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.

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.",
type 'a tag = int
val key1 : key1_ty tag = 1
val key2 : key2_ty tag = 2

... We have operations for adding and finding elements.
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.

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.

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.

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.

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.