9 Parallel data structures

Contemporary computers often have multiple processors.[1] To utilize all available computing power, programs typically start multiple *processes* or *threads*, which run in parallel on different processors. This brings a new challenge: designing data structures which support concurrent access. These are often called *parallel* or *concurrent* data structures.

Parallel computing involves a lot of technical details. We tried to keep the exposition as straightforward as possible, but you can find many details in footnotes.

9.1 Parallel RAM

Details of multi-processor computers are quite complex and they vary between machines. Therefore, we will study parallel algorithms in a simple theoretical model instead, like we did with sequential algorithms. Our model is called the *Parallel RAM (PRAM).* It consists of several instances of the Random-Access Machine. We will call each instance a *processor.*

Instructions of the machine can access data in *local memory* of the current processor and also in *global memory* shared among all processors. There are no instructions for computing directly with data in global memory — we always have to read the data to local memory, modify them there, and write them back to global memory.

All processors execute the same program. However, the program has access to the identifier of the current processor, so it can easily switch to different branches of code on different processors. The sequences of instructions executed by the processors are called *processes.*

Many variants of the PRAM model guarantee that the instructions are executed in lockstep – in a single tick of a global system clock, each processor executes a single instruction of its process. As this is not true on real hardware, we will make no such assumption and allow each processor to execute instructions at its own pace.

It remains to specify what happens when multiple processes access the same cell of the global memory simultaneously. Concurrent reads will be considered correct. Concurrent writes and combinations of reads and writes will have undefined behavior.

[1] Sometimes, these are called *cores* or *hardware threads,* but this is a purely technical difference. Unless you are programming at a very low level, these behave as separate processors.

You can argue that it is almost impossible to produce correct parallel programs on machines with such weak semantics. We agree, so we are going to extend the machine later in this chapter by introducing *locks* and *atomic operations* with well-defined semantics.

9.2 Locks

Let us start small. Imagine a simple counter in global memory called *cnt* with an increment operation. Since we cannot compute in global memory, the increment must consist of three steps:

Procedure GlobalInc

1. $t \leftarrow cnt$
2. $t \leftarrow t + 1$
3. $cnt \leftarrow t$

On a single processor, it is certainly correct. But consider that $cnt = 1$ and two processors try to increment it concurrently. It can happen that both read *cnt* to their local variables, both increment the local variable to 2, and then both write 2 back to *cnt* (in either order). So the end result is 2 instead of 3.

This is a prototypical example of what is called a *race condition* – the result of a computation depends on the exact ordering of operations across processors.

Race conditions are frequently avoided using *synchronization primitives.* The simplest such primitive is a *mutex* (short for "mutual exclusion", sometimes also called a *lock*). At any given moment, the mutex is either unlocked or locked. It supports two operations:

- Lock – If the mutex is unlocked, lock it. If it is locked, wait until it is unlocked by somebody else and then lock it.
- UnLock – If the mutex is locked, unlock it. If it is unlocked, crash (this should not happen in a correct program).

At this moment, we do not know how to implement the mutex. For the time being, we can consider it an extra feature of our machine. Later, we will show that it can be constructed from atomic instructions.

Typically, each instance of a data structure has its own mutex. Every operation on the structure is wrapped in an Lock/UnLock pair. This guarantees that at most one process is working with the data structure at any given time (this is the mutual exclusion). Hence, all operations are *atomic* — if we perform an operation, any other observer sees it either not started, or completely finished.

The dreaded deadlock

Problems arise if we want to perform atomic operations which affect more than one instance of the data structure. Consider the following situation: We have doubly linked lists, each list with its own mutex. Insertion and deletion is easy, but what if we want to move an item x from list A to list B atomically? (This means that for any outside observer, the item is always seen either in A, or in B.) An obvious solution would be:

Procedure AtomicMove

1. Lock the mutex of A.
2. Lock the mutex of B.
3. Delete x from A.
4. Insert x to B.
5. Unlock the mutex of B.[2]
6. Unlock the mutex of A.

Although this seems to be intuitively correct, it can fail badly. What if a process P_1 tries to move an item x from A to B, while another process P_2 tries to move y from B to A? It can happen that when P_1 obtains the mutex for A, P_2 obtains the mutex for B. In the next step, P_1 wants the mutex for B, while P_2 wants the mutex for A. So each process is waiting for a mutex held by the other process, and both processes will wait infinitely long.

This is called a *deadlock* and next to the race condition, it is the most frequent bug in parallel programs. Generally, more than two processes can participate in a deadlock. Let us consider the *dependency graph.* It is a directed graph, whose vertices are processes and an edge from i to j means that process i is waiting for a mutex currently locked by process j. If there is a directed cycle in this graph, the processes are obviously deadlocked. Otherwise, the computation can proceed from the source vertices of the graph (those with no incoming edges).

There is a simple solution to deadlocks: Establish an *ordering of all mutexes* in the system and always lock mutexes in increasing order. E.g., we can order them by their addresses in memory. This way, no deadlock can occur, because a cycle in the dependency graph would imply a cycle in the order. (In fact, the order need not be total — a partial order suffices as long as all locks taken by every single process are totally ordered.)

This technique can prevent all deadlocks as long as we can tell in advance, which mutexes will be needed by an operation (we have to sort them first). This is not true in all cases: for example, we might want to move an item to a list chosen according to the item's value.

[2] The order of unlocks is arbitrary, but we prefer to have the lock-unlock pairs nested properly.

Granularity of locks

Sometimes, one lock per instance is not the best possible granularity. If we have many small data structures, individual locks can consume too much memory. In such cases, we can have an array of mutexes and a hash function mapping an address of an instance to a mutex in the array. This can be very efficient, but when constructing operations working with multiple instances, beware that two instances can be mapped to the same mutex.

On the other hand, if all processes spend most of their time accessing a single instance, the lock guarding this instance will become a performance bottleneck. If it is so, we should try splitting the data structure to parts guarded by different locks. This is particularly easy if the data structure is a hash table: we can assign a separate mutex to each bucket. As long as different processes operate on different buckets (which is quite likely), they run with no contention. Beware that this setup does not support rehashing — the main pointer to the array of buckets is not guarded by a lock, so it must stay read-only.

We will see more examples of fine-grained locking in section 9.3.

Problems with locking

Locking is conceptually simple (though tricky to implement correctly), but it is not a panacea. There are multiple issues associated with locking:

- *Deadlocks* — operations which need to acquire more than one lock can cause deadlocks. As we saw, deadlocks can be usually prevented by ordering the locks, but it is not always possible.
- *Lack of composability* — intuitively, if operations α and β can be each performed atomically, it should be possible to do the composition $\alpha\beta$ atomically, too. This is not the case with locks, especially if we are not allowed to look inside α and β. Even if we are allowed, the problems with deadlocks arise. Full composability would require a completely different approach, for example transactional semantics. We will not discuss it here.
- *Fairness* — generally, there is no guarantee that a single process will finish in finite time. If there is a lot of contention at a lock, a process can wait indefinitely if other processes can always obtain the lock faster. This situation differs from the deadlock, because if other processes are stopped, the current process finishes. This is usually called *starvation.* It can be prevented by implementing the locks in a way which guarantees fairness.
- *Priority inversion* — if we have a system where processes have different priorities (e.g., real-time calculations have a high priority, while interaction with the user a lower one), a high-priority process can be blocked by a low-priority process if

the former waits on a lock held by the latter. The standard fix for that in real-time systems is *priority inheritance* — a process holding a lock has its priority raised to the maximum priority of processes currently waiting for the lock. This further complicates implementation of locks.

- *Performance* — although modern operating systems supply a well-optimized implementation of locks, it still slows down the program and consumes memory. Especially if locking is fine-grained.
- *Fault tolerance* — if a process is terminated abnormally, all locks it held are left locked forever. If any other process tries to access the data structures protected by these locks, it will wait forever. The operating system could break the locks forcefully, but that would make other processes access the data structure in a potentially inconsistent state.

Some (but not all) of these problems will be solved by lock-free data structures introduced in section 9.4.

9.3 Locking in search trees

Let us try to combine binary search trees with fine-grained locking. We add a mutex to each node of the tree in attempt to make operations as parallel as possible. We will consider operations Find, Insert, and Delete on the tree.

Locking a path

Trivial solution first. Whenever we want to operate on an item, we follow a path from the root downwards and we lock the nodes on the path as we visit them. Find just follows the path. Insert without balancing follows the path and adds a new leaf at the end of the path. Delete follows the path and it either finds a node with at most 1 child (which can be cut out), or a node with 2 children, which is to be replaced by its successor, but finding the successor just extends the path further down.

This is obviously correct as all decisions we make (to go to the left or to the right) remain valid until the end of the operation — all nodes which affected the decision are kept locked. Even better, we can add rebalancing, which traverses the same path from the bottom to the top.

It can be also easily proven that no deadlock can occur: we can order the nodes by their depth in the tree and keep nodes at the same depth incomparable. This is a partial order in which every downward path forms a chain.

However, there is one huge fault in this approach: every path contains the root. So all operations are serialized by the lock in the root and the other locks have no effect.

Locking a sliding window

If the only operations on the tree are FIND and INSERT with no balancing, there is no need to keep the nodes we already passed locked. It is sufficient to have locked only the current node and its child where we want to go. This corresponds to sliding a window of size 2 over the path and locking only the nodes inside the window.

In a single step of FIND(x), we compare x with the key in the current node and read the pointer to the corresponding child. We did this with the current node locked, so we do not race with other processes modifying the same node. Then we lock the child, unlock the current node and make the child the new current node. If we are INSERTing, we finish by creating a new node (it need not be locked, since we are the only process possessing a pointer to it) and attaching it as a child to the current node (whose lock we hold).

This approach allows much greater concurrency — even though all paths start at the root, the root is quickly unlocked and we can expect that the paths will soon diverge, especially when accessing random items. Still, if the number of processes exceeds the height of the tree, the contention at the root can be significant.

Deadlocks are still impossible, because we take the locks in the order of increasing depth in the tree.

However, it is not clear how to add DELETE and most importantly balancing. Traditional methods of balancing (AVL or red-black) do not work as they need to propagate changes upwards, possibly up to the root.

Arguing correctness: serializability

We would like to argue that our construction is correct, but we need to give an appropriate definition of correctness first. For example, what is the right answer if we are searching for an item, while another process is inserting an item with the same key?

The standard way of formulating correctness in concurrent systems is using *serializability.* A sequence of operations on a data structure is *serializable* if there exists a linear (total) order on all operations such that (1) the result of each operation is consistent with the operations preceding it in the order, (2) from the point of view of every process, the order of operations executed by that process is consistent with the total order.

Try to prove that the previous construction with INSERT and FIND is correct in the sense of serializability.

Sometimes, a stronger concept is used instead of serializability. It is called *sequential consistency* and it requires a single linear order common to all data structures. Reasoning in this model is much easier, but it is inefficient to implement on current hardware.

Top-down (a,b)-trees

The idea of locking by sliding a window over a path is much better suited to (a, b)-trees. We will use the version of (a, b)-trees with top-down balancing from section **??**.

In Insert, we hold a lock on the current node and its parent. In each step, we split the current node if needed. Then we unlock the parent, find the appropriate child, lock it, and move there.

Delete is similar, but besides the current node and its parent, we sometimes need to lock a sibling. This requires more careful ordering of locks to avoid deadlocks. The primary ordering will be still by level (depth), but at the same level we will order the locks from the left to the right. This can be a problem, if after examining the current node we decide to look at the left sibling. One solution is to always lock the left sibling before the current node, even if it is not accessed. For another, see exercise 2.

Another problem in Delete is deletion of a key which is not at the lowest level. This is usually solved by replacing the key by its successor, but when looking for the successor, we need to keep the current node locked. This can be slow (e.g., if the current node is the root, we are effectively locking the whole tree). If this is a problem, we can keep the key there and mark it as deleted.

Exercises

1. Modify top-down Delete so that it guarantees that at most a half of all keys is marked as deleted.

2. Prove that that we can safely lock the children of a common parent in arbitrary order if the parent is locked first.

9.4 Lock-free data structures

We already mentioned multiple problems with locking. In this section, we will investigate data structures that do not need locks.

Atomic instructions

Without locks, we need to achieve consistency by other means. Fortunately, real machines typically offer a small set of instructions which guarantee atomicity of some kind. We can add them to our parallel RAM, too. Here are typical examples of *atomic instructions.* They usually operate on cells of global memory called *atomic registers.* Operations on individual atomic registers are serializable.

- *Read and write* — the most basic guarantee is that an atomic register can be read or written as a whole. Concurrent accesses to the same atomic register are resolved in an unknown, but consistent order.⟨3⟩
- *Exchange* — exchange contents of an atomic register with a value in local memory. This is sufficient for implementing a mutex: an unlocked mutex will have a zero value. Locking the mutex will exchange the register with 1. If the original value is zero, we have succeeded. If it was 1, we retry. To unlock the mutex, we write a zero atomically.⟨4⟩
- *Test and set bit* — set a given bit of an atomic register and return its original value. This is another popular building block for locks.
- *Fetch and add* — add a number to an atomic register and return its original value.
- *Compare and swap (CAS)* — it is given an atomic register R and values *old* and *new*. If R equals *old*, it is changed to *new*. Otherwise R is not changed. In all cases, the original value of R is returned.
- *Load linked and store conditional (LL/SC)* — a linked load is a normal atomic read, but the processor remembers the address and keeps monitoring it for access by other processors. A later store conditional to the same address succeeds if the address was not written to by other processors. Otherwise it reports failure.⟨5⟩

Current hardware implements atomic read/write and either CAS or LL/SC. The other atomic operations are either implemented or they can be simulated using CAS or LL/SC. CAS can be simulated using LL/SC, but not vice versa — the difference will play an important role in the next section.

Lock-free stack

Let us build a simple lock-free implementation of a stack. It will be represented as a linked list of *nodes.* Each node will contain data of the item and an atomic pointer to the *next* item. We will also keep an atomic pointer to the *head* of the list (the most recent item in the stack).

⟨3⟩ Surprisingly, real hardware does not guarantee atomicity for normal memory accesses. For example, a 64-bit value on a 32-bit machine has to be written by two instructions, so another process can see a partially written value. More subtly, writes which cross a cache block boundary are also not atomic.

⟨4⟩ On a real operating system, we usually want to let the process sleep instead of actively checking the register in a loop. For our simplistic introduction, spinning in a loop will suffice. This known as a *spinlock.*

⟨5⟩ Real machines have various restrictions on LL/SC. Usually, only a very limited number of addresses can be monitored simultaneously. Also, monitoring is often based on cache blocks, so a write to another variable within the same cache block can also trigger SC failure.

We will implement the operations Push (insert a new item at the top of the stack) and Pop (remove an item from the top of the stack) as follows.

Procedure Push
Input: A new node n containing the item to be pushed.

1. Repeat:
2. $h \leftarrow stack.head$
3. $n.next \leftarrow h$
4. If CAS$(stack.head, h, n) = h$:
5. Return.

Procedure Pop

1. Repeat:
2. $h \leftarrow stack.head$
3. $s \leftarrow h.next$
4. if CAS$(stack.head, h, s) = h$:
5. Return h.

Output: A node removed from the top of the stack.

The CAS guarantees that if another process interferes with the operation, we detect the interference and restart the operation. In the worst case, we can loop indefinitely and never complete the operation — this is called a *livelock*. In practice, the livelock is extremely improbable, because there is a lot of random factors which affect scheduling of processes, so every process eventually succeeds.

The ABA problem

Livelocks aside, the implementation looks correct. It is tempting to prove that it is serializable by the order of CASes on the list head. There is however one subtle hole in this argument: it works only if we assume that all nodes ever pushed to the stack are distinct.

Let us see what can happen if they are not. We start with the situation displayed in the figure: a stack containing items A (at the top) and B.

One process performs:

Procedure Process1

1. $x \leftarrow$ Pop ◁ *We have* $x = A$.
2. $y \leftarrow$ Pop ◁ *We have* $y = B$.
3. Push(x) ◁ *We push* A *back to the stack.*

Another process starts a Pop, but it will be slower. It manages to perform steps 2 and 3 before Process1 starts popping, but step 4 will proceed after Process1 is done. So the second process sets $h = A$ and $s = B$. When it executes its CAS, the *head* is A again, so the *head* is changed to B. But this is wrong — B should not be in the list any longer as it was already removed by Process1's second Pop.

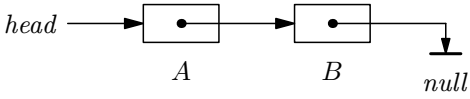

Figure 9.1: The list before Process1 starts

The core of the problem is that the original node A was confused with a logically different node stored at the same address.

This is called *the ABA problem* and it is a very typical fault of concurrent data structures. Let us consider some solutions:

- Use LL/SC instead of CAS. If step 2 of Pop is a LL and step 4 is a SC, the SC fails even if the *head* changed from A to B and back in the meantime.
- Have the machine support a *double-CAS* (alias DCAS or CAS2), which is a CAS on a pair of atomic registers simultaneously. Then we can replace the CAS in step 4 by

  $$\mathrm{CAS2}(\langle stack.head, h.next\rangle, \langle h, n\rangle, \langle n, n\rangle),$$

  which detects that the head node was re-connected and it has a different successor now. Alas, no current processor supports CAS2.
- Have the machine support a *wide CAS* (WCAS), also called *double-width CAS* (DW-CAS). It is a CAS2 on two variables which are *adjacent* in memory. This can be used for *versioning of pointers:* each pointer will be accompanied by an integer version, which will be incremented every time the pointer is changed.[6] Then step 2 becomes

  $$\langle h, v\rangle \leftarrow \langle stack.head, stack.head\_version\rangle$$

  and the test in step 4 becomes

  $$\text{If } \mathrm{WCAS}(\langle stack.head, stack.head\_version\rangle, \langle h, v\rangle, \langle n, v+1\rangle) = \langle h, v\rangle .$$

[6] The version number can overflow, but as long as it does not wrap around in a single invocation of Pop, overflows are harmless. Also, a 64-bit version number is not likely to overflow within the lifetime of our program.

Unlike general CAS2, WCAS is often supported by hardware.

- Avoid recycling of nodes: for every Push, allocate a new node.

Memory allocation

Another subtle issue with concurrent data structures is memory allocation. With a locking data structure, we can free memory used by a node once we Pop the node off the stack. In the lock-free version, this could have disastrous consequences: even though the node is already popped, other processes can still access it. This happens if they obtained a pointer to the node before its Pop finished. Later, their CAS will reveal that the node is no longer present. But in the meantime, they might have accessed invalid memory and crashed.

The usual solution is to collect all unused nodes in a *free list* — an atomic list managed like our lock-free stack — and free them after we make sure that no processes reference them. There are multiple possibilities:

- *Global synchronization* — from time to time, we synchronize all processes at a point where they hold no pointers of their own, and free up all chunks from the free list. This is simple, but in many cases the synchronization points are hard to find.

- *Reference counting* — in every node, we keep an atomic counter of references to the node (i.e., local variables which point to the node). Again, we keep a free list. From time to time, we scan the free list and free up all nodes with zero references.

  Time to scan the list can be amortized nicely. If we have P processes and each of them holds at most R references, there are at most RP references at any given moment. Therefore if we start the scan once the list accumulates at least $2RP$ items, we always free up at least a half of the nodes in the list. So time spent on scanning can be charged on the freed nodes, each node paying $\mathcal{O}(1)$ time.

  Maintenance of the reference counters is surprisingly tricky. In the short time window before we read a pointer to a node from memory and increment the node's reference count, the node might have been already freed. We can figure out that this happened by re-reading the pointer. In this case, we decrement the counter back and retry. But if the node's memory was already recycled for different use, we might have temporarily corrupted an unrelated data structure. To avoid this, we will recycle memory only as nodes of the same memory layout with the reference counter at a fixed position.

  Let us modify Pop to handle reference counting:

Procedure PopRefCnt

1. Repeat:
2. $h \leftarrow stack.head$
3. Increment $h.ref\_cnt$.
4. If $h \neq stack.head$:
5. Decrement $h.ref\_cnt$ and retry the loop.
6. $s \leftarrow h.next$
7. if CAS$(stack.head, h, s) = h$:
8. Decrement $h.ref\_cnt$ and return h.
9. Decrement $h.ref\_cnt$.

Output: A node removed from the top of the stack.

Note that we can safely decrease the *ref_cnt* in step 8, because as the „owner“ of the popped item, we are the only process which can free it. By the same argument, Push need not be modified because it references only the new node, which is owned by the current process.

- *Hazard pointers* – instead of keeping track of „hazardous references“ in each node, we collect all hazardous references in a single global array. The array is usually split to fixed-size blocks, each owned by a single process.

  Maintenance of hazard pointers is similar to that of reference counters. When a process wants to access a node, it sets one of its hazard pointers to that node. As with reference pointers, it is necessary to re-check that the node is still connected to the data structure.

  Let us see a version of Pop with hazard pointers. Again, Push need not be modified.

  Procedure PopHP

  1. Repeat:
  2. $h \leftarrow stack.head$
  3. $hp \leftarrow h$ ◁ *One of our hazard pointers*
  4. If $h \neq stack.head$:
  5. Retry the loop. ◁ *No need to reset hp here*
  6. $s \leftarrow h.next$
  7. if CAS$(stack.head, h, s) = h$:
  8. Let $hp \leftarrow \emptyset$ and return h.

  Output: A node removed from the top of the stack.

  When we want to scan the free list, we take a snapshot of the hazard pointer array and build a static data structure for the set of hazard pointers (in the simple case,

it is a sorted array with binary search). For each node in the free list, we query the data structure to see if the node is still being accessed.

It takes some effort to prove that no node can be freed under our hands:

- Before a node enters the free list, it was disconnected from the data structure by a PopHP.
- If any concurrent PopHP reached step 6 with h pointing to the node, the node was not in the free list yet.
- So the hazard pointer in the concurrent PopHP is set before the node enters the free list.
- Hence the snapshot taken when freeing memory includes all relevant hazard pointers.

Hierarchy of concurrent data structures

Concurrent data structures differ in guarantees they provide. The typical guarantees form a hierarchy with each level stronger than the preceding one:

- *blocking* – the data structure is correct, but an operation can wait indefinitely, for example for a lock held by another process.
- *obstruction-free* — if all other processes stop, my operation will succeed in finite time.
- *lock-free* — if multiple processes execute operations, at least one of them will succeed in finite time. However, there is no guarantee of fairness — livelocks are allowed.
- *wait-free* — every operation is guaranteed to succeed in finite time.
- *bounded wait-free* — in addition to that, there is an upper bound on the time (typically a function of the number of processes competing for the data structure).

Our stack is indeed lock-free, but not wait-free.

Trouble with hardware and compilers*

In our discussion of concurrency, we made several simplifying assumptions, which are not always valid on real computers. We briefly mention what can go wrong, but will leave the details to texts on low-level programming.

Most importantly, we assumed that all observers see the same order of writes to global memory: if we write a node to the memory and then we make an atomic register point to this node, everybody who sees the new value of the atomic pointer will also see the

contents of the node. This is generally not true in practice — writes to memory can be re-ordered by the hardware and different observers can see different order of writes.

To alleviate this problem, hardware usually supports *memory barrier* instructions. Memory operations issued before the barrier will complete earlier than memory operations issued after the barrier.⟨7⟩

Locking data structures need not care, because locks implicitly include memory barriers. Lock-free data structures typically require explicit barriers. Details vary between machines.

Also, reasoning about concurrent programs can be badly broken by program optimizations performed by compilers. Most compilers assume sequential semantics, so for example:

```
x = 1;
while (x == 1)
        // Do something, which does not involve x.
```

will be compiled to an infinite loop, even though another process can modify `x` and thus interrupt the loop. Another example is:

```
if (node_valid) {
        x = node;
        // Do something with x.
}
```

In sequential semantics, this can be rewritten as:

```
x = node;
if (node_valid) {
        // Do something with x.
}
```

which is not equivalent in a concurrent program.

To avoid problems of this type, compilers must be made aware that the variables can be accessed by other processes. For example, recent standards of the C and C++ languages define a formal memory model with concurrency, which includes explicit types for atomic variables.

⟨7⟩ This is called a *full barrier*. There are also weaker barriers, for example one which orders writes, but not reads. Weaker barriers are faster than full ones.