Readers-writer locks are used to protect a data structure that is used in read-only operations and read-write operations. The threads that execute read-only operations are called readers. The threads that execute read-write operations are called writers. Multiple readers can execute concurrently; however, a writer thread needs exclusive access to the critical section. To understand this, consider the following code:

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    node->next->prev = node->prev;
    node->prev->next = node->next;
    free (node);
}
int search (struct node *list, int key) {
    int ret = -1;
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
           break;
        }
        list = list->next;
    }
    return ret;
}
```

In this example, a list is used to store key-value pairs. A search operation walks the list and returns the value corresponding to a given key. The delete operation deletes a node from the list. Because, the list can be modified and read by concurrent threads, we must use locking to ensure the correctness of the operations. One option is to use a spin lock. The spin lock will allow only one thread to enter the critical section. The spin lock doesn't take advantage of the fact that the search operation never modifies the list. It is perfectly fine to allow multiple threads to execute search operation when no thread is concurrently deleting a list node.

This is the main idea behind a readers-writer lock. A writer lock ensures that no reader or other writers can run concurrently when a read-write operation is being executed. In the code below, acquisition at write_acquire will ensure that no other thread can enter in the search and delete critical sections.

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    write_acquire (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    write_release (&lock);
    free (node);
}
int search (struct node *list, int key) {
    int ret = -1;
    read_acquire (&lock);
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
           break;
        }
        list = list->next;
    }
    read_release (&lock);
    return ret;
}
```

In the implementation of the readers-writer lock, we have used atomic_add and atomic_sub. atomic_add and atomic_sub atomically add and subtract a value to/from an input memory location respectively, and returns the updated value of the memory location.

The pseudo-code presented on the slide has two weakness.

1. Both readers and writers perform atomic operations on the same lock variable that cause cache line bouncing.
2. A writer may starve infinitely if some readers are active all the time.

The second problem can be solved by disallowing new readers when a writer is active. For the first problem, an alternative locking strategy called big-reader lock is used.

In the big-reader lock, each CPU has its own private lock. The locks for different CPUs do not share the same cache line. A reader acquires/releases the lock corresponding to the current CPU when it enters/exits the critical section. Preemption is disabled in the read critical section to disallow threads from getting scheduled in the read critical section. Why?

Because the per-CPU locks don't share the same cache line, the problem of cache line bouncing among readers is resolved.

A writer acquires locks corresponding to all CPUs. The acquisition of per-CPU locks of all cores prevents a reader or other writer from entering the critical section when a writer is active.

Few shortcomings of a big-reader lock.

1. The memory required for the lock is proportional to the number of cores
2. The lock implementation needs details about the cache line size of the hosting platform
3. lock acquisitions by writers are slow (need to acquire locks for all CPUs)

Another lock primitive which is widely used in Linux kernel is called read-copy update (RCU).

RCU locks have lock-free read-critical section. Preemption is not allowed in the read critical section. RCU locks work for workloads where a writer is compatible with the lock-free readers. For example, consider the following case:

```
void delete (struct node *node) {
   spin_lock (&lock);
   node->next->prev = node->prev;
   node->prev->next = node->next;
   spin_unlock (&lock);
   free (node);
}
int search (struct node *list, int key) {
   int ret = -1;
   preempt_disable ();
   while (list != NULL) {
      if (list->key == key) {
         ret = list->val;
         break;
      }
      list = list->next;
   }
   preempt_enable ();
   return ret;
}
```

In the above example, delete operation can concurrently execute with the search operation because the updates that matter for search is the update of the next field of a linked list node. Notice, that in delete update of next (node->prev->next = node->next), is atomic because stores are atomic. If a delete is executing concurrently, the search routine will either see the new list or the old list depending on whether next is updated or not.

If the search routine has seen the old node that is being deleted concurrently, as long as we do not free the memory corresponding to the node (at free (node)), the search will work correctly. In the RCU scheme, a free operation is delayed until the updater is 100% sure that the other threads do not have a reference to the object that is going to be deleted. If the updates on which a reader depends is not atomic in writer then the RCU cannot be used.

For example, RCU cannot be used in the case below.

```
struct node {
    int key, val;
    struct node *next, *prev;
}
void delete (struct node *node) {
    spin_lock (&lock);
    node->next->prev = node->prev;
    node->prev->next = node->next;
    spin_unlock (&lock);
    free (node);
}
int search (struct node *list, int key) {
    int ret = -1;
    while (list != NULL) {
        if (list->key == key) {
            ret = list->val;
            break;
        }
        assert (!list->prev || list->prev->next == list);
        list = list->next;
    }
    return ret;
}
```

Notice that the search routine now depends on both next and prev fields. Because the updates of both next and prev filed together are not atomic, the reader (search) may see an inconsistent linked list.

In the RCU scheme, a writer calls rcu_free instead of free. rcu_free ensures that all the readers have exited at least once from their critical section after the rcu_free is called. The above check makes certain that the other threads do not have an active reference to the object which is going to be deleted.

To achieve this, rcu_free calls wait_for_rcu that enforce a quiescent state on every core. A quiescent state is a point in the code that is guaranteed to outside of read-critical section. One example of a quiescent state is the schedule routine. Because preemption is not allowed in the read-critical section, invocation of schedule suggests that the particular core is not executing a read-critical section.

wait_for_rcu schedules the current thread on all cores to enforce a safe point for a pending free. In the kernel code, scheduling is not allowed at all program points, e.g., interrupt handlers, spin_locks, etc. In these program points, instead of calling wait_for_rcu, rcu_free queues the pointers that needs to be freed. A different routine is called to free all the pointers in the queue after calling wait_for_rcu. The above routine can be called asynchronously at program points where scheduling is allowed.

Can we use RCU locks for data structure likes a tree? Notice that a tree search may depend on multiple updates that cannot be done atomically.

We can use RCU with trees by making all the updates visible in just one atomic operation. An update in the tree can be done in following steps to make them compatible with RCU.

1. make a copy of the tree
2. make all the changes in the newly created copy
3. update the root of the old tree with the root of the new tree in one atomic operation
4. free the old tree after wait_for_rcu

The above solution is acceptable when the writers are not frequent.

Because RCU lock and big-reader lock require disabling of preemption, they cannot be used in user-mode.

Let us discuss how a processor can communicate with another processor. For example, during a page table update, a CPU may need to send requests to other CPUs to invalidate their TLBs. Notice, that the other CPUs might be doing something else. To facilitate communication among CPUs, x86 allows inter-processor interrupt (IPI). Using IPI a processor can send an interrupt to another processor. A CPU may send IPIs to other CPUs when a TLB invalidation is needed on other CPUs. In the IPI handlers, the other CPUs can invalidate their TLBs and resume their normal execution.