Fiber-local storage

Fiber-local storage (fls) is a mechanism for associating a fiber with some implicit, local state. There are three core operations. The first creates an empty local storage; the second sets the fls for the host vproc; the third gets the fls on the host vproc.

```plaintext
type fls  
val new : unit -> fls  
val set : fls -> unit  
val get : unit -> fls  
```

To make fls useful, we provide some basic dictionary operations. Keys into the dictionary are represented by the `<code>tag</code>` type. The argument to the tag type is a phantom type, "i.e."

```plaintext
type 'a tag = int  
val key1 : key1_ty tag = 1  
val key2 : key2_ty tag = 2  
...
```

We have operations for adding and finding elements.

```plaintext
val add : (fls * ('a tag * 'a)) -> fls  
val find : (fls * 'a tag) -> 'a option  
```

Fiber-group storage

In certain cases, we wish to associate some implicit, shared state with a "group" of fibers. For read-only access, our fls mechanism is already sufficient. But for read-write access, we need synchronized memory.

Example: scheduler initialization for futures

Our architecture allows each thread to use multiple schedulers. To avoid unnecessary overhead, we delay initialization until the last possible moment, when the first spawn operation occurs.

Unfortunately, because of our architecture, we need to synchronize the process of initializing schedulers. Suppose that we have two versions of futures, one subject to a breadth-first scheduling policy and the other to a depth first policy. It is easy to imagine a program that uses both schedulers, such as the one below. Unfortunately, we cannot know which call to `<code>futureDFS</code>` happens first, so we need some synchronized memory.

```plaintext
futureBFS(fn () => futureDFS(fn () => e1)) + futureDFS(fn () => e2)  
```

The function below handles the initialization of our DFS scheduler. As a convenience, it returns the ready queue of the scheduler, hence the name. This function starts by finding the shared scheduler state in the fls. It then uses `Set-once memory` to start the scheduler and get the handle on the ready queue.

```plaintext
fun getDFSReadyQ () = let  
  val fls = FLS.get()  
  val SOME schedState = FLS.find(fls, tag(futureDFS))  
in  
  SetOnceMem.set(schedState,  
  fn () => let  
    val readyQ = LockedQueue.new()  
in  
    startDFSSched readyQ;  
    readyQ  
  end)  
end  

Now it is easy to spawn a future on the DFS scheduler. We allocate the future, put it on the ready queue, and return the handle.

```plaintext
fun futureDFS thunk = let  
  val fut = alloc(EMPTY_FUTURE, thunk)  
in  
  LockedQueue.enqueue(getDFSReadyQ(), fut);  
fut  
end  
```

Thread capabilities

Because threads consist of one or more fibers, they necessarily have read-only access to fls. Thus, any local storage entry must be added during the `<code>spawn</code>` operation. To address this issue, we use the notion of "thread capabilities". For example, a given thread could have a capability for depth-first or breadth-first futures.

```plaintext
spawn(f1, FutureDFS.capability);  
spawn(f2, FutureBFS.capability)  
```

Capabilities are just entries in the fiber-local storage dictionary. Before spawning a thread, we add all the capabilities to its local storage.
type 'a capability = 'a tag * 'a
fun spawn (thunk, capabilities) = let
  val fls = List.foldl FLS.add (FLS.new()) capabilities
in
  VProcQueue.enqueue(fls, Control.fiber thunk);
  fls
end

Our implementation defines a set of default capabilities for threads. See `<code>src/lib/basis/runtime/utils/default-thread-capabilities.pml</code>` for the complete list.