Доработки библиотеки hpx

> Вырастает из темы: *Доработки hpx*

p0233 – 2017-10-15 HP

p0566 – 2018-05-06 Proposed wording HP and RCU

p1121 – 2019-01-20 Proposed wording and interface for HP

folly – Реализация, приближенная к p0566. Использует части библиотеки folly(SingletonThreadLocal, SingletonManager, folly:Executor).

libcds – Другая реализация HP.

folly умеет использовать не только thread_local, libcds использует только thread_local storage.

В p0233 написано “Due to the performance advantages of using TLS, the library implementation should allow the programmer to choose implementation paths that benefit from TLS when suitable, and avoid TLS when incompatible with the use case.”

libcds

```
cds::Initialize(); once
cds::gc::HP hpGC; once
cds::threading::Manager::attachThread(); every thread
cds::threading::Manager::detachThread(); every thread
```

http://libcds.sourceforge.net/doc/cds-api/index.html#cds_how_to_use

Не умеет в разные HP_domain, умеет только в собственный thread_local tls, не особо гибкий. Хотя умеет в

```
set_memory_allocator(
    void* ( *alloc_func )( size_t size ),   ///< \p malloc() function
    void( *free_func )( void * p )  )
```

class HP – главный класс, Before use any HP-related class you must initialize \p %HP by contructing \p %cds::gc::HP object in beginning of your \p main().

class Guard – A guard is a hazard pointer. Additionally, the Guard class manages allocation and deallocation of the hazard pointer.

```
Guard::protect(atomics::atomic<T> const& toGuard)
Guard::protect(atomics::atomic<T> const& toGuard, Func f)
```

Приводит T* к void* и работает с этими указателями.

```
template <class Disposer, typename T>
static void retire( T * p )
```

Disposer это шаблонный параметр, и на самом деле тип, а не объект. В стандарте должен быть объектом, “Registers the expression reclaim(static_cast<T*>(this)) to be evaluated asynchronously”

if на linux и membarrier

folly

Аналог Guard, protect == get_protected

```
hazptr_holder: Class that owns and manages a hazard pointer.
T* hazptr_holder::get_protected(const Atom<T*>& src) noexcept;
```

В folly Используется T*, в libcds: T

Все T обязаны наследоваться от hazptr_obj_base<T>

```
template <
    typename T,
    template <typename> class Atom = std::atomic,
    typename D = std::default_delete<T>>
class hazptr_obj_base {
    void retire( D deleter = {}, hazptr_domain<Atom>& domain =
default_hazptr_domain<Atom>());
}
```

hazptr_domain в folly есть, но он не умеет в разные аллокаторы, хотя по стандарту должен.

Дополнительные вещи, не из стандарта:

hazptr_array<N> для N hazptr-ов сразу, быстрее.

hazptr_local<N> немного быстрее, но обязывает иметь ровно 1 активный hazptr_* на поток

p1121 (последний)

```
header <hazard_pointer>
```

```
// ?.?, Class hazard_pointer_domain:
class hazard_pointer_domain;

// ?.?, Default hazard_pointer_domain:
hazard_pointer_domain& hazard_pointer_default_domain() noexcept;
```

```
// ?.?, For a set of hazard_pointer_obj_base objects O in □domain□ for
which O.retire(reclaim, domain) has been called, ensures that O has been
reclaimed.
void hazard_pointer_clean_up(    hazard_pointer_domain& domain =
hazard_pointer_default_domain());

// ?.?, Class template hazard_pointer_obj_base:
template <typename T, typename D = default_delete<T>>  class
hazard_pointer_obj_base;4

// ?.?, Class hazard_pointer
class hazard_pointer;

// ?.?, Construct non-empty hazard_pointer
hazard_pointer make_hazard_pointer(    hazard_pointer_domain& domain =
hazard_pointer_default_domain());

// ?.?, Hazard pointer swap
void swap(hazard_pointer&, hazard_pointer&) noexcept;
```

```
class hazard_pointer_domain {
        public:
        // ?.?.? constructor:
        explicit hazard_pointer_domain(
std::pmr::polymorphic_allocator<byte> poly_alloc = {});
      ...
      }
      class hazard_pointer {
      public:
...
bool empty() const noexcept;
template <typename T>    T* protect(const atomic<T*>& src) noexcept;
template <typename T>  bool try_protect(T*& ptr, const atomic<T*>& src)
noexcept;
template <typename T>  void reset_protection(const T* ptr) noexcept;
void reset_protection(nullptr_t = nullptr) noexcept;
void swap(hazard_pointer&) noexcept;};
```

Аргумент против standard proposal(Hooks)

Boost.Intrusive

В стандарте сейчас используется что-то похожее на base_hook

В Boost.Intrusive говорится(И это и есть аргумент):

```
Sometimes an 'is-a' relationship between list hooks and the list value types
is not desirable.
```

```
In this case, using a member hook as a data member instead of 'disturbing'
the hierarchy might be the right way:
```

А потом ещё и

```
A programmer might find that base or member hooks are not flexible enough in
some situations.
In some applications it would be optimal to put a hook deep inside a member
of a class or just outside the class.
Boost.Intrusive has an easy option to allow such cases: function_hook.
```

Как можно исправить

И кажется, что в стандарт можно записать эти три варианта сразу. То есть позволить программисту самому определять, как он хочет, чтобы указатель на его структуру преобразовывался в указатель на HP.

Стандарт говорит, что “Наследуйся от hazptr_obj_base и приводиться будет к нему”.

Я могу предложить аналог member_hook: “Вставь в себя typedef на тот тип, к указателю на который ты будешь приводиться при помещении в массив HP”

А аналог function_hook могу предложить такой:

Boost.Intrusive:

```
//This functor converts between value_type and a hook_type
      struct Functor
      {
         //Required types
         typedef /*impl-defined*/      hook_type;
         typedef /*impl-defined*/      hook_ptr;
         typedef /*impl-defined*/      const_hook_ptr;
         typedef /*impl-defined*/      value_type;
         typedef /*impl-defined*/      pointer;
         typedef /*impl-defined*/      const_pointer;
         //Required static functions
         static hook_ptr to_hook_ptr (value_type &value);
         static const_hook_ptr to_hook_ptr(const value_type &value);
         static pointer to_value_ptr(hook_ptr n);
         static const_pointer to_value_ptr(const_hook_ptr n);
      };
```

My version(hook → hazptr):

```
//This functor converts between value_type and a hazptr_type
      struct Functor
      {
         //Required types
```

```
        typedef /*impl-defined*/      hazptr_type;
        typedef /*impl-defined*/      hazptr_ptr;
        typedef /*impl-defined*/      const_hazptr_ptr;
        typedef /*impl-defined*/      value_type;
        typedef /*impl-defined*/      pointer;
        typedef /*impl-defined*/      const_pointer;
        //Required static functions
        static hazptr_ptr to_hazptr_ptr (value_type &value);
        static const_hazptr_ptr to_hazptr_ptr(const value_type &value);
        static pointer to_value_ptr(hazptr_ptr n);
        static const_pointer to_value_ptr(const_hazptr_ptr n);
     };
```

То есть с помощью Functor нужно будет самому задать, указатели на объекты какого типа нужно будет хранить в массиве HP.

Кажется бесполезное преимущество, т.к. добавляет память : Более того, можно будет добавить данных в хуки(hazptr_obj_base), и тем самым влиять на структуру массива HP(сделать его Boost.Intrusive.list). Я не придумал ещё полезных применений(можно хранить timestamp, можно хранить указатель на функцию, которую вызывать при освобождении hp(для статистики мб))

В массиве HP в libcds хранится каст к void*. В proposal'е каст к hazptr_obj_base*.

Моя идея в том, чтобы как и в Boost.Intrusive, добавить к hazptr_domain или hazard_pointer::protect шаблонный параметр, указывающий на то, как получать hazptr_ptr из нашего класса

Почему их hazptr_obj_base -- плохо?

Как говорится в Boost.Intrusive: Sometimes an 'is-a' relationship between list hooks and the list value types is not desirable.

Мои варианты: Множественное наследование от двух типов, каждый из которых наследуется от hazptr_obj_base Function hook решает эту проблему.