# 20. Hardware Assisted Synchronization

We hereby explore different OS/hardware assisted synchronization primitives. We briefly discuss how their implementation can be carried out and focus on their semantics. Deep inside, modern computing systems are far more complex than what the average “programmer” sees. Consider the following segment of code:

```c
int main(void)
{
    int a = 10;
    int b = 2;
    a = a * 10;
    b = b + 10;
    return a + b;
}
```

Now, you have learned in elementary school that execution of code is sequential, meaning that the underlying hardware will first perform the multiplication on variable a, and then the addition on variable b. On modern systems, this is not true, not even in an approximate sense. This property is called **out-of-order execution**. All the high-end processor architectures nowadays (Intel, ARM, PowerPC, ...) are out-of-order (OOO). OOO processors have complicated machinery to guarantee that while things can go OOO under the hood, they will “appear” as in order on the surface, with respect to the code of the considered program. If you want, this is a sort of contract between hardware manufacturer (HWM) the programmer (PR):

- HWM: *Do you mind if I scramble things to optimize performance?*
- PR: *No way! I don’t want to think about your hardware when I code.*
- HWM: *How about if I scramble things while making sure that your code appears to progress in order?*
- PR: *Will my code run faster?*
- HWM: *You betcha!*

- PR: *You got yourself a deal.*

If you remember, in the previous lectures we have discussed how the interleaving between instruction flows belonging to different processes poses a problem when it comes to mutual exclusion. One thing we relied on was that at least the instructions in a single thread process in a certain order. In OOO processors this is not true. In this case, software-only synchronization is not possible. Fortunately, the need to perform mutual exclusion and more in general synchronization among multiple, potentially parallel execution flows is well understood by hardware manufacturers.

## 20.1 Hardware Assisted Mutual Exclusion

Look back at the discussion on software-only mutual exclusion. There are always two operations: (i) read/**check** for the value of a flag; (ii) write/**set** the flag accordingly to mark entrance in the critical section. The root of all the problems is the fact that we may be interrupted by another execution flow between the check and set operations. The statement of the problem gives away the solution: we need have something that performs those two operations in an indivisible fashion. This way, there is no chance that anyone will be able to sneak in between check and set.

This property is formally called **atomicity**. If two (or more) simple operations are part of the same atomic block, then the hardware will process those operations in *one shot*. An example of a typical atomic instruction used for synchronization is test-and-set. In test-and-set, the address of a flag to be set is passed. Atomically, if the flag is 0, it is set to 1 and the operation succeeds; otherwise (if the flag was already 1) the operation fails. Listing 20.1 provides an high-level description of the basic logic for a test-and-set operation. Thanks to atomicity, lines 3-7 execute as an indivisible block.

```
1 bool test_and_set(bool & flag)
2 {
3   if (flag == 0) {
4     flag = 1;
5     return true;
6   } else {
7     return false;
8   }
9 }
```

Listing 20.1: High-level logic of test-and-set operation. The block is executed atomically by the hardware.

Now that we have this gift from the HWM, how can we use it to implement mutual exclusion ? Well, for example we could keep trying the test-and-set operation on a given flag until it succeeds. When that happens, we know we are the only one with the *right* to enter the critical section. Let us produce the corresponding pseudo-code corresponding to this approach in Listing 20.2.

```
/* Global, shared variable */
bool flag = 0;

while (true) /* Repeat forever */
{
  while ( test_and_set(flag) == false ); /* Breaks when test_and_set returns
    true  */
7   /* Perform work in critical section */
8   flag = 0; /* This will allow some other task's test_and_set to succeed */
9   /* Perform work outside critical section */
10 }
```

Listing 20.2: High-level logic for mutually-exclusive entrance in critical section using test-and-set

What kind of properties we can infer from the behavior of the code in Listing 20.2? First off, if our code enters the critical section, it is definitely alone. We also know that if flag is initialized correctly, test-and-set will succeed at some point. There is no way, however, to tell when the test-and-set will succeed. This depends on how many tasks contend to enter the critical section, and on how the OS is scheduling the tasks on the processor. Remember when we discussed that breaking ties based on job IDs was fair enough? Well now that decision may actually lead to starvation under certain circumstances[1].

In our example, we have used test-and-set to build mutual exclusion. It is important to remember that test-and-set is just an example of an instruction that executes check+set logic atomically. There exist other examples that are variations on the theme and that can be equivalently used to achieve mutual exclusion. For instance, Intel CPUs typically provide the compare-and-exchange (CMPXCHG) primitive, where a certain register/memory location (1st parameter) is set to a certain value (2nd parameter) only if its current value is the one given in a 3rd parameter. For our purposes, let us just stick to test-and-set.

## 20.2 Spinlocks

Let us look again at Listing 20.2. We can distinguish three main blocks. The first block corresponds to line 5; the second to line 6 (execution of the critical section); the third to line 7 (flag reset). The expression on line 5 allows us to **acquire** the right to enter the critical section. The expression on line 7 is a declaration that we have **released** the previously acquired right.

We can then create a pair of functions that do just that: acquisition and release of the right to enter a critical section. We can say that this *right* is a **lock**. We can then acquire and release the lock to the critical section, which is represented by the flag, defined in line 1 (and that is shared by all competing threads). Consider the transformation of Listing 20.2 provided in Listing 20.3.

```
bool flag;

void acquire(bool & flag)
{
  while ( test_and_set(flag) == false ); /* Breaks when test_and_set returns true  */
  barrier(); /* Not passed unless all previous instructions done */
}

void release(bool & flag)
{
  barrier(); /* Not passed unless all previous instructions done */
```

[1]Try to think about what happens if there are three tasks always ready on the CPU.

```
12   flag = 0; /* This will allow some other task's test_and_set to succeed */
13 }
14
15 while (true) /* Repeat forever */
16 {
17   acquire(flag);
18   /* Perform work in critical section */
19   release(flag);
20   /* Perform work outside critical section */
21 }
```

Listing 20.3: High-level logic for spinlock implementation

As can be seen, the main application code (lines 15-21) is much cleaner and invokes acquire/release functions that are provided by a library as needed. Notice that when the process invokes the `acquire` function (line 17), it may end up busy-waiting inside that function. In other words, it will **spin** on line 5 until it is able to acquire the lock. Because of this property, this type of synchronization primitive is called a **spinlock**. In a spinlock, tasks that attempt to acquire the lock keep the processor busy. This is not always the best way to perform mutual exclusion.

Note one more thing: we said that in OOO processors, things can be re-ordered. So what guarantees us that the lock release is not performed before the end of the critical section? Or even worse, before the acquire operation? Nobody. But HWM understand this and provide so-called **barrier instructions** (line 6, 11). A barrier placed somewhere is a directive to the processor to make sure it has completed all the operations **before** the barrier, before starting executing any instruction **after** the barrier. Those programmers who coded the synchronization primitives that we use (including spinlocks) have been kind enough to use barriers in their acquire/release code to enforce the required ordering.

## 20.3 Semaphores

We have discussed in the context of I/O operations that maybe keeping the processor busy is not always the best idea. This is especially true for those cases when the event may take a while to happen. In case of synchronization, the "event" in question is the moment in which the lock is finally acquired. For I/O operations, we had a structure where the task requesting an I/O operation becomes blocked (and releases the processor) on an I/O event. The task is later awaken when the I/O event finally occurs.

A similar thing can be done with critical sections. Specifically, we can have a scheme where the task: (i) attempt to acquire the lock; (ii) if it succeeds, it continue with execution; and (iii) if it fails, it becomes blocked and enter a lock-specific queue. This, in a nutshell, is the idea behind semaphores. Look at the typical structure of a semaphore, two are the fundamental components:

1. **count**: an integer counter that captures the current status of the semaphore;
2. **queue**: a queue of processes blocked on the resource/critical section for which the semaphore is being used.

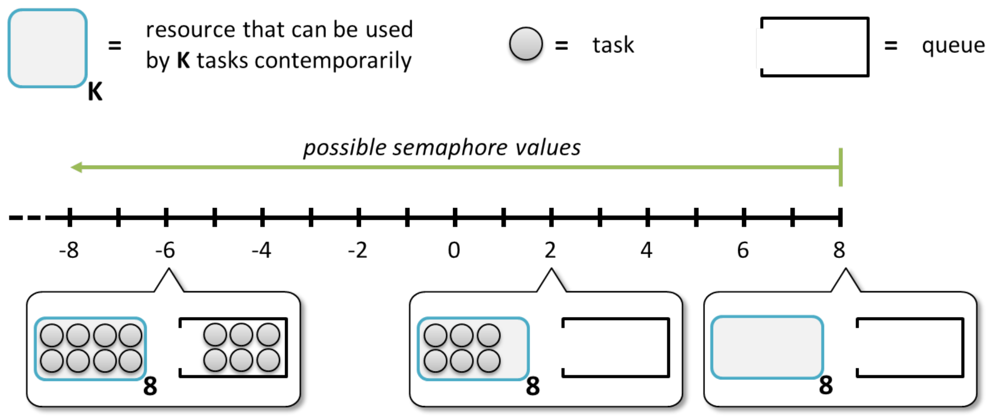
= resource that can be used by K tasks contemporarily
K
= task
= queue
possible semaphore values
-8
-6
-4
-2
0
2
4
6
8
8
8
8

Figure 20.1: Resource and semaphore status for a resource that supports up to 8 concurrent processes/tasks.

Let us start from the **count** component. Have you ever seen those parking lots with a counter on the entrance that report how many spots are available in the parking lot? When the value of the counter is non-negative, that is exactly the meaning of the counter. Basically, it tells us how many "seats" for resource being synchronized are available for newcomers. **If the value is positive**, as soon as a newcomer arrives, it is allowed access to the shared resource and the counter is decremented by one. **What happens if the counter reaches 0?** Now there are no parking spots left. So if a newcomer arrives, he/she is asked to wait until some spot frees up. **The counter goes to -1** tracking the fact that one person is waiting for a spot. We also do not want to forget that a certain person is waiting, so we make them enter a queue from where they will be picked as soon as spots become available.

So there you have it: a non-negative value (i.e. $\geq 0$) of the count variable captures how many more tasks are allowed for concurrent usage of the underlying resource. A negative value captures the number of tasks waiting to be allowed to use the resource. The **queue** contains references to tasks blocked on the resource, so that when other tasks releases the resource, it is possible to unblock one or more of the waiters. It goes without saying that if the semaphore count is 0 or positive, then the queue is empty. Conversely, if there are $N$ tasks blocked on the considered resource, then the value of count will be $-N$.

In order to visualize the state of a synchronized resource using a semaphore, consider Figure 20.1. The figure depicts the status of a resource and corresponding semaphore, for the case in which the resource supports up to 8 concurrent processes/tasks. The maximum value for the semaphore is 8. When the value is 8, no tasks are waiting and/or using the resource. When the value is 2, no tasks are waiting and $8-2=6$ processes are currently using the resource. When the value is $-6$, 6 tasks are waiting and 8 processes are currently using the resource.

To manipulate a semaphore, an interface that consists of two functions (similar to spinlock) is used. In this case, the functions are called **wait** and **signal**. The wait function is used by the process to see if there is an available seat on the resource (e.g. an available parking spot), and if there is, the process does not wait (despite the name of the function) and acquires the resource. If there are no seats, the task is inserted into the waiting queue. In both cases, the semaphore's count is decremented by one.

The signal function is invoked by a processor that was previously sitting on the resource and that intends to release its seat because it has finished using the resource (at least for the moment). When the signal function is invoked, the semaphore's count is increased by one. Now, if there are processes waiting to be awaken in the queue, a wake-up signal (hence the name) is sent to **one of the processes**, and the chosen one is allowed on the resource. A high-level description of the logic behind wait and signal functions is provided in Listing 20.4.

```
1  class semaphore {
2    int count;
3    queue queue;
4  };
5
6  void wait(semaphore & sem)
7  {
8    sem.count--; /* Decrement semaphore's counter */
9    if (sem.count < 0) { /* No seats available ? */
10     sem.queue.enqueue(current_process); /* Go into the waiting queue */
11     block(current_process); /* Block until unblocked by a signal */
12   }
13 }
14
15 void signal(semaphore & sem)
16 {
17   sem.count++; /* Increment semaphore's counter */
18   if (sem.count <= 0) {  /* Anybody waiting ? */
19     some_process = sem.queue.dequeue(); /* Dequeue some process */
20     unblock(some_process);  /* Allow that process on the resource */
21   }
22 }
```

Listing 20.4: High-level logic for semaphore implementation

Note that the wait and signal functions themselves manipulate two variables (count and queue) that are shared among all the processes competing for the resource. Wait and signal functions would not work if different processes can arbitrarily interleave in the execution of lines 8-12, and lines 17-21 in Listing 20.4. As such, we have to make sure that the execution of wait and signal functions is done as a monolithic block. But we have already learned in Section 20.2 how to enforce mutual exclusion of a critical section. Hence, it is enough to make the body of the signal and wait functions **two critical sections using spinlocks**.

### 20.3.1 Implementation of Semaphores

The operations described in Listing 20.4 need to be implemented typically as part of an OS library, just as system calls for I/O operations are made available for processes to use.

The semaphore as a data structure is itself shared amongst the various processes who may be using its `wait()` and `signal()` operations. As such, we must ensure that the concurrent execution of these processes will not in any way *mess up* the state of the semaphore. Thus, we must ensure that the `wait()` and `signal()` operations are executed in a mutually exclusive fashion. In other words, at any point in time, for any semaphore *S*, at most one process may be executing the `wait()` or `signal()` operations. How could we do that?

We can certainly use any one of the solutions we discussed before for implementing mutual exclusion for critical sections. However, of the techniques we discussed before, the only truly feasible way to achieve mutual exclusion for *N* processes on modern multi-processor and potentially OOO hardware is through test-and-set operations. Notice however, that such a technique does not get rid of the busy-waiting, which was a concern of ours in the first place. Luckily though, the amount of busy waiting is small, since the critical sections defined by the `wait()` and `signal()` are very short—typically ten or so instructions.

If we go for this solution, which is quite close to the actual implementation in real systems, we will add a `flag` field in the internal state of the semaphore to be used for test-and-set operations. We will then appropriately insert spinlock calls to acquire/release the internal lock of the semaphore when appropriate within the code of the `wait()` and `signal()` operations. The final implementation is provided in Listing 20.5, where the additional state/logic is highlighted in red.

```
class semaphore {
  int count;
  queue queue;
  bool flag;
};

void wait(semaphore & sem)
{
  acquire(sem.flag);
  sem.count--; /* Decrement semaphore's counter */
  if (sem.count < 0) { /* No seats available ? */
    sem.queue.enqueue(current_process); /* Go into the waiting queue */
    release(sem.flag);
    block(current_process); /* Block until unblocked by a signal */
  } else {
    release(sem.flag);
  }
}

void signal(semaphore & sem)
{
  acquire(sem.flag);
  sem.count++; /* Increment semaphore's counter */
  if (sem.count <= 0) {  /* Anybody waiting ? */
    some_process = sem.queue.dequeue(); /* Dequeue some process */
    release(sem.flag);
    unblock(some_process);  /* Allow that process on the resource */
  } else {
    release(sem.flag);
  }
}
```

Listing 20.5: High-level logic for semaphore implementation

### 20.3.2 Initialization of a Semaphore

Just as with any abstract data structure, and *especially* with semaphores, one must specify what the initialization of the data structure ought to be.

When a semaphore is created (to manage a resource, be that a critical section of code, or a device, etc.) the semaphore `queue` must be initialized to being empty—since at that stage there are no processes waiting to acquire the semaphore. Also, the semaphore `count` must be initialized to some integer value. The initialization of the semaphore `count` is precisely what specifies the nature of the synchronization implemented by that semaphore! The following are two distinct possibilities among many others.

- **Semaphore initialized to 1:** This indicates that the semaphore is used to control access to an initially available resource in such a way that at most one process could acquire that resource at any point in time. This is precisely the kind of synchronization amongst processes needed to coordinate access to a critical section! Indeed, the code in Listing 20.6 hows how one would solve the critical section problem using a semaphore called `mutex` and initialized to 1. A semaphore used to allow at most one process to acquire the resource it is managing is called a *binary semaphore*.

```
/* Global, shared variable */
semaphore mutex = 1;

Process Pi:
   repeat:
      /* remainder section */
      wait(mutex);
      /* critical section */
      signal(mutex);
      /* remainder section */
   forever
```

Listing 20.6: Using a semaphore mutex (initialized to 1) to implement mutual exclusion for a critical section.

- **Semaphore initialized to some positive integer $K$:** This indicates that the semaphore is used to control access to an initially available resource in such a way that at most $K$ processes could acquire that resource at any point in time. One can see that this is simply a generalization of the binary semaphore, which allows for a more general control of the multiprogramming level (MPL) allowed for a resource. Namely, by initializing a semaphore to $K$, we imply that the multiprogramming level for the resource controlled by that semaphore is upper-bounded by $K$. If you recall the considerations we made at the beginning of this course, it should become clear that a semaphore is an *implicit way* to limit the MPL of a given resource. It is implicit because it does not require a central arbiter to make sure that the MPL cap is respected—-as long as each process correctly uses the resource-specific semaphore.

### 20.3.3 Basic Uses of Semaphores

Semaphores are primitives which can be used in many different ways to provide a variety of synchronization capabilities. The following are a few of these.

- **Mutual Exclusion:** As we have discussed in the context of Listing 20.6, a binary semaphore (initialized to 1) could be used by a set of processes to ensure mutually exclusive execution of a critical section of code. This would allow us, for example, to control access to shared data structures to ensure that race conditions will not result in the $15+1+1 = 16$ problem we have described in Section 19.1.
- **Controlled MPL:** As we have discussed above, a counting semaphore initialized to $K$ could be used by a set of processes to ensure than no more than $K$ of these processes are using a resource concurrently. This could be used for admission control purposes. For instance, consider how you may be able to guarantee that a file sharing application does not allow for more than $K$ uploads from your machine concurrently. If each request for an upload results in a new process, you can enforce this condition by initializing a counting semaphore to $K$ and then by requiring that each upload process `wait()` on that semaphore before it is allowed to proceed; and of course to `signal()` that semaphore when it is done.
- **Enforcing Relative Ordering of Parallel Code Segments:** Consider two processes P1 and P2. Let S1 be a particular point (e.g., a specific instruction) in P1 and S2 be a point in P2. Assume that it is required that the execution of the concurrent processes P1 and P2 be required to satisfy the following property: *No instruction below S2 in P2 can ever execute before all instructions up to (and including) S1 in P1 have finished execution*. This is an example of two programs within which we may want to enforce a particular relative ordering of execution. For instance P1 may be producing results that are needed in P2 and thus we may not want to execute the segment of P2 (namely what follows point S2) until these results from P1 (namely those produced by instructions before S1) are available. One way to do this is through the use of a semaphore (call it `synch`). If we initialize such a semaphore to 0, then the precedence relationship that S1 must precede S2 can be enforced by preceding S2 with a `wait(synch)` and following S1 by a `signal(synch)` function call as shown in Listings 20.7 and 20.8.

```
Process P1:
   ...
   S1;
   signal(synch);
   ...
```

Listing 20.7: P1 Code

```
Process P2:
   ...
   wait(synch);
   S2;
   ...
```

Listing 20.8: P2 Code

- **Rendezvous Synchronization:** In the previous case, only one process waits for the other to reach a certain point in its execution. When this property is symmetric, we say that the processes implement a *rendezvous*.
  For this case, consider two instructions, S1 in process P1 and S2 in process P2. Assume that it is required that the execution of the concurrent processes P1 and P2 be required to satisfy the following two properties: (1) No instruction below S2 in P2 can ever execute before all instructions up to S1 in P1 have finished execution and (2) No instruction below S1 in P1 can ever execute before all instructions up to S2 in P2 have finished execution. Another way to saying this is that we require that S1 and S2 be defined as *rendezvous points* for P1

and P2—i.e., we want the execution of P1 and P2 to "honor an appointment" (from French for appointment *rendezvous*) at S1 and S2, respectively. One can see that this rendezvous requirement is nothing more than saying that we want to have *instructions before S1 to happen before those after S2* and *instructions before S2 to happen before those after S1*. Thus, we can use two semaphores (say) `synch1` and `synch2` to symmetrically enforce the *happens after* constraint as we have done above. This is shown in Listings 20.9 and 20.10.

```
Process P1:
   ...
   signal(synch1);
   wait(synch2);
   S1;
   ...
```

Listing 20.9: P1 Code

```
Process P2:
   ...
   signal(synch2)
   wait(synch1);
   S2;
   ...
```

Listing 20.10: P2 Code

The above 2-party rendezvous synchronization can be extended to $N$ parties. A straightforward implementation would use $N*(N-1)$ semaphores, whereby a process P$i$ will have $N-1$ semaphores $S_{i,j}$, one for each other process P$j$. Upon arrival to the rendezvous point, a process P$i$ would signal all $S_{i,j}$ semaphores and will wait on all $S_{j,j}$ semaphores. Clearly, all processes must reach their rendezvous point for any one of these processes to proceed beyond the rendezvous point. This solution is expensive, as it requires $N^2$ semaphores. One can show that $N$ processes could achieve the same rendezvous synchronization using only two semaphores.

## 20.4 Read-Copy Update - RCU

As we have seen so far, semaphores (an spinlocks) represent incredibly versatile structures to perform multi-party synchronization. In a computing world where the amount of parallelism is continuously increasing, however, they might fall short in terms of efficiency. That is because semaphore-based synchronization is based on blocking. In fact, the strategy to guarantee the consistency of a resource (or data structure) protected by a semaphore is to allow only one process a time to make changes to the shared data structure, while all the others are blocked.

More specifically, let us assume that we have a large number of processors accessing the same shared data structure, e.g. a linked list. Because the list is shared, different threads on multiple processors need to synchronize over the structure, otherwise the consistency of the list can be jeopardized. Imagine what would happen is some readers are accessing elements, while other threads are removing/adding elements in parallel. If we do nothing, we will end up with a crippled list with dangling pointers and unreachable elements, apart from a bunch of segmentation faults for threads that attempted at accessing list items that whose memory has been free'd.

Clearly, if many readers read something at the same time, that is not a problem. However, if any of the threads wants to update the structure (e.g. by deleting a new element), then we are in trouble. Thankfully, we learned how to mutually exclude processes from accessing the same shared resource/data structure. Suppose that we have $m$ processors, a number of readers and exactly one writer. How things progress in time if we use spinlocks (or semaphores) to perform mutual exclusion

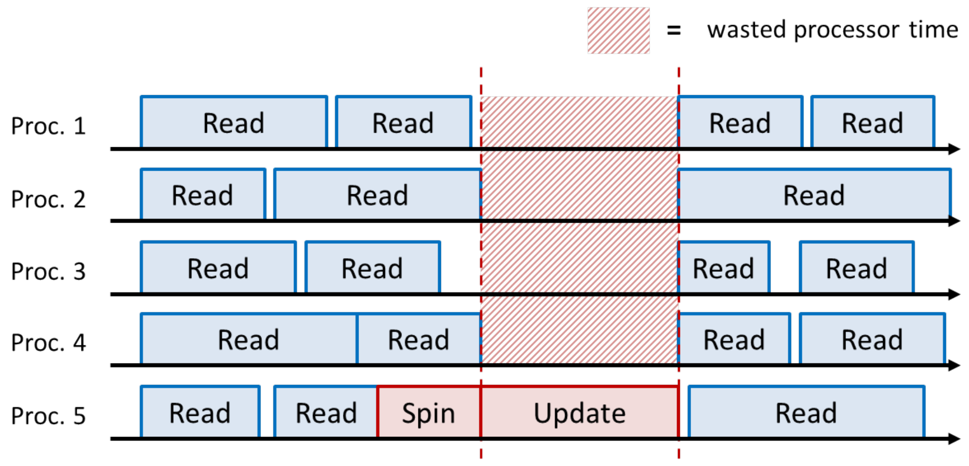

Figure 20.2: If traditional mutual-exclusion is used with multiple readers and few updaters, the waste of processor time can be significant.

of our shared linked list? Figure 20.2 provides a visualization of what happens. If only readers are active, the can progress in parallel. An updater, instead, first spins to wait for all the readers to complete the current operation. When the updater finally acquires the lock, all the readers are not allowed to access the shared data structure. This goes on until the updater completes. All the readers can now access the resource again. As highlighted in the figure, the activity of a single updater can stall all $m-1$ cores! This is an unacceptable performance loss for large multi-processor systems.

Can we do better? Well, if the problem that we want to solve is having multiple processors see a non-broken data structure, albeit not necessarily the same one at the same time, then we can use **Read-Copy Update** RCU locks. The principle based on which RCUs work is simple, although their implementation entails a number of nitty-gritty details which we will not discuss. Let us look at Figure 20.3. Initially, we only have a single version of the considered data structure. When the updater process on processor 5 arrives, it only needs to synchronize with other potential updaters, if necessary. Other than that, it can proceed right away.

The updater first performs a read-copy of the data structure. The copy can also be partial if the data structure is composed of separable element, like in the case of a linked list. The updater then changes its personal copy, producing a new version of the data structure. After the updater is done, two versions of the same data structure exist. The key here is: all the readers that begin a read **after** the update's completion will receive a reference to the new structure, not the old one. As such, as readers progressively complete, the number of processes referencing the old data structure can only decrease until reaching 0. Once this number reaches zero, it is safe to de-allocate from memory the old data structure (or again, only a part of it). The time between the end of the update, and the time instant at which it is finally safe to de-allocate the old data structure is called **grace period**. It is highlighted in green in Figure 20.3.

In order to understand how things work for RCUs, let us use an example of a linked list. Suppose that we initially have a linked list of integers, depicted in Figure 20.4(a). In parallel with the readers, the updater creates a new list element that is initially visible only locally, as depicted in Figure 20.4(b),

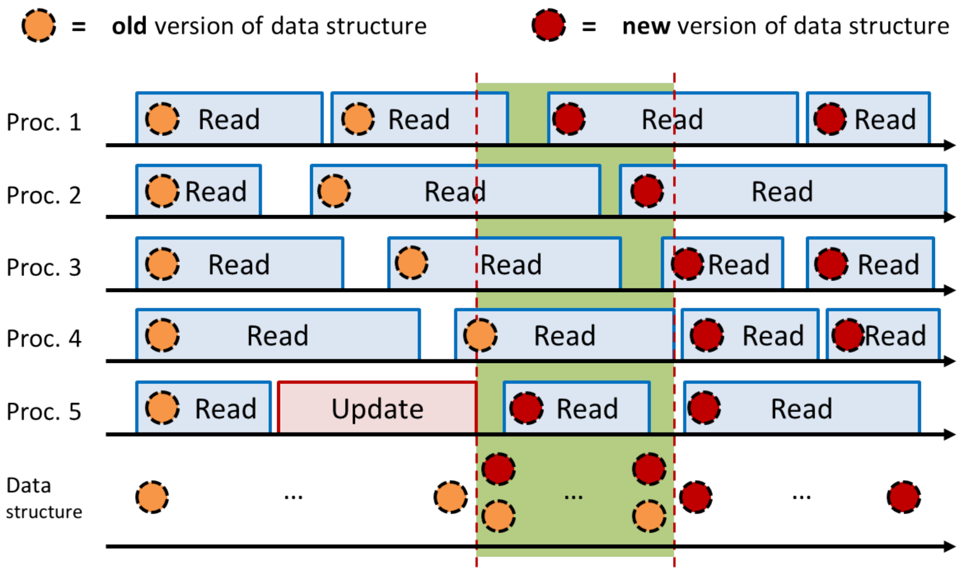
= old version of data structure
= new version of data structure
Proc. 1
Proc. 2
Proc. 3
Proc. 4
Proc. 5
Read
Update
Data structure
...

Figure 20.3: Key idea behind RCUs. Different versions of the same data structure can coexist for a limited amount of time. Different readers may see slightly different versions of the same structure, but each of them is consistent in its way.

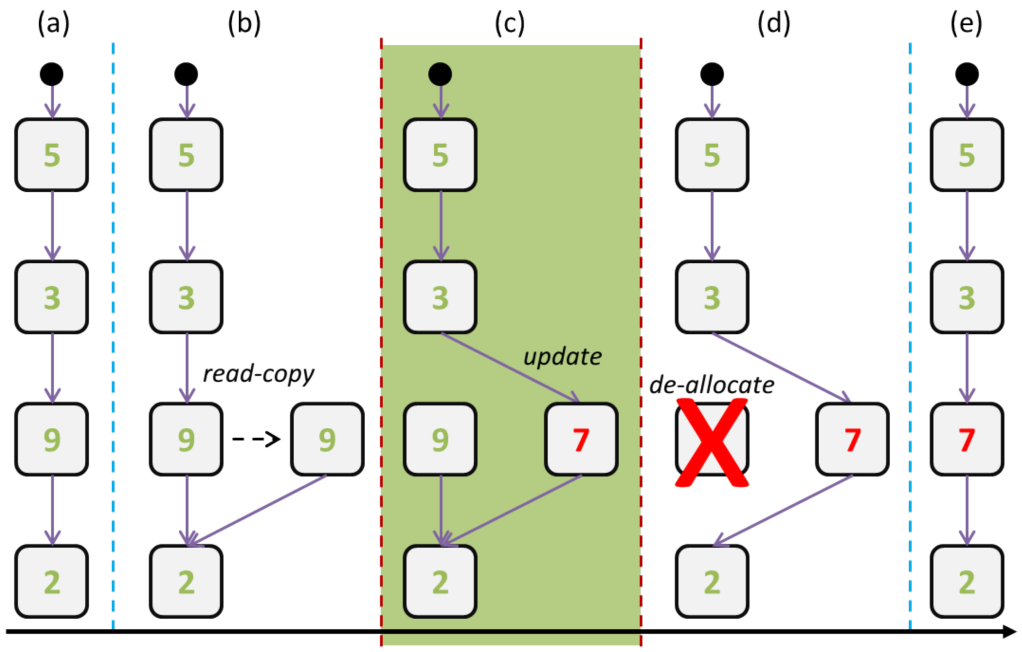

Figure 20.4: Sequence to update a RCU-synchronized linked list: (a) initial list; (b) partial read-copy operation; (c) update, lasting through the grace period; (d) partial de-allocation of old structure; (e) final list after grace period.

and read-copies the data from the original data structure. The data consists of the integer key, as well as the pointer to the next element. Note that this read-copy operation is only partial, as only one node of the structure is touched by the update. The same concept can be generalized to many elements or to the whole linked list.

Next, the fields of the newly created element are updated as needed Figure 20.4(c). In this case, the key is changed from 9 to 7. This also marks the beginning of the grace period. At this stage, the old list element is not de-allocated because other readers may be accessing it. This effectively means that two versions of the same data structure exits. One that is visible only to readers that started working on the structure before the beginning of the grace period and has the elements (5, 3, 9, 2); and one that is visible to all the readers that acquired the RCU lock after, with the elements (5, 3, 7, 2).

Once the last reader that is accessing the old version of the data structure releases the RCU lock, then the grace period is over. At this point, it is safe to de-allocate the memory for the old list element – the one with key 9, see Figure 20.4(d). This leaves us with only the updated version of the data structure, as shown in Figure 20.4(e).

### RCU Performance and Use

In a world increasingly dominated by multi-core systems, RCUs provide a big boost in performance. This is obvious if we compare Figure 20.2 and Figure 20.3: RCUs allow readers to continue while the update is still being carried out. Clearly, this optimization cannot be played always. In fact, RCUs are not always suited (and have good performance) in all the scenarios:

1. Read-mostly data structures, in a system where there is no need for an absolutely identical view of shared data for all the readers – here RCUs are the best;
2. Read-mostly data structures where absolutely identical view of shared data for all the readers is required – RCUs are okay;
3. Read-Write with need for absolutely identical view of shared data – RCUs may be okay;
4. Write-mostly data structures where absolutely identical view of shared data for all the readers is required – RCUs are a bad idea.

After all being said, and given the additional complexity of implementing RCUs, is it worth the effort? Figure 20.5 helps us answering this question. In the figure, the red line represents the overhead arising from the use of **read-write spinlocks** (rwlocks) which, unlike simple spinlocks, allow readers to access the shared resource in parallel – see Figure 20.2. On single-core systems, rwlocks have an overhead of about 90 ns, soaring to about 6000 ns for a 16 cores processor. On the other hand, RCUs deliver consistent performance from 1 to 16 cores, with overheads of about 50 ns.

For this reason, RCUs have been increasingly used in the Linux kernel to implement efficient data sharing across multiple processors. Figure 20.6 reports a plot of the number of uses of the RCU primitives within the Linux kernel over the years that go from 2002 until 2016.

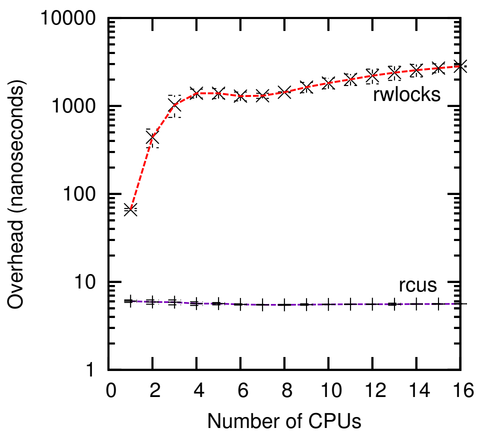

Figure 20.5: RCU vs. read-write spinlocks.

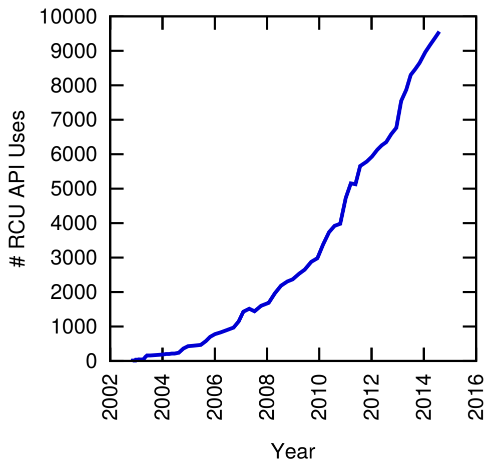

Figure 20.6: RCU usage in the Linux kernel.