the C++ code in Figures 5.13, 5.14 and 5.15, the only difference being that our C++ implementation of auxiliary operations COLLECT_VALUES

```
inline uint64_t ll(loc_struct_base *a) noexcept {
  while (true) {
    TAG_NUMBERS[my_thread_id()]++;
    uint64_t old_val = read(a);
    SAVED_VAL[my_thread_id()] = old_val;
    uint64_t temp_val = make_TemporalValue(my_thread_id(),
        TAG_NUMBERS[my_thread_id()]);
    if (cas(&a->value, old_val, temp_val)) {
      a->snapshot_ts = temp_val;
      return old_val;
    }
  }
}
```

Figure 5.10: `LL` operation `C++` code.

```
inline bool sc(loc_struct_base *a, uint64_t new_prog_val)
 noexcept {
  uint64_t temp_val = make_TemporalValue(my_thread_id(),
 TAG_NUMBERS[my_thread_id()]);
  return cas(&a->value, temp_val, new_prog_val);
}
```

Figure 5.11: `SC` operation `C++` code.

```
inline void reset(loc_struct_base *a) noexcept {
  uint64_t old_val = a->value;
  if (is_TemporalValue(old_val)) {
    cas(&a->value, old_val, SAVED_VAL[thread_id(old_val)]);
  }
}

inline uint64_t read(loc_struct_base *a) noexcept {
  while (true) {
    uint64_t val = a->value;
    if (!is_TemporalValue(val))
      return val;
    reset(a);
  }
}
```

Figure 5.12: `reset` and `read` operations `C++` code.

and `COLLECT_SNAPSHOT_TS` do not return the array of values or tags, but instead modify an array parameter they receive by value reference.

```
inline void snapshot(std::size_t k, loc_struct_base **a, uint64_t
  *values_1) noexcept {
  uint64_t timestamps_1[k], timestamps_2[k];
  uint64_t values_2[k];

  while (true) {
    collect_snapshot_ts(k, a, timestamps_1);
    collect_values(k, a, values_1);
    collect_values(k, a, values_2);
    collect_snapshot_ts(k, a, timestamps_2);

    unsigned i = 0;
    while (i < k && timestamps_1[i] == timestamps_2[i] &&
 values_1[i] == values_2[i])
      ++i;

    if (i == k)
      return;
  }

  return;
}
```

Figure 5.13: SNAPSHOT operation C++ code.

```
inline constexpr void collect_snapshot_ts(const std::size_t k,
    loc_struct_base **a, uint64_t *t) const noexcept {
  for (unsigned int i = 0; i < k; ++i) {
    t[i] = a[i]->snapshot_ts;
  }
}
```

Figure 5.14: COLLECT_SNAPSHOT_TS operation C++ code.

```
inline void collect_values(const std::size_t k, loc_struct_base
 **a,
    uint64_t *v) noexcept {
  for (unsigned int i = 0; i < k; ++i) {
    v[i] = read(a[i]);
  }
}
```

Figure 5.15: COLLECT_VALUES operation C++ code.

5.5. Implementation of KCSS method

The C++ code for the KCSS operation can be found on Figure 5.16. Note that KCSS is the only operation among the ones we have described (LL, SC, SNAPSHOT, ...) that is public for the user to access.

The KCSS C++ implementation slightly varies from the pseudo-code provided in

Figure 4.12 (see Section 4.6 for the detailed explanation of this operation). It varies in that arguments are provided using C++ templates with a **function parameter pack** (C++ capability which allows defining a function parameter that accepts zero or more function arguments), and therefore these individual parameters have to be converted into arrays (see lines 7 to 9 on Figure 5.16). Also note that each location and its expected value, except the first, is provided as a pair using a single argument. Another important remark are the calls to method `to_value_t` in lines 8 and 10 of Figure 5.16, which, as we mentioned on Section 5.3, are necessary to transform the user parameter values of type `T` into their corresponding KCSS representation as a formatted word.

```
1  template<typename T0, typename ...Ts>
2  bool kcss(loc_struct_t<T0>& a0, T0 a0_expv, T0 a0_newv, Ts &&...
     args) noexcept {
3
4    constexpr std::size_t k = sizeof...(args) + 1;
5
6    uint64_t old_vals[k];
7    uint64_t exp_vals[k] = { a0.to_value_t(a0_expv),
8        (args.first.to_value_t(args.second))... };
9    loc_struct_base* a[k] = { &a0, (&args.first)... };
10   uint64_t new_v = a0.to_value_t(a0_newv);
11
12   while (true) {
13     old_vals[0] = ll(a[0]);
14     snapshot(k - 1, a + 1, old_vals + 1);
15     for (unsigned int i = 0; i < k; ++i) {
16       if (old_vals[i] != exp_vals[i]) {
17         sc(a[0], old_vals[0]);
18         return false;
19       }
20     }
21
22     if (sc(a[0], new_v)) {
23       return true;
24     }
25   }
26
27   return false;
28 }
```

Figure 5.16: KCSS operation C++ code.

5.6. KCSS Usage Example

On this section, we will present a C++ code example of the usage of the KCSS operation implementation. This example, together with the rest of the code for KCSS,

can be found at https://github.com/Liidiacnog/KCSS_Project. In particular, we have included in this repository several test cases using KCSS. Two of them act on linked lists under the influence of actions carried out by either one thread (see Figure 5.20 for the C++ code) or two threads (see Figure 5.21 for the C++ code); and the last test displays the behaviour of KCSS on a highly contended case: having n threads concurrently attempting to modify the same value (see Figure 5.22 for the C++ code). We will provide a detailed explanation of the case involving linked lists with two concurrently executing threads, and we invite the reader to review the remaining examples on their own.

Let us first study the environment setup to carry out the aforementioned tests. The corresponding C++ code for this setup can be found at Figure 5.18.

We will first define the structure of nodes making up the linked list: struct node. The peculiarity of these nodes is that, since we want their fields to be susceptible targets of the KCSS operation, they have to be declared as such. So if our linked list is made up of integer numbers, the main content of nodes (field node->x) has to be declared of type KCSS::loc_struct_t<int>. The same applies to the linking pointer to subsequent nodes on the list (field node->next), which will be of type KCSS::loc_struct_t<node*>.

As for the print_list() method (Line 16 on Figure 5.18), it is simply a method that takes as parameter the first node in a linked list, and traverses it in order to print it on standard output. What is remarkable not only on this method but in general in any code that makes use of KCSS::loc_struct_t<T> types, is that users cannot simply access the content of variables declared as such by accessing their memory location. Because of KCSS's format restrictions on memory locations, the content of a KCSS::loc_struct_t<T> structure does not comply with the data format a user would expect (in our case, if field node->x is of type KCSS::loc_struct_t<int>, the user would expect a 64bit integer to be returned, but we already saw that the internal representation of 64bit integers for KCSS is made up of 63bit integers plus a special bit). This means we need a method to decode a KCSS::loc_struct_t<T> variable's contents into their user-format, by means of an auxiliary method: KCSS::get(). KCSS::get(), whose C++ code can be found at Figure 5.17, is a static method that does not only perform the necessary call to the "decoder" method from KCSS format to user-format, but it also accesses a location's contents using method READ. Once again, this call is necessary to ensure that the value read is not being concurrently modified by other threads, as explained on Section 4.4. Therefore, method print_list() makes use of the KCSS::get() operation provided by KCSS in order to access field node->next when traversing the given linked list.

The same applies to method cleanup(), which traverses a given linked list to delete its contents from heap memory using the KCSS::get() method.

Let us now look at the test case consisting of two concurrent modifications to a linked list by two threads (method two_threads() in Figure 5.21). Initially, we build a linked list consisting of three nodes in the following order: a, c and d, which store integers 1, 2 and 4, respectively. The goal of this test is to spawn two threads, and observe the behaviour of the linked list when undergoing concurrent

```
1   template<typename T>
2   inline T get(loc_struct_t<T> &a) noexcept {
3     return a.from_value_t(read(&a));
4   }
```

Figure 5.17: `KCSS::get()` method C++ code.

```
1  // Instantiate KCSS class
2  KCSS kcss_instance;
3
4  // Linked List definition: linked nodes using the following node
       structure
5  struct node {
6    node():
7      x(0), next(nullptr) {
8    }
9    node(int _x_val, node* _next_val):
10     x(_x_val), next(_next_val) {
11   }
12   KCSS::loc_struct_t<int> x;
13   KCSS::loc_struct_t<node*> next;
14 };
15
16 void print_list(node* first_node){
17   node* next_node = first_node;
18   while(next_node != nullptr){
19     std::cout << kcss_instance.get(next_node->x) << "->";
20     next_node = kcss_instance.get(next_node->next);
21   }
22   std::cout << "nullptr\n\n";
23 }
24
25 void cleanup(node* first_node){
26   node* next_node = first_node;
27   node* following;
28   while(next_node != nullptr){
29     following = kcss_instance.get(next_node->next);
30     delete(next_node);
31     next_node = following;
32   }
33 }
```

Figure 5.18: C++ code setup for testing.

KCSS operations to disjoint locations of the list. The first thread, which we will call T_1, will attempt to delete node `d` from the list (using operation `delete_d()` on Lines 13 to 23), while the second thread (T_2) will insert node `b` containing integer `2` (using operation `insert_b()` on Lines 24 to 33). These operations simply make use of the `kcss` operation itself, with variable arguments. Inserting `b` will require KCSS with `K = 1`, because the only location whose contents need to remain unmodified while we insert `b` is node `a` itself. After a successful execution of the `kcss` operation inserting

b, node a->next field will no longer point to node c but point to b instead; and b->next will point to node c because it has been initialized as such, and node c is guaranteed to not have been deleted in the course of kcss's execution because this would have required modifying a concurrently.

With respect to method delete_d() (Lines 13 to 23 on Figure 5.21), it requires KCSS with K = 2, because not only do we have to ensure that the node preceding d is not modified, but also that d is not undergoing another concurrent operation e.g. an insertion of a new node after it. A remarkable aspect on this method is the use of auxiliary method KCSS::mp(). This method stands for "make pair", and it is necessary in order to provide arguments to kcss besides from the initial location and its expected and new value. A call to KCSS::mp() is needed for every new location involved in KCSS, and every pair created has to be made up of the location pointer, and its expected value (see the C++ source code for KCSS::mp() at Figure 5.19).

```
1  template<typename T>
2  constexpr static std::pair<loc_struct_t<T>&, T> mp(loc_struct_t<T
    > &a, T b) noexcept {
3    return std::move(std::pair<loc_struct_t<T>&, T>(a, b));
4  }
```

Figure 5.19: KCSS::mp() method C++ code.

The final step in test method two_threads() is to generate two virtual threads, assigning a task to each one among the aforementioned insert_b() and delete_d() (Lines 35 to 38 on Figure 5.21), and waiting for them to finish executing (Lines 40 and 41). The expected output is the successful deletion of node d and successful insertion of b, because this locations are disjoint and therefore concurrent calls to KCSS will not cause invalidations of a thread's LL/SC calls.

This would not be the case if we executed test n_threads(), because in it, n threads are spawned to concurrently attempt to increment by one the value of a location v1. Without getting into further implementation details, we can say that the expected behaviour in this test case will be observing that the value at location v1 has not been incremented at all, or at most once. This is because the n concurrently executing threads will prevent each others' KCSS operations from succeeding because they all involve the same location. Since KCSS is *obstruction-free*, no thread is guaranteed to make progress unless left to run without interference from others. This is precisely why we might expect one increment to be successful but not more: this will correspond to the last attempt to increment v1 by the last thread to finish executing, when the rest have already terminated and it is left to run in isolation.

5.7. Concluding Remarks

We finish this chapter with some concluding remarks. Note that KCSS's mechanism, which is based on the CAS operation, is correct because this operation is only

```
void one_thread() {

  // Initial list: a(1) -> b(2) -> c(3) -> e(5)
  node *e = new node(5, nullptr);
  node *c = new node(3, e);
  node *b = new node(2, c);
  node *a = new node(1, b);

  std::cout << "Creating initial list consisting of:\n";
  print_list(a);

  // We want to delete node e(5). We will make use of KCSS where K = 2.
  //  The first location involved is c->next, and the second one is e->next
  std::cout << "Deleting node e(5)...\n";
  // Apply kcss operation:
  bool success = kcss_instance.kcss(c->next, e, (node*) nullptr, KCSS::mp(e->next, (node*) nullptr));

  if(success){ // Now it is safe to delete node e
    std::cout << "Deletion operation successful. Current list:\n";
    print_list(a);

    delete(e);
  }
  else{ // Failure of kcss
    std::cout << "Deletion operation unsuccessful. Current list:\n";
    print_list(a);

    // ... (possibly retry) ...
  }

  // Perform the desired actions with the list (...)

  cleanup(a);
}
```

Figure 5.20: KCSS operation test using one thread over a linked list, on C++ code.

ever applied to values stored in the `loc_struct_base->value` field. This field has to comply with the memory requirements imposed by CAS operations, but this is immediate thanks to the KCSS value-format given by *program* and *temporal* values' bit-wise restrictions, which ensure memory locations are word-aligned and occupy 64bits (1 memory word). On the other hand, the KCSS operation requires CAS to be *linearizable*. We are using version

```
bool __sync_bool_compare_and_swap (T *ptr, T oldVal, T newVal)
```

of CAS, whose implementation is based on the assembly instruction CMPXCHG. This operation is *atomic* by construction, and therefore *linearizable*, so this second constraint is also satisfied.

Remember that we also mentioned in Section 4.7 that we require *linearizable* `Load` and `Store` operations on memory. In fact, this requirement can be slightly relaxed and the `KCSS` algorithm is correct in multiprocessor systems that provide only the TSO memory model[2]. For multiprocessor systems that do not comply with it by construction, `C++`'s `atomic` library provides `Load` and `Store` operations taking as parameter the desired memory model ordering. To comply with the needs of `KCSS`'s algorithm, the `Release-Acquire ordering` model should be employed on any `Load` and `Store` operation over field `loc_struct_base->value`, which is the only one that could cause data races among threads due to being concurrently accessed.

Additionally, it is worth mentioning that in use cases where `KCSS` is needed with `K = 1`, the functionally equivalent `CAS` primitive is more performing and should therefore be chosen instead. Moreover, in use cases where `K = 2` we suggest implementing a `2CSS` method independent from the `KCSS` class in the following way: we can replace the `SNAPSHOT` operation collecting `K - 1 = 1` values and timestamps by a simple `READ` operation. This `2CSS` operation will also entail an efficiency improvement with respect to employing the original `KCSS` operation.

To conclude this chapter, we propose a code improvement to our own `C++` proposal that would speed up execution of the `KCSS` algorithm (this improvement has not been directly applied to the code explained in previous sections to not downgrade readability). It has to do with certain operations that consist of a loop carrying out a fixed number of iterations. Due to their nature, this operations can be redefined so that the compiler unfolds every iteration and encrusts it directly in the compiled code (instead of simulating the loop by jumping back and forth the corresponding loop instructions). An example of this technique can be seen on Figure 5.24 for the `COLLECT_VALUES` operation. As we can see, the new definition of this operation consists of a templated call taking a number `k` as argument, which corresponds to the remaining number of times that the template has to be unfolded. This redefinition concludes with the "base case" where `k = 0` (Line 8 on Figure 5.24), which corresponds to the case when all iterations have been unfolded.

The same technique can be applied to method `COLLECT_SNAPSHOT_TS`, to the `SNAPSHOT` operation as well as to `SNAPSHOT`'s inner loop iterating over the collected values and tags to check for inequalities, and to `KCSS` itself. The code for the redefined `COLLECT_SNAPSHOT_TS` can be found at Figure 5.25, the corresponding code for the redefined `SNAPSHOT` operation and its inner equality evaluation condition can be found at Figure 5.26, and that of `KCSS` at Figure 5.27.

[2]For more information on the TSO memory model refer to (SPARC International, 1994), pages 266 (RMO, TSO) and 186 (Memory Barriers).

```
void two_threads() {

  // Initial list: a(1) -> c(3) -> d(4)
  node *d = new node(4, nullptr);
  node *c = new node(3, d);
  node *a = new node(1, c);

  std::cout << "Creating initial list consisting of:\n";
  print_list(a);

  // We want T1 to delete node d(4), while T2 inserts b(2).

  auto delete_d = [&]() { // Will make use of KCSS where K = 2.
    //  The first location involved is c->next, and the second one is d->next
    std::cout << "Deleting node d(4)...\n";
    // Apply kcss operation:
    bool success = kcss_instance.kcss(c->next, d, (node*) nullptr, KCSS::mp(d->next, (node*) nullptr));
    if(success){ // Now it is safe to delete node e
      std::cout << "Deletion operation successful.\n";
    }else{ // Failure of kcss
      std::cout << "Deletion operation unsuccessful.\n";
    }
  };
  auto insert_b = [&]() { // Will make use of KCSS where K = 1.
    node *b = new node(2, c);
    std::cout << "Inserting node b(2)...\n";
    bool success = kcss_instance.kcss(a->next, c, b);
    if(success){ // Now it is safe to delete node e
      std::cout << "Insertion operation successful.\n";
    }else{ // Failure of kcss
      std::cout << "Insertion operation unsuccessful.\n";
    }
  };

  int num_threads = 2;
  std::thread t[num_threads];
  t[0] = std::thread(delete_d);
  t[1] = std::thread(insert_b);

  t[0].join();
  t[1].join();


  std::cout << "Current list:\n";
  print_list(a);

  // Perform the desired actions with the list (...)

  cleanup(a);
}
```

Figure 5.21: KCSS operation test using two threads over a linked list, on C++ code.

```
void n_threads(std::size_t n) {

  KCSS::loc_struct_t<int> v1(10);
  KCSS::loc_struct_t<int> v2(20);
  KCSS::loc_struct_t<int> v3(30);

  auto f = [&]() {
    while (true) {
      int x = kcss_instance.get(v1);
      if (x > 100000)
        break;
      kcss_instance.kcss(v1, x, x + 1, KCSS::mp(v2, 20), KCSS::mp(
    v3, 30));
    }
  };

  std::thread t[n];
  for (auto i = 0u; i < n; i++)
    t[i] = std::thread(f);
  for (auto i = 0u; i < n; i++)
    t[i].join();


  std::cout << "\nThe final value of v1 is: " << kcss_instance.get(
    v1) << std::endl;

}
```

Figure 5.22: KCSS operation test using n threads concurrently attempting to modify a single location, on C++ code.

```
int main(){

  std::cout << "\nTest 1\n";
  one_thread();

  std::cout << "\nTest 2\n";
  two_threads();

  std::cout << "\nTest 3\n";
  n_threads(6);

  return 0;
}
```

Figure 5.23: KCSS operation testing main() method, on C++ code.

```
template<std::size_t k>
inline void collect_values_(loc_struct_base **a, uint64_t *v)
 noexcept {
  collect_values_<k - 1>(a, v);
  v[k - 1] = read(a[k - 1]);
}

template<>
inline void collect_values_<0>(loc_struct_base**, uint64_t*)
 noexcept {
}
```

Figure 5.24: Redefinition of operation `COLLECT_VALUES`, on `C++` code.

```
template<std::size_t k>
inline constexpr void collect_snapshot_ts_(loc_struct_base **a,
    uint64_t *t) const noexcept {
  collect_snapshot_ts_<k - 1>(a, t);
  t[k - 1] = a[k - 1]->snapshot_ts;
}

template<>
inline constexpr void collect_snapshot_ts_<0>(loc_struct_base**,
    uint64_t*) const noexcept {
}
```

Figure 5.25: Redefinition of operation `COLLECT_SNAPSHOT_TS`, on `C++` code.

```
1  // eval_cond_ method for redefined snapshot_
2  template<std::size_t k>
3  inline bool eval_cond_(uint64_t *values_1, uint64_t *values_2,
    uint64_t *timestamps_1,
4      uint64_t *timestamps_2) noexcept {
5    return eval_cond_<k - 1>(values_1, values_2, timestamps_1,
    timestamps_2)
6           && timestamps_1[k - 1] == timestamps_2[k - 1]
7           && values_1[k - 1] == values_2[k - 1];
8  }
9
10 template<>
11 inline bool eval_cond_<0>(uint64_t*, uint64_t*, uint64_t*,
12     uint64_t*) noexcept {
13   return true;
14 }
15
16 // snapshot_ method
17 template<std::size_t k>
18 inline void snapshot_(loc_struct_base **a, uint64_t *values_1)
    noexcept {
19   uint64_t timestamps_1[k], timestamps_2[k];
20   uint64_t values_2[k];
21
22   while (true) {
23     collect_snapshot_ts_<k>(a, timestamps_1);
24     collect_values_<k>(a, values_1);
25     collect_values_<k>(a, values_2);
26     collect_snapshot_ts_<k>(a, timestamps_2);
27
28     if (eval_cond_<k>(values_1, values_2, timestamps_1,
    timestamps_2))
29       return;
30   }
31
32   return;
33 }
34
35 template<>
36 inline void snapshot_<0>(loc_struct_base**, uint64_t*) noexcept {
37 }
```

Figure 5.26: Redefinition of operation `SNAPSHOT`, on `C++` code.

```
template<typename T0, typename ...Ts>
bool kcss_(loc_struct_t<T0> &a0, T0 a0_expv, T0 a0_newv, Ts &&...
 args) noexcept {

  constexpr std::size_t k = sizeof...(args) + 1;

  uint64_t old_vals[k];
  uint64_t exp_vals[k] = { a0.to_value_t(a0_expv),
      (args.first.to_value_t(args.second))... };
  loc_struct_base *a[k] = { &a0, (&args.first)... };
  uint64_t new_v = a0.to_value_t(a0_newv);

  while (true) {
    old_vals[0] = ll(a[0]);

    snapshot_<k - 1>(a + 1, old_vals + 1);
    if (eval_kcss_cond_<k>(old_vals, exp_vals)) {
      sc(a[0], old_vals[0]);
      return false;
    }
    //The previous code block is equivalent to the following:
    //  snapshot(k - 1, a + 1, old_vals + 1);
    //  for (unsigned int i = 0; i < k; ++i) {
    //    if (old_vals[i] != exp_vals[i]) {
    //      sc(a[0], old_vals[0]);
    //      return false;
    //    }
    //  }

    if (sc(a[0], new_v)) {
      return true;
    }
  }

  return false;
}
```

Figure 5.27: Redefinition of operation KCSS, on C++ code.

```
template<std::size_t k>
inline bool eval_kcss_cond_(uint64_t *old_values, uint64_t *
 exp_values) noexcept {
  return eval_kcss_cond_<k - 1>(old_values, exp_values)
      || old_values[k - 1] != exp_values[k - 1];
}

template<>
inline bool eval_kcss_cond_<0>(uint64_t*, uint64_t*) noexcept {
  return false;
}
```

Figure 5.28: Auxiliary condition evaluation operation for the redefinition of operation KCSS, on C++ code.

Chapter 6

Conclusions and Future Work

Through this work we have motivated the need for further research on tools that exploit the capabilities of multi-core systems, concurrency being the answer to Moore's Law standstill. Analysing the challenges that Concurrent-Data-Structure design faces, we have understood the delicate balance between efficient communication and autonomy between concurrently executing threads, and how perfect timing of their actions is key for correctness.

The study of fine-grained and coarse-grained software synchronisation mechanisms has allowed us to understand the nature of concurrent settings both from the practical and the theoretical point of view, establishing a knowledge baseline to guide the reader on switching from sequential to concurrent reasoning. Coarse-grained mechanisms provide simple solutions at the cost of efficiency and adaptability. Fine-grained mechanisms tip the balance towards the opposite side, in exchange for higher code complexity that makes the study of these concurrency mechanisms a delicate task.

At this point, the `KCSS` software primitive has been suggested as an adequate commitment between efficiency and adaptability, and code complexity and usability. A profuse explanation of every relevant and non-trivial feature of `KCSS` has been assembled, to serve as an educational resource easing the understanding of its inner workings. We have explored `KCSS`'s applicability to the design of Concurrent Data Structures, more specifically Concurrent **Linked Data Structures**, emphasising its performance with respect to alternative solutions to the synchronisation challenges posed by these Data Structures.

To bring to light the real tangible impact of this primitive, a fully-functional, efficient and transparent-to-the-user `KCSS` implementation has been provided. This task has challenged our initial commitment to make it easy to use, due to the strict memory-location interpretation needs of `KCSS` and our desire to make it available for all common data types. Nevertheless, `C++`'s ample capabilities (templates, bit fields, function parameter packs) have allowed us to hide all implementation details from the user while maintaining efficiency and coverage of all common data types.

Finally, we have studied how this primitive can be further improved, and how re-

lated work on multi-location software synchronisation primitives could complement it. In particular, the ideas put forward to design KCSS can be put into practice to develop a transactional model with the same functionality as KCSS but improved efficiency and intuitiveness of the process. The main idea is to have *transactional loads* in charge of recording the information collected in the first half of the SNAPSHOT operation that our current KCSS implementation performs, and *transactional commits* doing the second part of this operation: determining if any of the values read has been modified by a concurrent operation since being read by the *transactional load*.

KCSS could also yet undergo further concurrency improvements: it is possible to reduce "collisions" between LL/SC operations at the cost of a more complex SNAPSHOT operation. It is also possible to reuse collected values and timestamps by this operation from one iteration to the other, downgrading readability but improving overall efficiency.

Conclusión

A través de este trabajo hemos motivado la necesidad de seguir investigando herramientas que exploten las capacidades de los sistemas multinúcleo, siendo la concurrencia la respuesta al estancamiento de la Ley de Moore. Al analizar los desafíos a los que se enfrenta el diseño de Estructuras de Datos Concurrentes, hemos comprendido el delicado equilibrio entre tener comunicación eficiente y autonomía entre hilos ejecutándose simultáneamente, y cómo la sincronización perfecta de sus acciones es clave para la corrección.

El estudio de los mecanismos de sincronización de software de granularidad fina y gruesa nos ha permitido comprender la naturaleza de los entornos de ejecución concurrentes tanto desde el punto de vista práctico como teórico, estableciendo una base de conocimiento para guiar al lector en el cambio de razonamiento secuencial a concurrente. Los mecanismos de granularidad gruesa proporcionan soluciones simples a costa de la eficiencia y la adaptabilidad. Los mecanismos de granularidad fina inclinan la balanza hacia el lado opuesto, a cambio de una mayor complejidad del código que hace que el estudio de estos mecanismos de concurrencia sea una tarea delicada.

En este punto, la primitiva de software `KCSS` se ha sugerido como un compromiso adecuado entre eficiencia y adaptabilidad, y complejidad y facilidad de uso del código. Se ha elaborado una explicación detallada de cada característica relevante y no trivial de `KCSS`, sirviendo como recurso educativo que facilita la comprensión de su funcionamiento interno. Hemos explorado la aplicabilidad de `KCSS` al diseño de Estructuras de Datos Concurrentes, más específicamente **Estructuras de Datos Enlazadas** Concurrentes, enfatizando su desempeño con respecto a soluciones alternativas a los desafíos de sincronización planteados por estas Estructuras de Datos.

Para sacar a la luz el impacto tangible real de esta primitiva, se ha proporcionado una implementación de `KCSS` completamente funcional, eficiente y transparente para el usuario. Esta tarea ha desafiado nuestro compromiso inicial de hacerla fácil de usar, debido a las estrictas necesidades de `KCSS` al interpretar las ubicaciones de memoria y debido a nuestro deseo de que sea compatible con todos los tipos de datos comunes. Gracias a las amplias capacidades de `C++` (plantillas, campos de bits, paquetes de parámetros de función) hemos podido ocultar los detalles de im-

plementación al usuario, manteniendo la eficiencia y cobertura de todos los tipos de datos.

Finalmente, hemos estudiado cómo se puede mejorar aún más esta primitiva y cómo el trabajo relacionado con las primitivas de sincronización de software de ubicación múltiple podría complementarla. En particular, las ideas propuestas para diseñar `KCSS` se pueden poner en práctica para desarrollar un modelo transaccional con la misma funcionalidad que `KCSS` pero con mayor eficiencia y siendo más intuitivo. La idea principal es tener *lecturas transaccionales* (*transactional loads* en inglés) a cargo de registrar la información recopilada en la primera mitad de la operación `SNAPSHOT` que actualmente realiza nuestra implementación de `KCSS`, y *guardados transaccionales* (*transactional stores*) realizando la segunda parte de esta operación: determinar si alguno de los valores leídos ha sido modificado por una operación concurrente desde que fue leído por la *lectura transaccional*.

`KCSS` también podría experimentar mejoras de su grado de concurrencia: es posible reducir las "colisiones" entre operaciones `LL/SC` a costa de una operación `SNAPSHOT` más compleja. También es posible reutilizar los valores recopilados y los sellos de tiempos (*timestamps*) recopilados por esta operación, de una iteración a la otra, lo que reduce la legibilidad pero mejora la eficiencia general.

Bibliography

Barnes, G. A method for implementing lock-free shared-data structures. In *Proceedings of the fifth annual ACM symposium on Parallel algorithms and architectures - SPAA '93*. ACM Press, 1993.

Barreira, J. A. C. Multiprocessors: Coherence and synchronization, computer architecture course slides. 2023.

Doherty, S., Herlihy, M., Luchangco, V. and Moir, M. Bringing practical lock-free synchronization to 64-bit applications. In *Proceedings of the twenty-third annual ACM symposium on Principles of distributed computing*. ACM, 2004.

Greenwald, M. B. Non-blocking synchronization and system design. 1999.

Harris, T. L., Fraser, K. and Pratt, I. A. A practical multi-word compare-and-swap operation. In *Lecture Notes in Computer Science*, 265–279. Springer Berlin Heidelberg, 2002.

Herlihy, M., Eliot, J. and Moss, B. Transactional memory: Architectural support for lock-free data structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. IEEE Comput. Soc. Press, ????

Herlihy, M., Luchangco, V., Moir, M. and Scherer, W. N. Software transactional memory for dynamic-sized data structures. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*. ACM, 2003.

Herlihy, M. and Shavit, N. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. ISBN 0123705916.

Luchangco, V., Moir, M. and Shavit, N. Nonblocking k-compare-single-swap. *Theory of Computing Systems*, Vol. 44(1), 39–66, 2008.

Moir, M. and Shavit, N. Concurrent data structures. In *Handbook of Data Structures and Applications*, 47–1–47–30. Chapman and Hall/CRC, 2004.

Shavit, N. and Touitou, D. Software transactional memory. *Distributed Computing*, Vol. 10(2), 99–116, 1997.

SPARC International, C., Inc. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., USA, 1994. ISBN 0130992275.