# Proper maintenance and care of multi-threading locks

The following strategies are used to ensure that the code is dead-lock free (generally by addressing the 4th Coffman condition: circular wait).

> 1. structure code such that only one lock will need to be acquired at a time
> 2. always acquire shared locks in the same order, as given by the table below
> 3. avoid constructs that expect to need unrestricted recursion

## Locks

Below are all of the locks that exist in the system and the mechanisms for using them that avoid the potential for deadlocks (no Ostrich algorithm allowed here):

The following are definitely leaf locks (level 1), and must not try to acquire any other lock:

> - safepoint
>
>   > Note that this lock is acquired implicitly by `JL_LOCK` and `JL_UNLOCK`. use the `_NOGC` variants to avoid that for level 1 locks.
>   >
>   > While holding this lock, the code must not do any allocation or hit any safepoints. Note that there are safepoints when doing allocation, enabling / disabling GC, entering / restoring exception frames, and taking / releasing locks.
>
> - shared_map
> - finalizers
> - pagealloc
> - gc*perm*lock
> - flisp
> - jl*in*stackwalk (Win32)

> > flisp itself is already threadsafe, this lock only protects the `jl_ast_context_list_t` pool

The following is a leaf lock (level 2), and only acquires level 1 locks (safepoint) internally:

> - typecache
> - Module->lock

The following is a level 3 lock, which can only acquire level 1 or level 2 locks internally:

> - Method->writelock

The following is a level 4 lock, which can only recurse to acquire level 1, 2, or 3 locks:

> - MethodTable->writelock

No Julia code may be called while holding a lock above this point.

The following are a level 6 lock, which can only recurse to acquire locks at lower levels:

> - codegen
> - jl*modules*mutex

The following is an almost root lock (level end-1), meaning only the root look may be held when trying to acquire it:

> - typeinf

> > this one is perhaps one of the most tricky ones, since type-inference can be invoked from many points
> >
> > currently the lock is merged with the codegen lock, since they call each other recursively

The following lock synchronizes IO operation. Be aware that doing any I/O (for example, printing warning messages or debug information) while holding any other lock listed above may result in pernicious and hard-to-find deadlocks. BE VERY CAREFUL!

> - iolock
> - Individual ThreadSynchronizers locks
>
> > this may continue to be held after releasing the iolock, or acquired without it, but be very careful to never attempt to acquire the iolock while holding it

The following is the root lock, meaning no other lock shall be held when trying to acquire it:

> - toplevel
>
> > this should be held while attempting a top-level action (such as making a new type or defining a new method): trying to obtain this lock inside a staged function will cause a deadlock condition!
> >
> > additionally, it's unclear if *any* code can safely run in parallel with an arbitrary toplevel expression, so it may require all threads to get to a safepoint first

# Broken Locks

The following locks are broken:

- toplevel

> doesn't exist right now
>
> fix: create it

- Module->lock

> This is vulnerable to deadlocks since it can't be certain it is acquired in sequence. Some operations (such as `import_module`) are missing a lock.
>
> fix: replace with `jl_modules_mutex`?

- loading.jl: `require` and `register_root_module`

> This file potentially has numerous problems.
>
> fix: needs locks

# Shared Global Data Structures

These data structures each need locks due to being shared mutable global state. It is the inverse list for the above lock priority list. This list does not include level 1 leaf resources due to their simplicity.

MethodTable modifications (def, cache, kwsorter type) : MethodTable->writelock

Type declarations : toplevel lock

Type application : typecache lock

Global variable tables : Module->lock

Module serializer : toplevel lock

JIT & type-inference : codegen lock

MethodInstance/CodeInstance updates : Method->writelock, codegen lock

> - These are set at construction and immutable:
>   - specTypes
>   - sparam_vals

> - def

> - These are set by `jl_type_infer` (while holding codegen lock):
>   - cache
>   - rettype
>   - inferred

```
* valid ages
```

> - `inInference` flag:
>   - optimization to quickly avoid recurring into `jl_type_infer` while it is already running
>   - actual state (of setting `inferred`, then `fptr`) is protected by codegen lock

> - Function pointers:
>   - these transition once, from `NULL` to a value, while the codegen lock is held
> - Code-generator cache (the contents of `functionObjectsDecls`):
>   - these can transition multiple times, but only while the codegen lock is held
>   - it is valid to use old version of this, or block for new versions of this, so races are benign, as long as the code is careful not to reference other data in the method instance (such as `rettype`) and assume it is coordinated, unless also holding the codegen lock

LLVMContext : codegen lock

Method : Method->writelock

- roots array (serializer and codegen)
- invoke / specializations / tfunc modifications

---

« Bounds checking Arrays with custom indices »