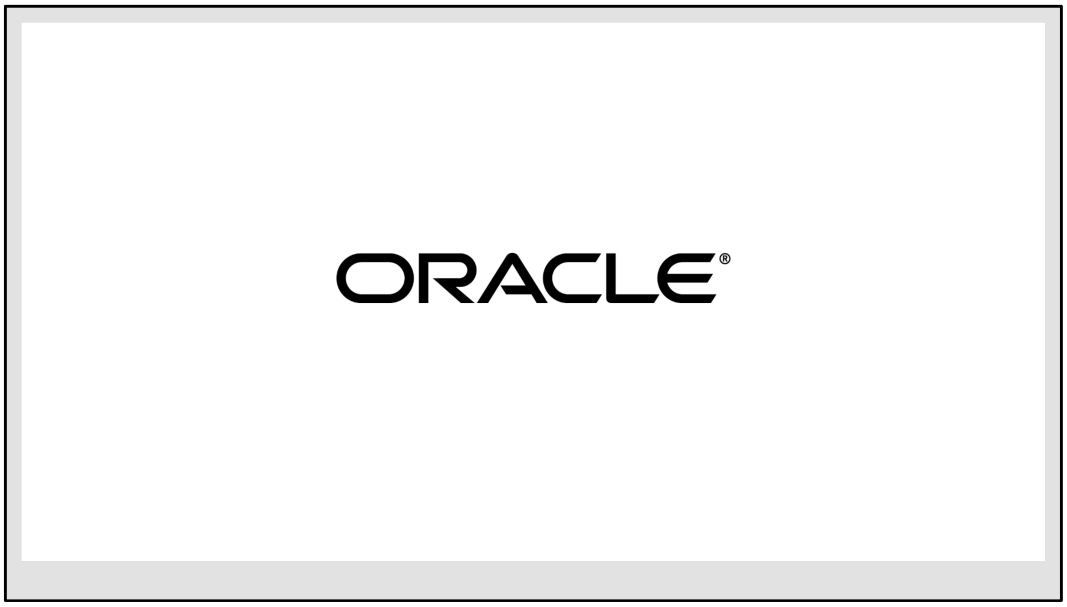
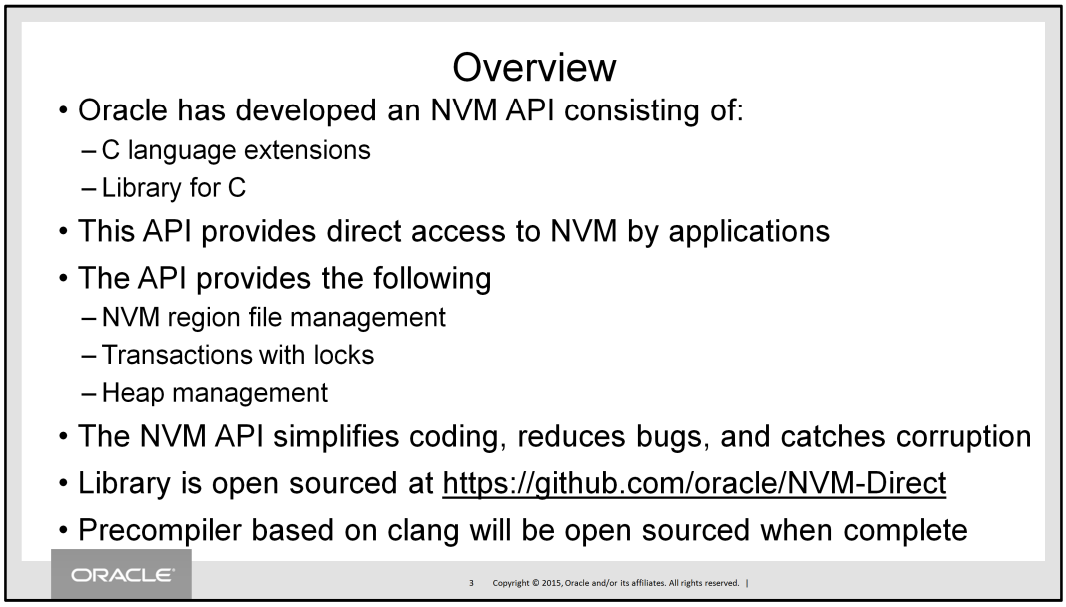

ORACLE®

NVM Direct
Open Source Non-Volatile Memory API

Bill Bridge
Software Architect
Oracle

The advent of persistent memory in enterprise computing introduces new problems related to managing consistent application state. To solve these problems Oracle developed a C library and a set of C extensions for applications utilizing NVM. This library and the extensions implement an application interface (API) that is applicable to a wide range of applications. The talk describes the API and its features.



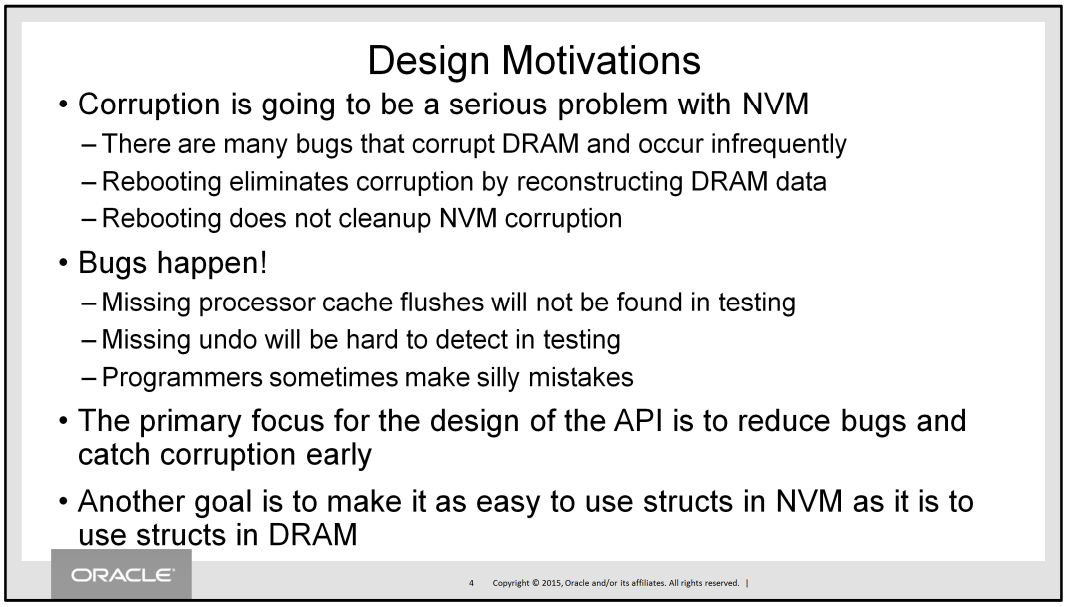
The library provides entry points called by the extensions as well as functions that applications call directly. It is possible to use the library without the C extensions, but the code will not be as robust.

The API provides facilities for an application to map NVM into its virtual address space and access it with loads and stores. This requires an OS file system that creates files in NVM and maps the NVM directly in the application address space when mmap is called.

- The API manages region files that are formatted for use by the library.
- The API provides transactions to atomically update complex data structures in NVM
- The API provides NVM mutexes that can be used to coordinate access to NVM data by a multi-threaded application
- The API supports multiple NVM heaps for allocating application defined structs in NVM

The language extensions simplify writing NVM code, automate some coding to reduce bugs, and add runtime checks to catch corruption early.

Oracle has open sourced the library code under a very permissive license. The code and associated documentation can be found at https://github.com/oracle/NVM-Direct. The code conditionally compiles to either use the C extensions or not. We will also open source a precompiler based on Clang. It will take the output from the standard preprocessor and convert the C extensions into standard C. It would be more efficient if the extensions were implemented directly in the compiler rather than as a precompiler.



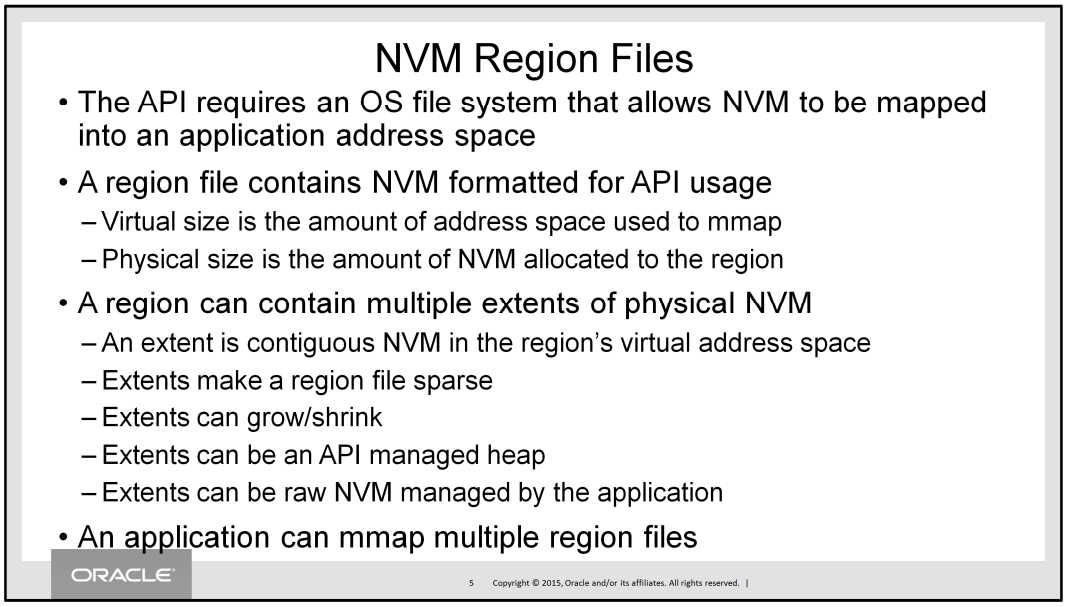
Reboot is the most powerful recovery tool that we have today. A reboot throws away all DRAM contents, including any corruption, and rebuilds DRAM from disk. There are many subtle bugs in existing production code that can cause DRAM corruption, but are rarely encountered. An OS panic or application crash will recover from these problems when they do occur. However NVM survives reboot and is just as susceptible to corruption as DRAM. This is going to be a big problem.

Applications that directly store data into NVM must flush the data out of the processor caches to ensure it is actually stored in NVM. Failure to code the flush will result in a bug that is very rarely encountered since data tends to eventually be flushed out of the processor cache. On rare occasions such a bug will result in a corrupt NVM data structure at a customer production site. This is so rare that it is unlikely to be caught in testing.

Failure to generate undo can also lead to subtle bugs that will be difficult to find in testing.

Prevention and early detection of corrupt NVM data structures is the primary goal of the API. Thus there are several features in this API that are designed to catch bugs at compile time, automate some operations, avoid some problems, and provide redundant data for runtime checking.

Another goal is to make NVM code easy to read and similar to code for data structures kept in DRAM. It needs to be clear which code is modifying NVM, both to the compiler and to a person browsing the code. I believe that C extensions are a requirement for writing robust NVM applications.



The API depends on the OS providing a file system where the files are allocated from NVM, and mmap maps the NVM of the file directly into the application address space. This provides access control and allocation of NVM to different applications on the same server. For Linux the ext4 file system is being extended to provide this capability.

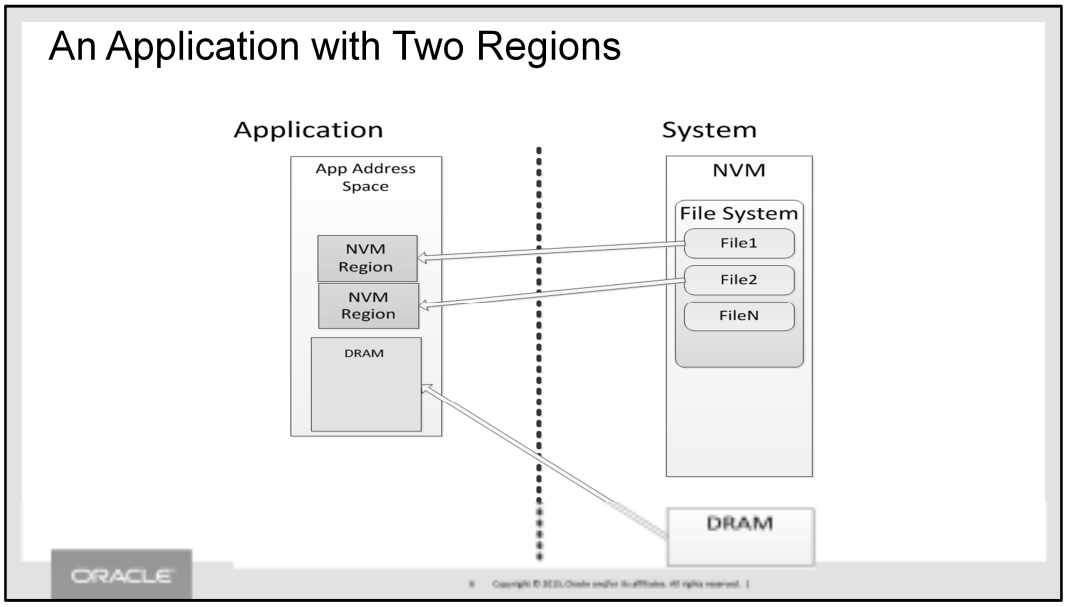
The API formats an NVM file with metadata that is recognized by the library. Only a properly formatted NVM file can be accessed by the library. A file formatted for use with the API is called a region file. When mapped into an application its virtual address space is called a region. It may be mapped at different virtual addresses by different processes, but only one multi-threaded process can map a file at the same time.

A region file has two sizes. The virtual size is the amount of application virtual address space that is consumed when it is mapped into an application. The virtual size is fixed when the region file is created. The physical size is the amount of NVM actually allocated to the file. This is generally many times less than the virtual address space. The physical size can change while the application is running. This allows the application to allocate more NVM from the file system if needed, or return space to the file system when it is no longer needed.

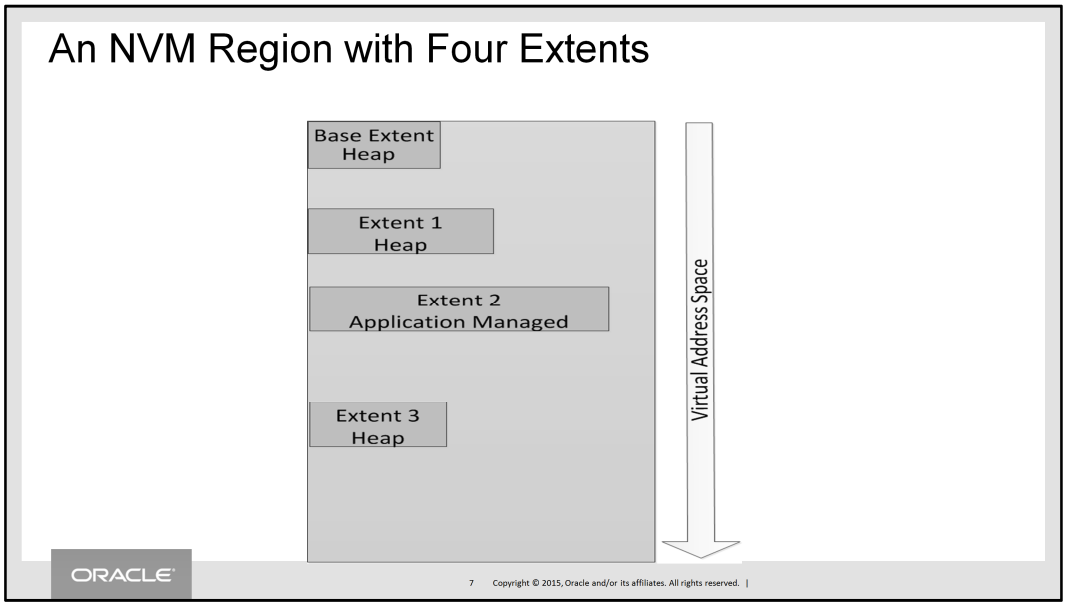
A region can contain multiple extents of physical NVM. Each extent occupies a contiguous chunk of the processe's virtual address space. Typically the extents have large amounts of empty address space between them. This allows the extents to grow as needed. Extents can be added, deleted, grown, or shrunk while the application is running. Access to a region virtual address that is not within an extent results in an error just like accessing an invalid DRAM address. This is so that a bug that does a wild store does not end up allocating unused NVM.

Currently the API supports two types of extents. A heap managed extent can be used to allocate and free persistent structs. An application managed extent is raw NVM that the application can use as it sees fit.

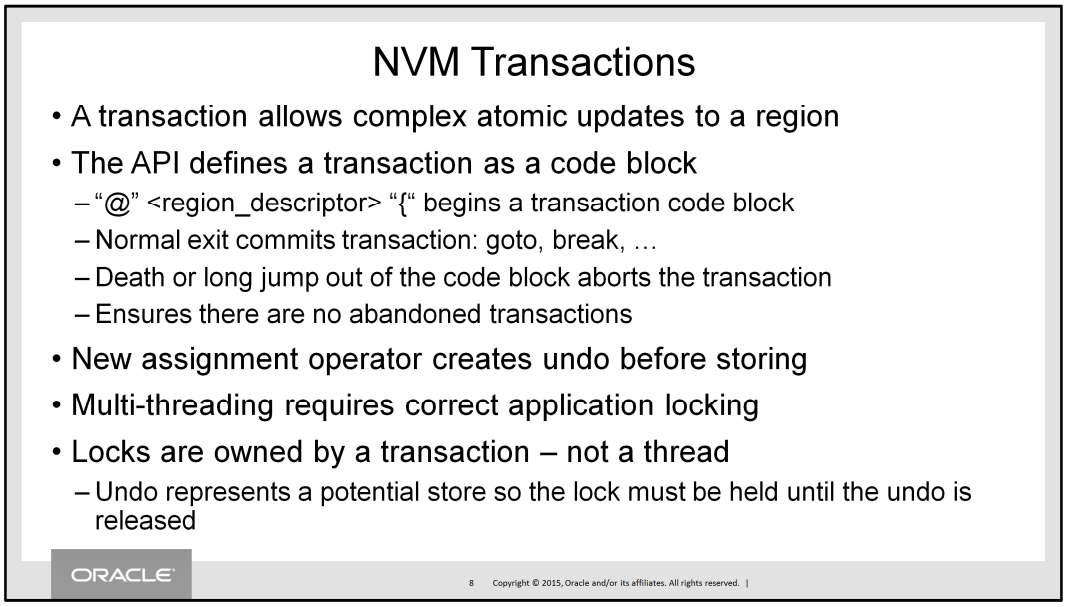
Multiple regions can be mapped into the same application address space. However, the API ensures that there are never any references between regions. This is necessary because there is no mechanism to ensure both regions are always mapped in at the same time, or that neither region is a restored backup. In general multiple extents in the same region can provide the functionality that multiple regions would provide.



This diagram shows an application that has two regions mapped into its address space from an OS file system that contains many NVM files, some of which might not be region files. The OS maps DRAM into the application as well as NVM.



This diagram shows that one of the application regions has four extents of NVM each of a different size. There is always a base extent that starts at byte zero of the region and contains some API metadata that is required to manage the region. The base heap is always heap managed. This application has two other heap managed extents and one application managed extent. Most of the region address space is unused.



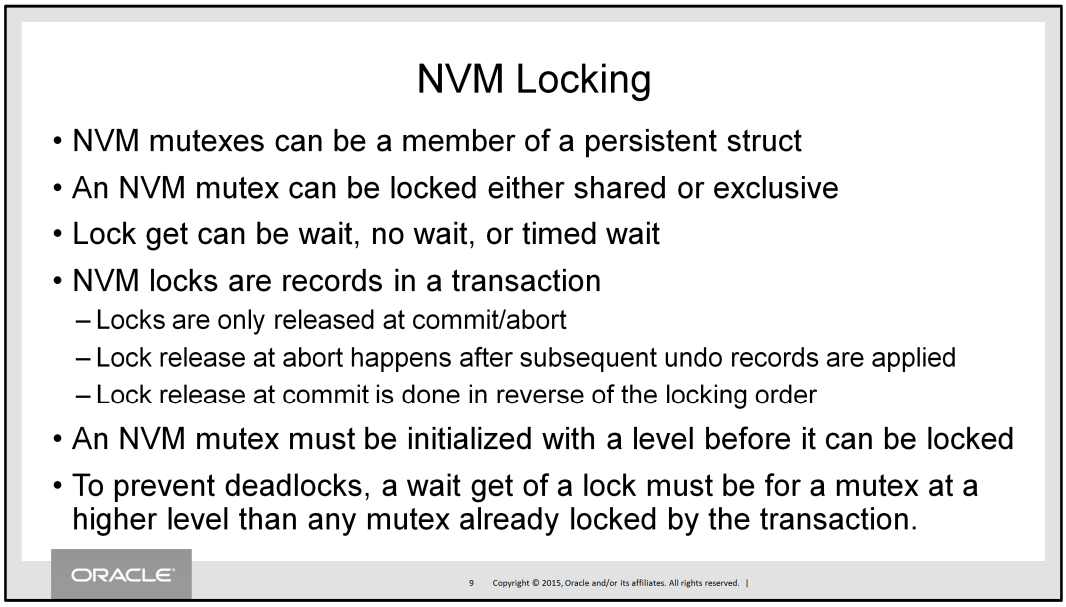
Unlike volatile memory, NVM retains its contents through a power failure or processor reset. This can happen at any time. If it happens in the middle of updating a data structure such as a doubly linked list, then a mechanism is needed to restore the linked list to a consistent state. This problem can also arise when an application thread signals an error or dies while manipulating persistent data structures.

A transaction mechanism is needed to atomically modify an arbitrary set of locations in NVM. Before a transactional store to NVM, the old value of the location will be saved in an NVM transaction data structure. This is the undo to apply if the transaction aborts or partially rolls back. When all stores for the transaction are complete, the undo is deleted thus committing the transaction. The application may choose to abort a transaction applying all the undo generated so far. An application may also create a savepoint and later apply all undo generated after the savepoint was created. This is a partial rollback of the transaction.

The API defines a C extension to label a code block as a transaction for a specific region that is in the application address space. The transaction is committed if the code block completes execution to the end or is exited via a specific statement in the code block such as goto, break, return, . . . If the code block is exited by process death or a long jump, then the transaction is aborted. The current transaction can be explicitly committed or aborted either from the code block or by a function called while the code block is active. This C extension ensures that a transaction cannot be started and then unintentionally abandoned without a commit or abort.

The API also defines a new NVM assignment operators that automates the creation of undo. Some stores to NVM should not generate undo so there are other NVM assignment operators that do not generate undo. The existing assignment operators produce compile errors if used for an address in NVM. Thus the developer must consciously decide if a store to NVM needs to be transactional or not.

The API supports multi-threaded applications that use NVM. It provides mutexes that are similar to pthread mutexes except that they are in NVM and are locked by transactions rather than threads of execution. Locks held by a transaction are tracked by undo records in the transaction. Undo generated after a lock is acquired represents a potential store to NVM so the lock protecting the data needs to be held until the undo is applied by rollback or discarded by commit.



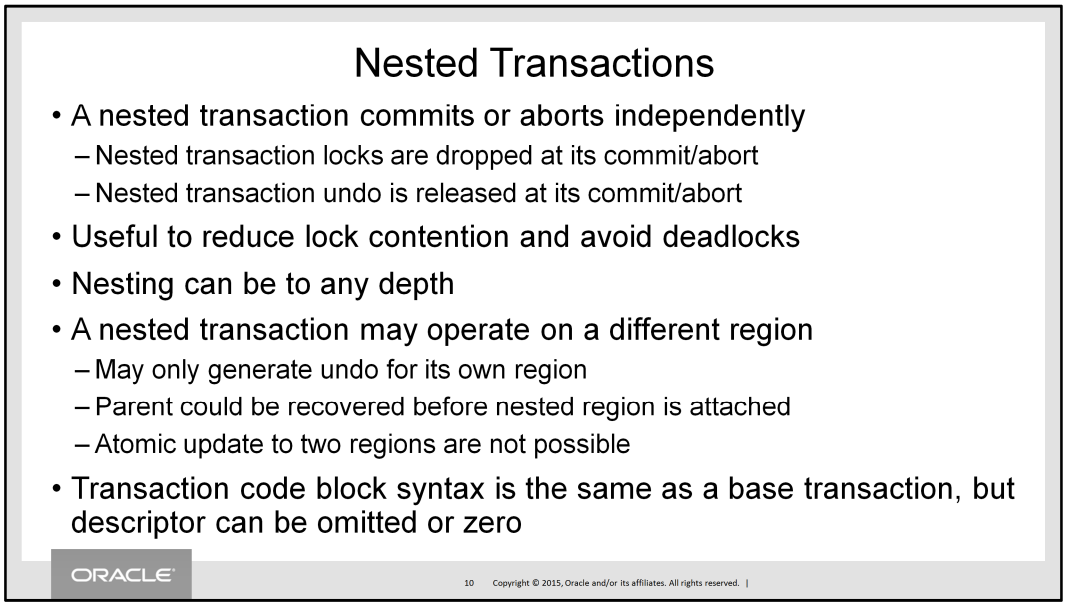
The NVM library introduces the concept of NMV mutexes for coordinating access to NVM data. NVM mutexes are declared as members of a persistent struct. NVM locks are similar to pthread mutex locks except that they are owned by a transaction rather than a thread of execution. They are released when the owning transaction commits, or if it rolls back all of the undo generated after the lock was acquired. Rollback can be the result of an abort or rollback to savepoint. Note that there is no means to explicitly release a lock without applying the undo or committing the transaction.

Locks can be either shared or exclusive. Once a lock is acquired exclusive, attempts to lock the same mutex by the same transaction are immediately granted. This is a no-op since locks are always released in the reverse order of acquisition.

A timeout in micro seconds can optionally be specified when acquiring a lock. The timeout can be zero for a no-wait get.

Since NVM locks are in NVM and associated with a transaction rather than a thread of execution, they survive process death or unmap of a region. When the region is reattached recovery will release all NVM locks as part of recovering all transactions. The locks need to be held during recovery because recovery may need to acquire locks in some cases.

To aid in deadlock avoidance, each NVM mutex has a level associated with it. A lock cannot be acquired with waiting if a lock of the same or higher level is already owned by the transaction. There is a lock initialization call that must be made to set the level.



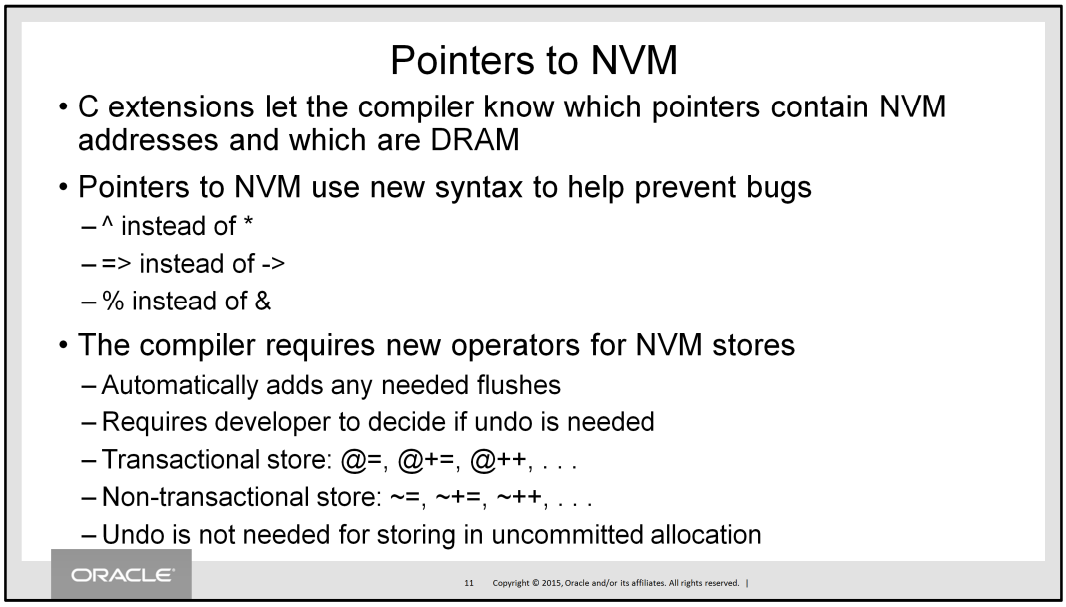
Nested transactions suspend operation of the current transaction so that all transactional operations happen in the nested transaction. The nested transaction can commit or abort independently from its parent transaction. When it commits, its changes are persistent even if the parent transaction aborts. The parent of a nested transaction can itself be a nested transaction.

Locks acquired by a nested transaction are released when the nested transaction commits/aborts. This makes nested transactions useful for avoiding deadlocks and for reducing the time a lock is held so that there is less contention. For example allocation from a heap is done in a nested transaction. An on abort operation is created in the parent transaction before the nested transaction is started. If the allocation commits, but the parent transaction aborts, then the on abort operation undoes the committed allocation.

On commit operations created during a nested transaction are executed when the nested transaction commits. The parent transaction does not become the current transaction until all the on commit operations complete and the nested transaction code block is exited.

A nested transaction may operate on a different region than its parent. An off region nested transaction can only acquire locks and transactionally store in its own region. It is not possible to atomically change two regions. Recovery happens at region attach time thus the parent and child might be recovered when the other region is not attached.

The syntax for a nested transaction is the same as a base transaction except that the descriptor may be omitted or be zero to indicate that the nested transaction is in the same region as the parent. A nested transaction in the same region as the parent is more efficient since the same transaction slot can be used.



The C extensions let the compiler, and anyone reading the code, know immediately which variables are pointers to NVM and which are pointers to DRAM. This is important because stores to NVM need to be treated very differently from stores to DRAM. The compiler inserts the necessary flush calls to ensure the store actually makes it out to NVM on the next persist barrier. The compiler can add runtime checks to ensure a self-relative pointer actually refers to an address within the same region, catching some corruption before it happens. When dereferencing a pointer to a struct in NVM the compiler can add a check to validate the struct starts with the correct signature. Assigning a DRAM address to an NVM pointer will generate a compile time error.

A pointer that contains an NVM address is declared with a "^" instead of "*". Similarly indirection through an NVM pointer uses a "^" instead of "*". Accessing a member of a struct in NVM uses "=>" rather than "->". Taking the address of an lvalue in NVM uses a "%" rather than "&". The compiler ensures the correct operators are used and NVM pointers really do point to NVM. This syntax also ensures developers know when they are dealing with NVM data rather than DRAM data.

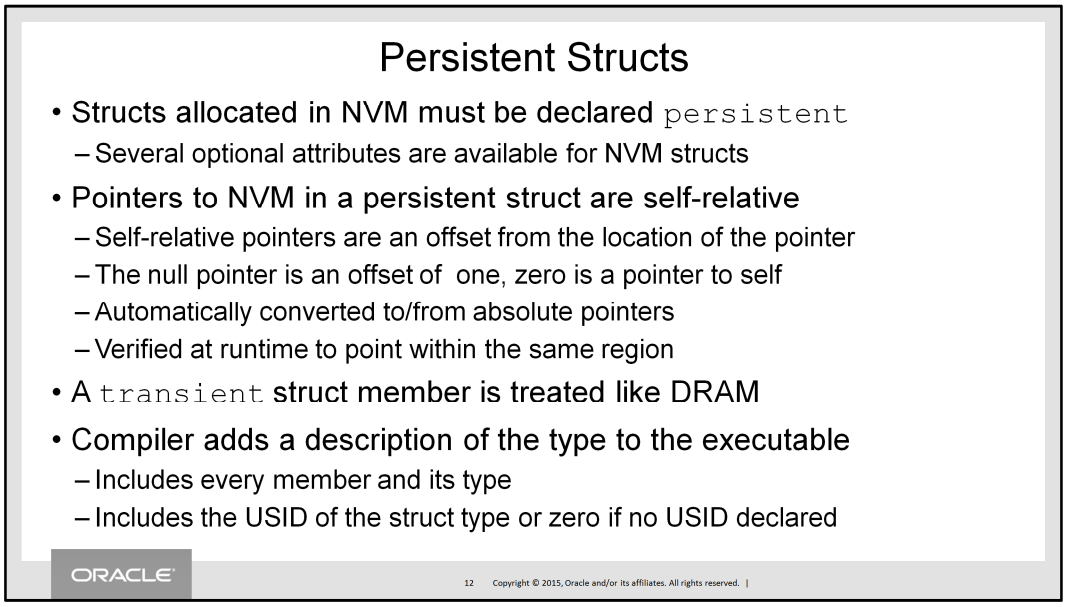
Storing into NVM also requires new operators. A store into NVM can be transactional or non-transactional. A transactional store must happen within a transaction and creates undo to restore the old value if the transaction aborts. A non-transactional store does not create undo, even if done within a transaction. The most common use of a non-transactional store is to initialize a newly allocated struct before the allocation is committed. Thus the choice depends on the logic of the code and must be done by the developer. Not allowing the standard store operators ensures the developer made this choice.

A transactional store uses the normal assignment, increment, or decrement operator with "@" in front of it. Thus the transactional operators are:

```
@= @+= @-= @*= @/= @%= @<<= @>>= @&= @|= @^= @++ @--
```

A non-transactional store uses the normal assignment, increment, or decrement operator with "~" in front of it. Thus the non-transactional operators are:

```
~= ~+= ~-= ~*= ~/= ~%= ~<<= ~>>= ~&= ~|= ~^= ~++ ~--
```



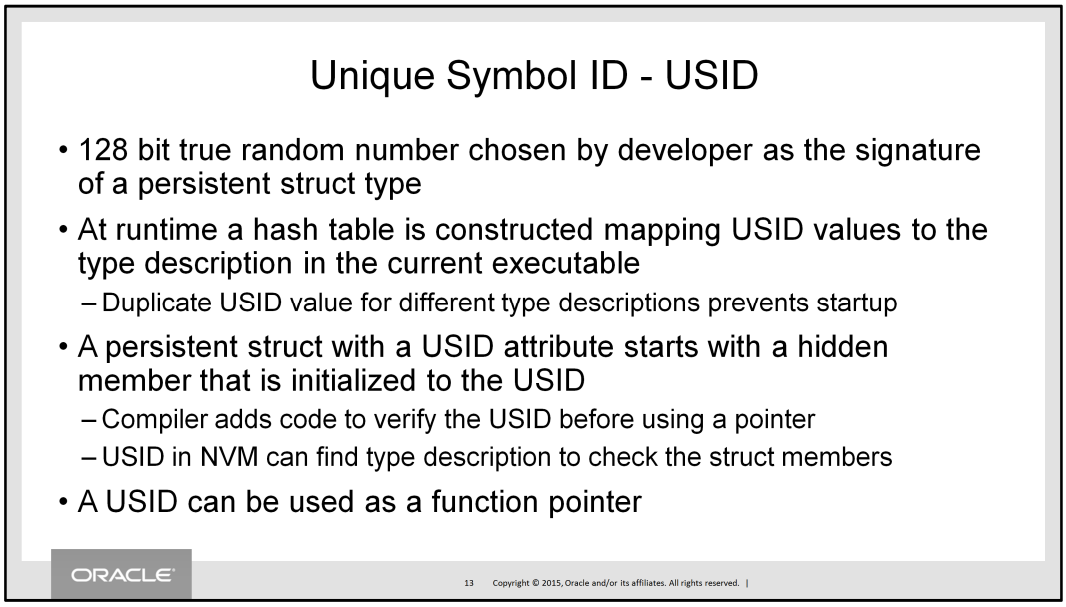
The new keyword “persistent” is used before the keyword “struct” to declare a struct as only being used in NVM. The “persistent” keyword must also be used in any incomplete declarations of the struct type. Allocation from an NVM heap is only allowed for persistent structs. Persistent structs can have a few additional attributes associated with them that help with debugging, runtime corruption checking, and software upgrades. These are described on later slides.

Pointers to NVM that reside in NVM need to be self-relative because the same region file may be mapped at different virtual addresses. A persistent struct member that is declared with a “^” is self-relative. Rather than containing a virtual address it contains the offset from the location of the pointer to the intended virtual address. A null self-relative pointer is represented as an offset of one. An offset of zero is a pointer to itself. Pointers to self are sometimes used to represent empty linked lists.

The compiler knows when a pointer is self-relative and automatically converts between self-relative and absolute pointers as needed. The developer does not need to be aware of self-relative pointers except when looking at NVM memory dumps. The compiler adds runtime checks to ensure every self-relative pointer is a pointer into the same region as the pointer itself.

Occasionally it is more efficient to keep some values in NVM that are only meaningful in the process that currently has the region mapped in. A struct member can be declared “transient” so that it will be treated like DRAM by the compiler, even though it is a member of a persistent struct. A pointer to DRAM can only be declared in a persistent struct if it is declared transient. Storing to a transient does not use an NVM store operator. It is the application’s responsibility to reinitialize all transient members before use after a new reattach. The region maintains an attach count that can be used to detect when a transient member is stale.

The compiler automatically generates a complete description of every struct that can be allocated in NVM. The description is const data in the executable. It includes the type and size of every member of the struct. It provides all the information necessary to initialize the struct to be empty with its correct signature. The builtin function shapeof(<type>) returns a pointer to the struct description.



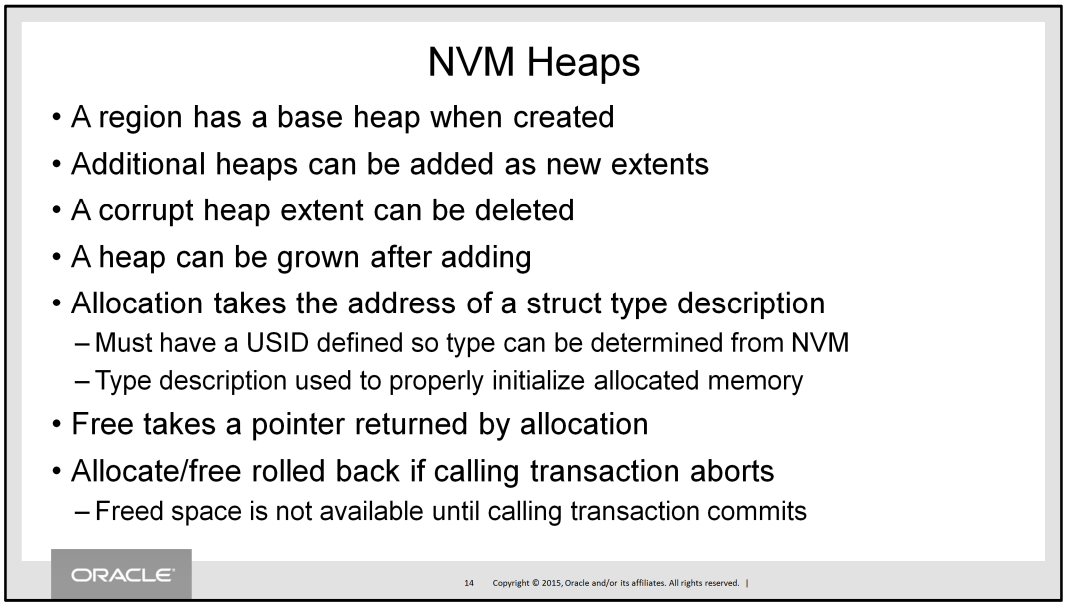
A Unique Symbol ID (USID) is a 128 bit random number chosen by the developer and added as an attribute to the persistent struct definition. It is similar to a GUID but does not conform to that standard. Random numbers that are the same regardless of endianess, are not allowed so that a region with the wrong endianess can not be used. Random numbers that are likely to appear as application values are not allowed. This makes it likely that if the value is found in memory, it was stored as a USID not for some other reason. This can be helpful in analyzing corruption.

Before an application maps in any region files, all the USID values it uses are registered in a hash table. The table entry gives the address of the corresponding type description. If a developer accidentally uses the same USID for two different types, then construction of the hash table will end the process at startup.

Not all persistent structs have a USID defined for them. Those that do begin with a hidden member that should always contain the USID defined for its type. This is a signature that allows an instance of the struct to be recognized in NVM. The compiler inserts runtime code to verify a pointer points to the expected USID before using the pointer. This will usually detect a stale pointer, a garbage pointer, or a struct that was unintentionally overwritten. This can prevent corruption or detect corruption before it spreads to other structs.

Structs allocated in an NVM heap must have a USID defined. This means that the type descriptions can be used to parse an NVM heap into all the persistent structs it contains. This allows detection of all bad pointers and most damaged structs in NVM.

A USID can actually be used to represent any global symbol in an executable. Thus they can be used as function pointers stored in NVM as well as type signatures.



When a region file is created it has one extent that contains an NVM heap. This initial heap is used for allocation of metadata used by the library. It is usually used for application data as well.

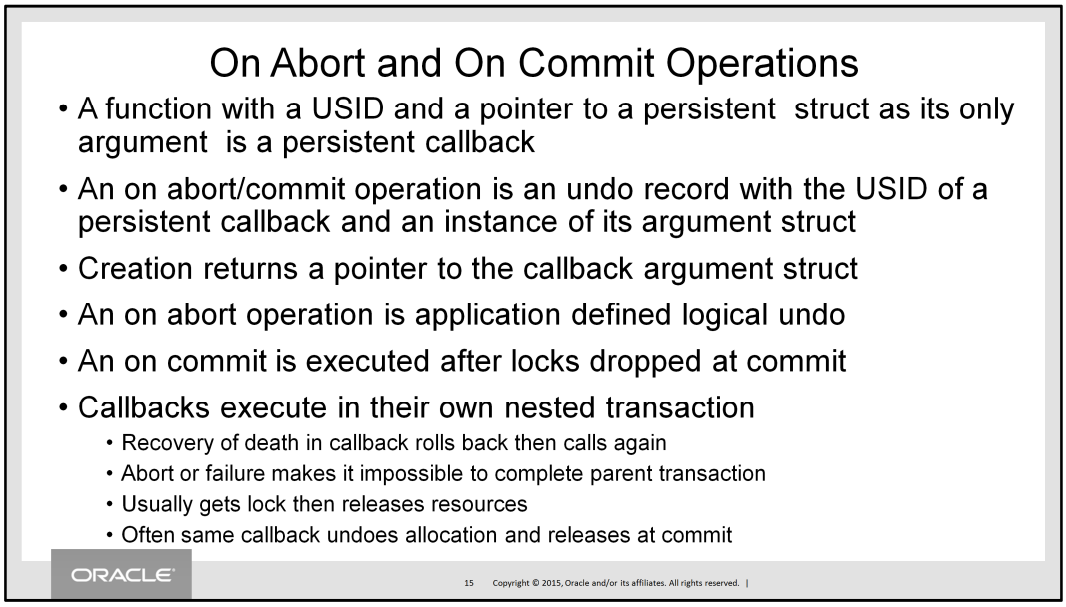
An application may create additional extents that are formatted as NVM heaps. This is convenient for grouping data that would be deleted together. It is not necessary to free each allocation individually if deleting the whole extent. Thus a corrupt heap can be deleted without looking at its contents. This can also be useful for eliminating corruption if it is limited to one extent.

As long as there is unused virtual address space at the end of a heap managed extent, the heap can be grown to allow more allocations. This is why extents should be created with large gaps between them.

Allocation from an NVM heap takes a pointer to the type description of the persistent struct to allocate. This allows the allocated NVM to be formatted with the correct USID signature, zero scalar values, and null self-relative pointers (i.e. offset one). Thus the application never gets any uninitialized data that could cause inconsistent behavior.

Similar to malloc/free, freeing space back to an NVM heap takes a pointer that was returned by allocation.

Allocation and freeing must be done in an NVM transaction that also stores or clears NVM pointers to ensure there are no leaked allocations or stale pointers . If the transaction does not commit then the allocation is undone or the free does not happen. Thus freed space is not actually returned to the heap until the transaction commits. Consequently the space in not available for allocation until the freeing transaction commits.



An application can create an on abort operation to be executed if the active transaction rolls back through the current point in the undo. This creates an undo record with a USID pointer to an application function and a persistent struct to pass to the function. If the undo is applied, the function is called in a new nested transaction. The nested transaction is committed when the function returns. This can be used to implement logical undo to reverse the effects of a nested transaction that committed. On abort operations are also useful for undoing OS operations, such as deleting a file that was created during a transaction if the transaction aborts.

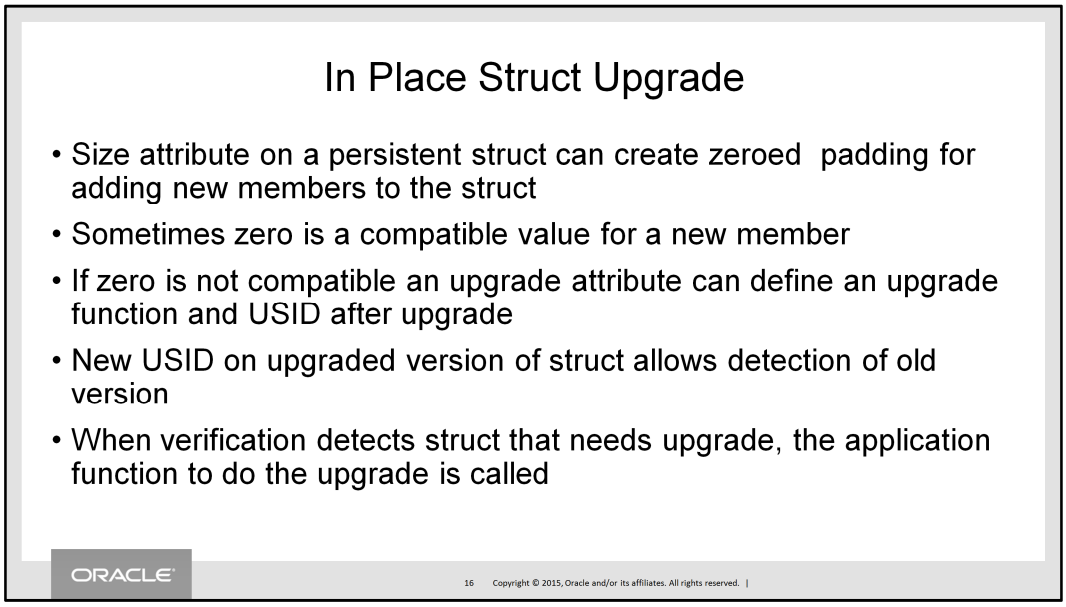
An application can create an on commit operation using a mechanism that is the same as creating an on abort operation except that the undo record does nothing if rolled back. All the on commit operations in a transaction are executed after the transaction has committed and all its locks released. While undo is applied in the reverse of the order it is created in, on commit operations are executed in the same order as they are created. Each on commit function is called in its own nested transaction that is committed when it returns.

On commit operations are useful for delaying the release of resources until the current transaction is guaranteed to commit. This can avoid generating large amounts of undo or acquiring locks that could cause deadlocks. On commit operations are also useful for delaying OS operations, such as deleting a file only if the transaction commits.

Creating an on abort/commit operation returns a pointer to the struct in the undo record so that arguments to the function can be modified later. It is common to have a nested transaction do a transactional store into the on abort operation so that rollback does nothing unless the nested transaction commits.

Note that the callback must not signal any errors, exit the process, or abort the nested transaction. This could result in an NVM region that could never be attached since every time recovery is attempted it would fail. Thus the only errors must be the result of serious NVM corruption.

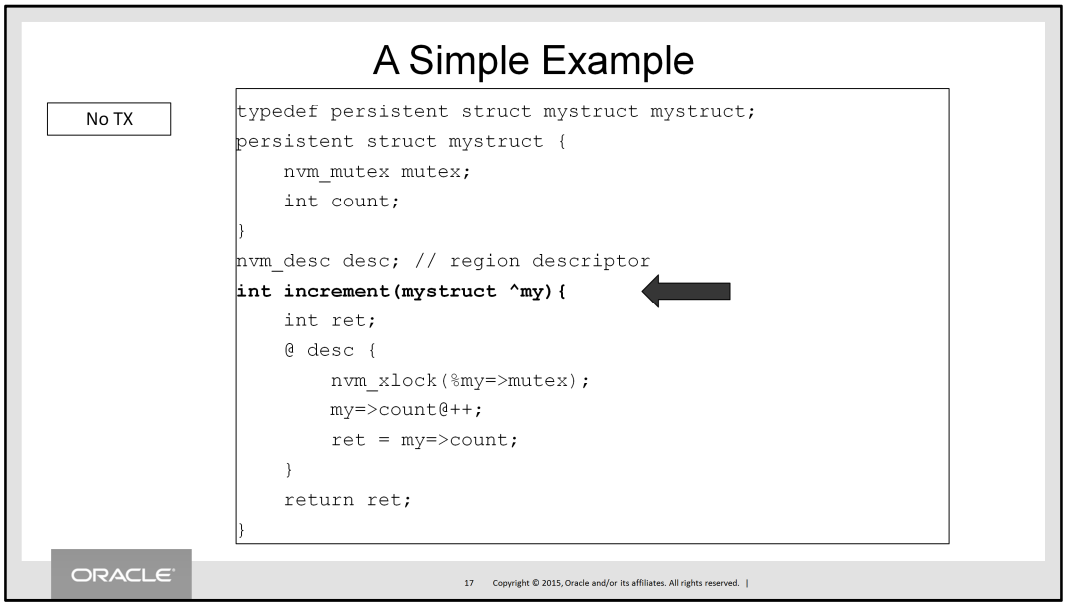
Note that the function could be called at recovery time so it must not attempt to access any volatile memory data structures created by the process that created the operation.



It is common for software upgrades to have new features that require changes in data structures. If the data structure is in NVM it is easiest to upgrade if there is unused space in the existing data structure that can be used for the new data. The API supports this type of struct upgrade.

A size attribute can be specified for a persistent struct. If the size attribute is too small to hold all the defined members then there is a compile error. If the size required to hold the defined members is less than the size attribute then padding is added to make the struct be the requested size. There may also be padding added to correctly align struct members. All the padding is cleared to zero when a struct is allocated. A new member can be added to an existing struct declaration in place of padding. If a value of zero in the new field is appropriate for backward compatibility, then that is all that is needed for upgrading.

Sometimes a new member needs to be non-zero, or members may need to be rearranged. This can be handled by specifying an upgrade attribute. The attribute specifies a new USID for the upgraded version of the struct and a function to do the upgrade. The upgraded struct will keep the name of the old struct, but have a new USID and maybe some new members. The old struct gets a different struct name but with all the same members. The old struct declaration has the upgrade attribute. Verifying the USID of an unconverted struct will see the wrong USID, but the existing USID will indicate a type that can be upgraded to the expected type. The verify routine will notice this and call the upgrade function to do an in place upgrade.



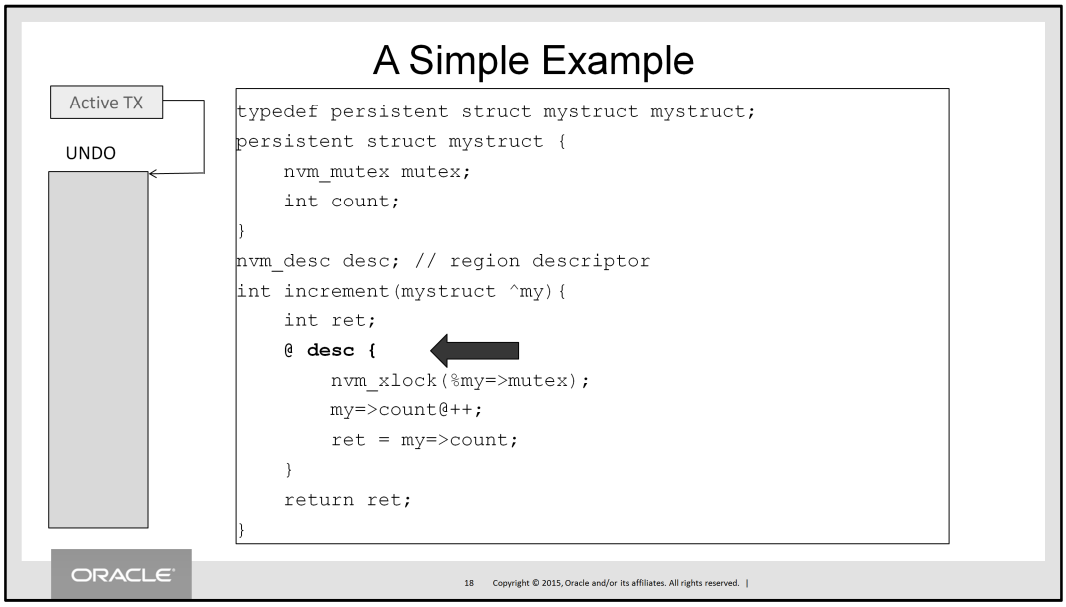
This is a simple example of some code that uses the C extensions to increment a counter in a persistent struct and return the new value. An NVM mutex is used to ensure only one thread at a time increments the counter.

The persistent struct mystruct has a mutex and the counter. Note that the typedef requires the persistent keyword.

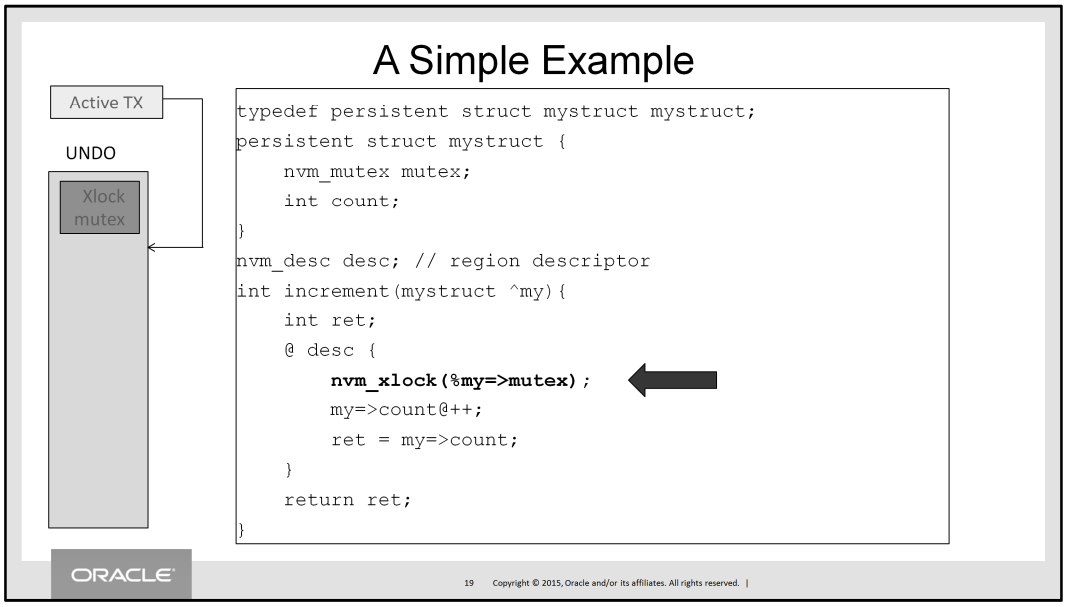
The desc global is presumed to be set to the region descriptor when the region file was attached.

The increment function takes a pointer to a mystruct which is a pointer to NVM.

There is no current transaction when increment is entered.

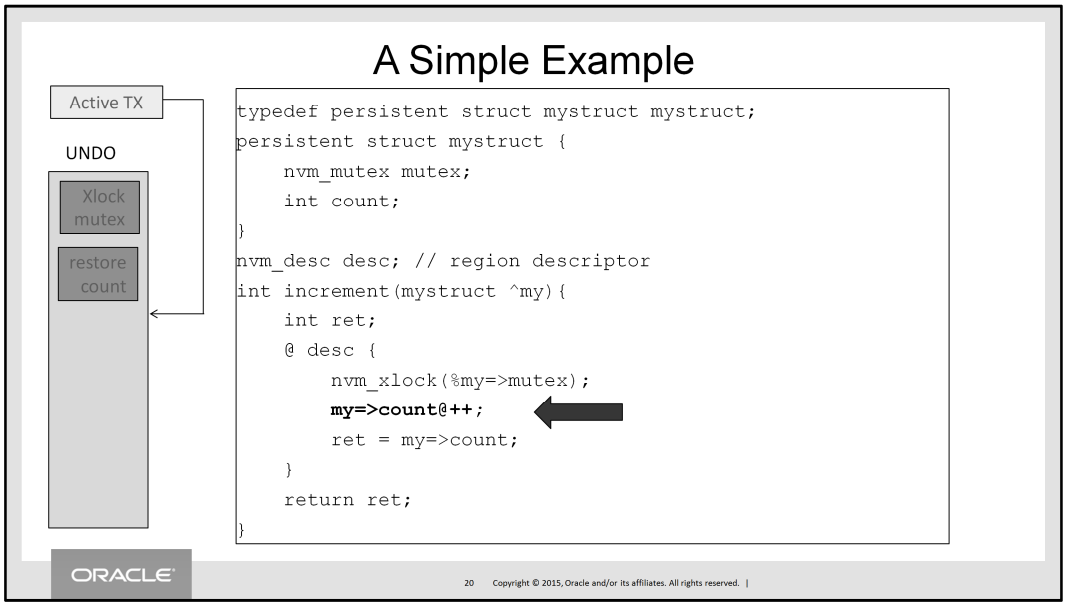


"@ desc " { begins a transaction in the region containing the mystruct. Initially the transaction is empty.



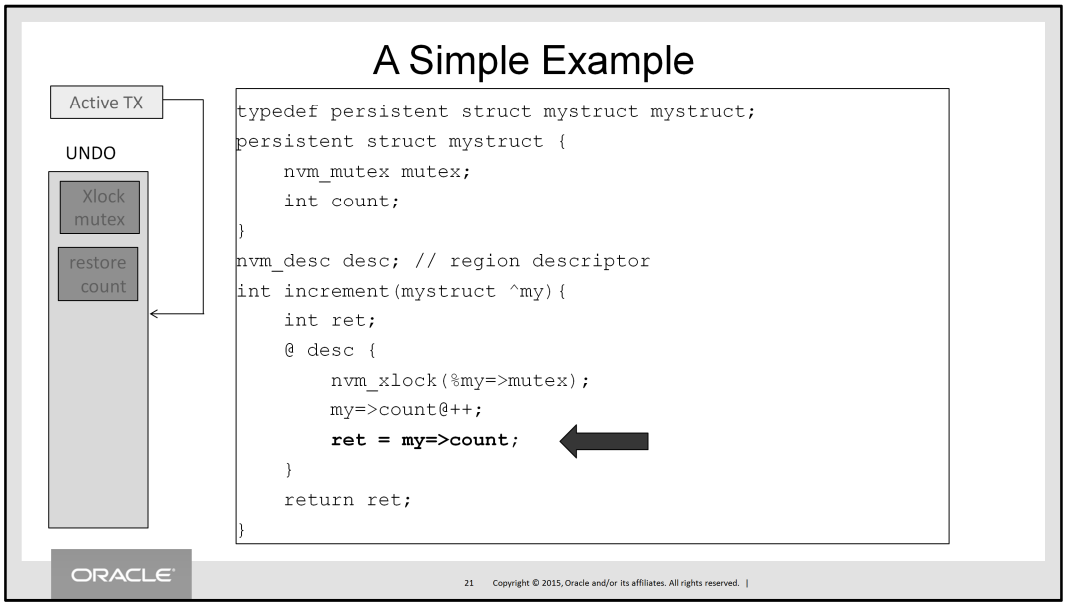
The function nvm_xlock() acquires an exclusive lock on a mutex, waiting if necessary. It takes the address of the mutex as its only argument. The NVM address of operator “%” is used to get the address rather than “&”. The NVM structure dereference operator “=>” is used rather than “->”.

After the lock is acquired the transaction has an undo record that represents the lock.

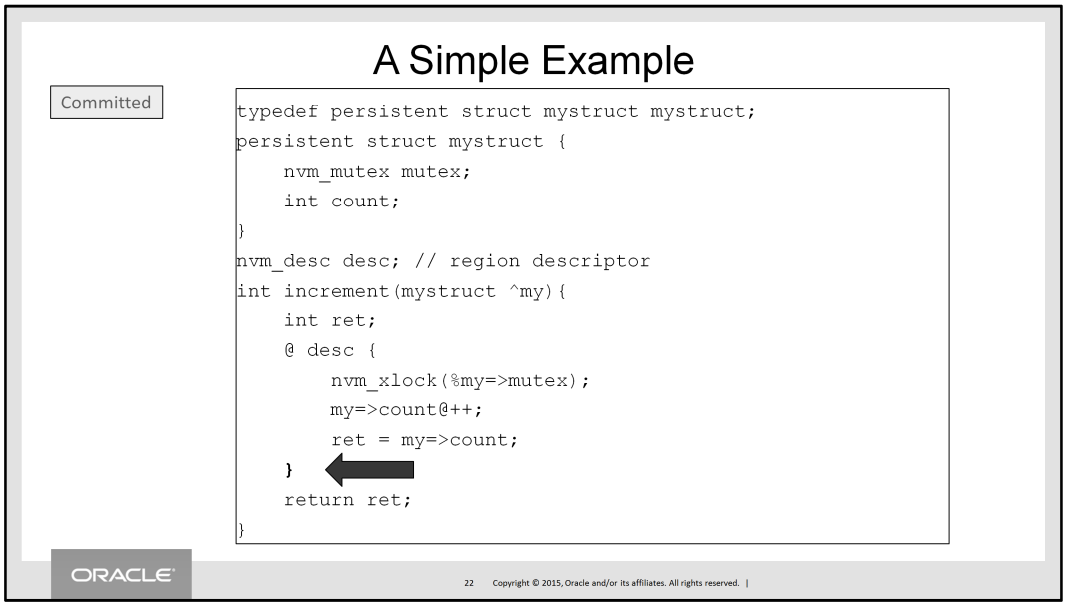


Before the count field is incremented an undo record is created to restore the old value of count if this transaction does not commit due to process death.

The NVM increment operator “@++” is used to transactionally update count creating the undo record. This also causes the value in the processor cache to be forced to NVM before the transaction commits.

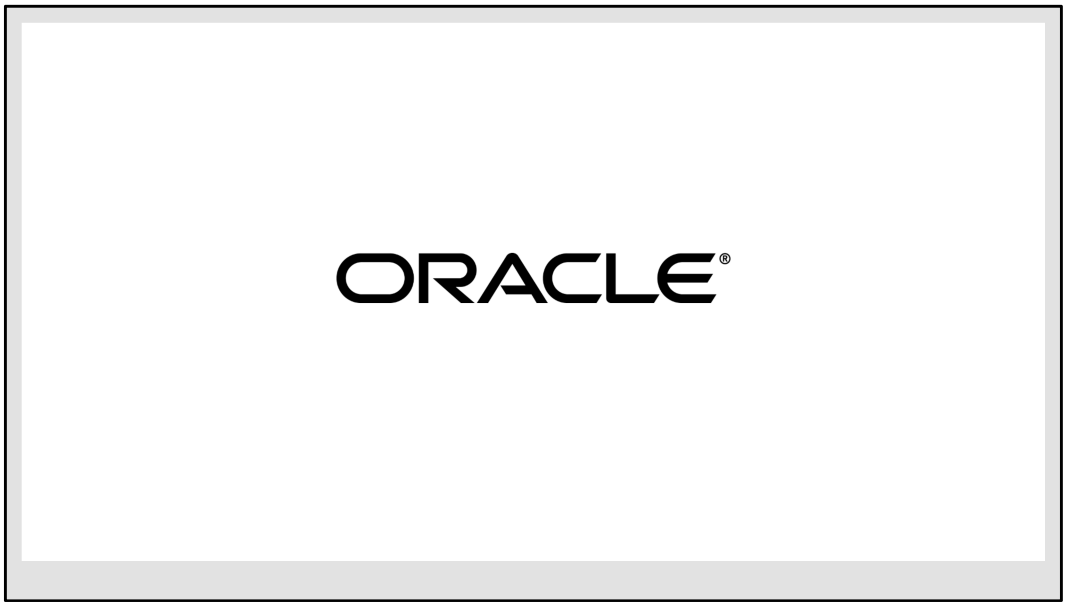


Before dropping the lock the new value of count is preserved for returning to the caller.



The "}" closes the transaction code block committing the transaction. This drops the lock on the mutex and discards the undo so that another transaction can use the undo space.

Q & A


ORACLE®