

**Private-by-default threads:** an approach that flips threads on their head. Right now, threads share all address space. The idea here is to create an implementation of threads with processes, à la Sheriff / Dthreads, where all data is private by default. Only by explicitly requesting shared objects (for example, by subtyping from a Shared class, or allocating from a special shared heap) would objects be accessible across threads.

## Implementation Plan

The memory model is shared nothing. Objects that are shared have to inherit from a shared object. The shared object contains metadata related to ownership and locks. When a shared object is allocated memory, the same allocation is done on all the other processes. The process which originally allocates memory for the object is considered to be the owner of the object.

The owner coordinates locking on the object. Any other process which wants to modify the object has to first obtain a lock on the object. Locking is simple and does not involve any queues. When a process wants the lock on an object, it first checks to see if it is the owner. If it is, it verifies whether the lock is already taken. If it is not, it takes the lock and proceeds. If it is already taken, it (busy?) waits till the lock is released. If the process is not the owner, it requests the owner to give it the lock. The owner replies with a success or a failure. The requestor retries until the lock is taken (exponential time between retries?) A process which holds the lock can modify any field it wants.

Updates to any of the fields have to be intercepted. I have to see if there is any elegant way of doing it, or just requiring that the setters of all the objects invoke a call to the library at entry and exit. This of course requires that elements are modified or accessed only via getters and setters. The scope of the locks is for an object, but locks are taken and released every time a field is modified, and so, locking is not composable. An additional lock is required to atomically modify 2 fields, which seems reasonable.

Any updates to a field triggers a message to all the other processes, and each such message contains the object ID, which is common to all the processes and shared during object creation, the offset of the field that is to be modified, the size of the field and the value to be written in. If the field is a pointer, the object it points to should also be a shared object. In this case, the ID of the “pointed-to object” is also shared. In the naïve implementation, pointers to the interior of an object are not permitted, but there is no reason not to send that offset too.

Reads require a lock local to the process, just to ensure that we are not reading while a write is in progress.

One aspect that has not yet been considered is how to refer to the type of the object created in the other processes, or if that matters at all. As soon as the new memory allocated is set to a pointer in an existing object, we know exactly what its type is. Before this is done, any fields that are modified in the object are modified in an identical manner using offsets. We will just need to maintain a hash table that maps the object ID to the actual memory location of the object.