

sync_ptr

pointer chaining and stealing using RAI technique

shared_ptr

```
struct Obj  
{ ... };  
  
std::shared_ptr<Obj> ptr1 = std::make_shared<Obj>();  
std::shared_ptr<Obj> ptr2 = ptr1;  
assert(ptr1 == ptr2);
```

shared_ptr

```
struct Obj
{ ... };

std::shared_ptr<Obj> ptr1 = std::make_shared<Obj>();
std::shared_ptr<Obj> ptr2 = ptr1;
assert(ptr1 == ptr2);

ptr1.reset(new Obj);
assert(ptr1 != ptr2);
```

shared_ptr

```
struct Obj
{ ... };

std::shared_ptr<Obj> ptr1 = std::make_shared<Obj>();
std::shared_ptr<Obj> ptr2 = ptr1;
assert(ptr1 == ptr2);

ptr1.reset(new Obj);
assert(ptr1 != ptr2);
// What if ptr1 == ptr2 ?
```

sync_ptr

```
if ptr1 == ptr2
```

Bound

Connection

Synchronization

→ Common Behaviour

sync_ptr

if **ptr1 == ptr2**

- Common Propagation
- Common Set
- Common Reset
- Common Release
- Common Steal
- ...

Single Call Operation

sync_ptr

```
struct Obj  
{ ... };  
  
sync_ptr<Obj> ptr1 = make_sync<Obj>();  
sync_ptr<Obj> ptr2 = ptr1;  
assert(ptr1 == ptr2);
```

sync_ptr

```
struct Obj  
{ ... };  
  
sync_ptr<Obj> ptr1 = make_sync<Obj>();  
sync_ptr<Obj> ptr2 = ptr1;  
assert(ptr1 == ptr2);  
  
ptr1.reset(new Obj);  
assert(ptr1 == ptr2);
```


Singleton

```
class widget
{
public:
    static widget * widget::instance();
    void do_something();
};

widget * widget::instance()
{
    static widget w;
    return &w;
}

widget::instance()->do_something();
```

Singleton

```
class widget
{
public:
    static widget * widget::instance();
    void do_something();
    void do_more();
    void do_even_more();
};
```

```
widget * widget::instance()
{
    static widget w;
    return &w;
}
```

```
widget::instance()->do_something();
widget::instance()->do_more();
widget::instance()->do_even_more();
```

Singleton

```
class widget
{
private:
    static std::shared_ptr<widget> widget_;
public:
    static std::shared_ptr<widget> widget::instance()
    { return widget_; }
    void do_something();
    void do_more();
    void do_even_more();
};
```

```
auto w = widget::instance();
w->do_something();
w->do_more();
w->do_even_more();
```

Singleton

```
class widget
{
private:
    static sync_ptr<widget> widget_;
public:
    static sync_ptr<widget> widget::instance()
    { return widget_; }
    void do_something();
    void do_more();
    void do_even_more();
};
```

```
auto w = widget::instance();
w->do_something();
w->do_more();
w->do_even_more();
```

Singleton

```
auto w = widget::instance();  
w->do_something();  
w->do_more();  
w->do_even_more();
```

```
// Our widget is exhausted, lets use a fresh one.
```

```
w.reset(new widget());  
w->do_something(); ...
```

Singleton

```
auto w = widget::instance();  
w->do_something();  
w->do_more();  
w->do_even_more();  
  
// Our widget is exhausted, lets use a fresh one.  
  
w.reset(new widget());  
w->do_something(); ...  
  
// Lets all use that fresh one.  
  
auto w2 = widget::instance();  
w2->do_something(); ...
```

Singleton

```
auto w = widget::instance();  
w->do_something();  
w->do_more();  
w->do_even_more();
```

```
// Our widget has better things to do.  
std::unique_ptr<widget> busy(w.exchange(new widget()));  
w->do_something(); ...
```

```
// busy can now go on with his duty, a new widget is in use.  
auto w2 = widget::instance();  
w2->do_something();
```

```
busy->do_something_else();
```

Context

```
struct context
{
    void execute(command * p_ptr) const
    { /* Do something with the command. */ }
};

struct command
{
    void do_something(context * p_ctx)
    {
        p_ctx->execute(this);
    }
};

context * ctx = new context();
command * cmd = new command();

cmd->do_something(ctx);

delete cmd;
delete ctx;
```


Context

```
struct context
{
    void execute(command * p_ptr) const
    { /* Do something with the command. */ }
};

struct command
{
    void execute(std::unique_ptr<context> const & p_ctx)
    {
        p_ctx->execute(this);
    }
};

std::unique_ptr<context> ctx = std::make_unique<context>();
std::unique_ptr<command> cmd = std::make_unique<command>();

cmd->execute(ctx);
```

Context

```
struct context
{
    void execute(command * p_ptr) const
    { /* Do something with the command. */ }
};

struct command
{
    // Lets be clever :p
    context const * ctx_;
    void execute(std::unique_ptr<context> const & p_ctx)
    {
        p_ctx->execute(this);
        ctx_ = p_ctx.get();
    }

    // Wups ...
    void not_so_clever_execute()
    {
        ctx_->execute(this);
    }
};
```

Context

```
struct command
{
    sync_ptr<context> ctx_;

    unit(sync_ptr<context> const & p_ctx) : ctx_(p_ctx) {}

    void execute()
    {
        ctx_->execute(this);
    }

    void not_so_clever_execute()
    {
        ctx_->execute(this);
    }
};

sync_ptr<context>          ctx = make_sync<context>();
std::unique_ptr<command> cmd = std::make_unique<command>(ctx);

cmd->execute();

// Change the context.
ctx.reset(new context());

cmd->not_so_clever_execute();
```

Context

```
struct immediate_context : public context
{
    void execute(command * p_ptr)
    { /* Do something with command immediately. */ }
};

struct deferred_context : public context
{
    void execute(command * p_ptr)
    { /* Enqueue command for later execution. */ }
};

auto ctx = make_sync<immediate_context>();
auto cmd = std::make_unique<command>(ctx);

// Immediate execution.
cmd->execute();

// Change the context.
ctx.reset(new deferred_context());

// Deferred execution.
cmd->execute();
```

Context

DirectX 11

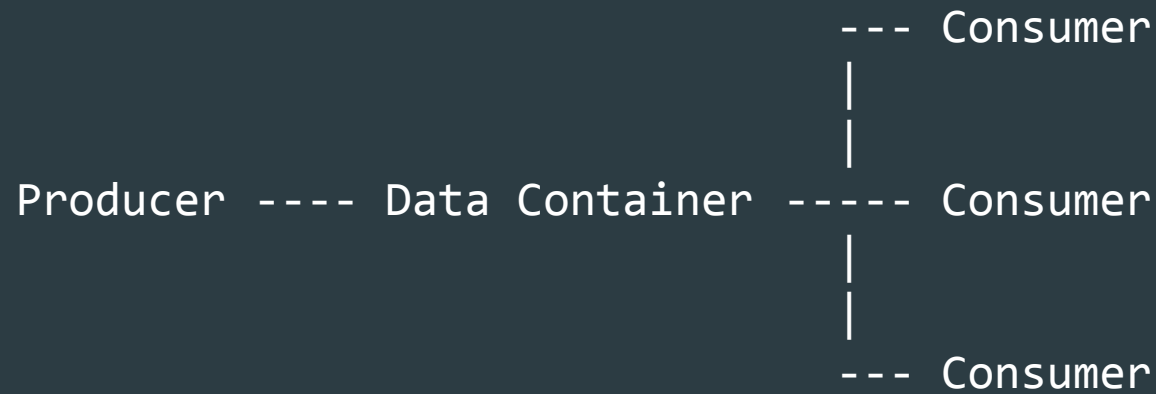
```
sync_ptr<ID3D11DeviceContext> immediate_;
sync_ptr<ID3D11DeviceContext> deferred_;

class cbuffer
{
private:
    ID3D11Buffer *          buffer_;
    sync_ptr<ID3D11DeviceContext> ctx_;

public:
    cbuffer(sync_ptr<ID3D11DeviceContext> const & p_ctx) : ctx_(p_ctx) { ... }

    void bind_vs(void) noexcept
    {
        ctx_->VSSetConstantBuffers(0, 1, &buffer_);
    }
}
```

Producer Consumer



Assume consumer only wants the most up to date data.

Producer Consumer

```
class producer
{
private:
    sync_ptr<Data> data_;

public:
    producer(sync_ptr<Data> const & p_data) : data_(p_data) { ... }

    void produce(void) noexcept
    {
        // Produce newData.
        ...

        // Data is free.
        if (!data_)
        {
            data_.reset(newData);
        }
    }
    ...
}
```

Producer Consumer

```
class consumer
{
private:
    sync_ptr<Data> data_;

public:
    consumer(sync_ptr<Data> const & p_data) : data_(p_data) { ... }

    void consume(void) noexcept
    {
        // Get data.
        auto data = data_.release();
        if (data)
        {
            // Consume data.
        }
    }
    ...
}
```


sync_ptr

`sync_ptr` stands for **synchronized pointer**.

`sync_ptr` objects that are **copy constructed** or **copy assigned** point to the same underlying pointer.

When the original `sync_ptr` or one of its copy underlying pointer changes, all `sync_ptr` and copies point to the updated pointer.

`sync_ptr` and underlying pointer are reference counted.

The underlying pointer memory is returned when the reference count drops to zero or another pointer is assigned.

Implementation

`sync_ptr` come in 2 different flavours

Policy

- strong execution guarantee
- thread safety using default policies
- easily extensible using the provided policies or any desired one

Default policy uses `std::recursive_mutex`.

Atomic

- faster concurrent environment execution (lock free and wait free)
- weak execution guarantee

All operation return their success state
leaving the programmer the choice in the response strategy.

Both offer **no-throw** exception guarantee.

Implementation

```
// Construct default empty object.  
sync_ptr();  
  
// Construct with compatible pointer.  
template<class TPtrCompatible>  
sync_ptr(TPtrCompatible * p_ptr);  
  
// Move ctor and assignment operator (p_rhs becomes empty).  
sync_ptr(sync_ptr && p_rhs);  
sync_ptr_t & operator=(sync_ptr_t && p_rhs);  
  
// Copy constructor and assignment operator  
// (create a bound, increase reference count).  
sync_ptr(sync_ptr const & p_rhs);  
sync_ptr_t & operator=(sync_ptr_t const & p_rhs) &;
```

Implementation

Policy

```
template <
    class TPtr,
    template <class T> class TDeleter,
    template <class T> class THolder,
    class TRefCounter>
class sync_ptr
{ ... }

TPtr           // Stored object type.
TDeleter       // Memory reclamation (free, delete, ...).
THolder        // Stored object holder (thread safe access, ...).
TRefCounter    // Stored object reference counter.
```

Implementation

Policy

Deleter

```
void free(T * p_ptr) const noexcept;
```

Holder

```
T * set(T * p_ptr) noexcept;
```

```
T * get() const noexcept;
```

Reference Counter

```
void increment(void) noexcept;
```

```
size_t decrement(void) noexcept;
```

```
void increment_ptr(void) noexcept;
```

```
size_t decrement_ptr(void) noexcept;
```

```
size_t count(void) const noexcept;
```

```
inline size_t count_ptr(void) const noexcept;
```

Implementation

Policy

Internal class

```
template<
    class TPtr,
    template <class T> class TDeleter,
    template <class T> class THolder,
    class TRefCounter>
class ref_count_ptr
    : private TDeleter<TPtr>
    , private THolder<TPtr>
    , private TRefCounter
{ ... }
```

Holds and managed the object

Implementation

Policy

```
sync_ptr<T> ptr1 = make_sync<T>();  
sync_ptr<T> ptr2(ptr1);  
sync_ptr<T> ptr3 = ptr2;
```

```
assert(ptr1 == ptr2);  
assert(ptr2 == ptr3);
```

They point to the same `ref_count_ptr`

Implementation

Policy

```
// Swap the internal ref_count_ptr
void swap(sync_ptr & p_rhs) noexcept
{
    auto tmp = ref_count_ptr_;
    ref_ = p_rhs.ref_count_ptr_;
    p_rhs.ref_count_ptr_ = tmp;
}
```


Implementation

Policy

```
// Set managed object and free previous one.  
void reset(TPtr * p_ptr) noexcept;  
// Set managed object to null and free previous one.  
void reset(void) noexcept;  
  
// Release the ownership of the managed object if any.  
// Return the previously owned pointer and set the current to null.  
TPtr * release(void) noexcept;  
  
// Set managed object and return previous one.  
TPtr * exchange(TPtr * p_ptr) noexcept;
```

Implementation

Policy

```
// Access managed object.  
TPtr * get(void) const noexcept;  
TPtr & operator*(void) const noexcept;  
TPtr * operator->(void) const noexcept;  
  
// Test managed object state (!null).  
bool valid(void) const noexcept;  
operator bool(void) const noexcept;
```

Implementation

Provided Policy

```
template<
    class TType>
struct default_deleter
{
    void free(
        TType * p_ptr)
        const noexcept
    {
        static_assert(
            0 < sizeof(TType),
            "can't delete an incomplete type");
        delete p_ptr;
    }
};
```

Implementation

Provided Policy

```
class ref_counter
{
private:
    size_t    ref_count_;
    ...

    inline void increment(
        void)
        noexcept
    {
        ref_count_++;
    }
    ...
}
```

Implementation

Provided Policy

```
class atomic_ref_counter
{
private:
    std::atomic<size_t> ref_count_;
    ...

    inline void increment(
        void)
        noexcept
    {
        ref_count_.fetch_add(1U);
    }
    ...
}
```

Implementation

Provided Policy

```
template <class TPtr>
class ptr_holder
{
private:
    TPtr *    ptr_;
    ...

    inline TPtr * set(TPtr * p_ptr) noexcept
    {
        auto p = ptr_;
        ptr_ = p_ptr;
        return p;
    }

    inline TPtr * get(void) const noexcept
    {
        return ptr_;
    }
    ...
}
```

Implementation

Provided Policy

```
// Recursive mutex protected holder.
template <class TPtr>
class ptr_holder_ts
{
private:
    TPtr *          ptr_;
    mutable std::recursive_mutex mtx_;
    ...

    inline TPtr * set(TPtr * p_ptr) noexcept
    {
        std::lock_guard<std::recursive_mutex> l(mtx_);
        auto p = ptr_; ptr_ = p_ptr; return p;
    }

    inline TPtr * get(void) const noexcept
    {
        std::lock_guard<std::recursive_mutex> l(mtx_);
        return ptr_;
    }
    ...
}
```

Implementation

Provided Policy

```
// Recursive mutex protected holder.  
struct Obj  
{ ... }  
  
auto s1 = make_sync<Obj>();  
auto s2 = make_sync<Obj>();  
  
if (s1.get() != s2.get())  
{ /* non recursive is fine */ }
```


Implementation

Provided Policy

```
// Recursive mutex protected holder.  
struct Obj  
{ ... }  
  
auto s1 = make_sync<Obj>();  
auto s2 = s1  
  
if (s1.get() != s2.get())  
{  
    /* non recursive -> deadlock */  
    /* recursive is fine */  
}
```

Implementation

Provided Policy

```
// Default
template<class TPtr>
using sync_ptr_deleter          = default_deleter<TPtr>;

template<class TPtr>
using sync_ptr_holder          = ptr_holder_ts<TPtr>;

using sync_ptr_ref_counter      = atomic_ref_counter;
```

Implementation

Policy

```
// Make.
template <
    class TPtr,
    template <class T> class TDeleter = sync_ptr_deleter,
    template <class T> class THolder = sync_ptr_holder,
    class TRefCounter = sync_ptr_ref_counter,
    class... TArgs>
inline typename std::enable_if<
    !std::is_array<TPtr>::value,
    mem::sync_ptr<TPtr, TDeleter, THolder, TRefCounter>>::type
make_sync(
    TArgs&&... p_args)
{
    typedef typename sync_ptr<
        TPtr,
        TDeleter,
        THolder,
        TRefCounter> sync_ptr_t;
    return (sync_ptr_t(new TPtr(std::forward<TArgs>(p_args)...)));
}
```

Implementation

Policy

```
// Make with allocator.
template <
    class TPtr,
    template <class T> class TAllocator,
    template <class T> class TDeleter = sync_ptr_deleter,
    template <class T> class THolder = sync_ptr_holder,
    class TRefCounter = sync_ptr_ref_counter,
    class... TArgs>
inline typename std::enable_if<
    !std::is_array<TPtr>::value,
    mem::sync_ptr<TPtr, TDeleter, THolder, TRefCounter>>::type
make_sync_with_allocator(
    TAllocator<TPtr> const & p_allocator,
    TArgs&&... p_args)
{
    typedef typename sync_ptr<
        TPtr,
        TDeleter,
        THolder,
        TRefCounter> sync_ptr_t;
    return (sync_ptr_t(p_allocator.allocate(std::forward<TArgs>(p_args)...)));
}
```

Implementation

Provided Policy

```
// Default Allocator
template<
    class TType>
struct default_allocator
{
    ...
    template<class ...TArg>
    TType * allocate(
        TArg && ...p_args)
        const
    {
        static_assert(
            0 < sizeof(TType),
            "can't allocate an incomplete type");
        return new TType(std::forward<TArg>(p_args)...);
    }
    ...
}
```

Implementation

Atomic

```
// Construct default empty object.  
sync_ptr();  
  
// Construct with compatible pointer.  
template<class TPtrCompatible>  
sync_ptr(TPtrCompatible * p_ptr);  
  
// Move ctor and assignment operator (p_rhs becomes empty).  
sync_ptr(sync_ptr && p_rhs);  
sync_ptr_t & operator=(sync_ptr_t && p_rhs);  
  
// Copy constructor and assignment operator  
// (create a bound, increase reference count).  
sync_ptr(sync_ptr const & p_rhs);  
sync_ptr_t & operator=(sync_ptr_t const & p_rhs) &;
```

Implementation

Atomic

```
// Set managed object and free previous one
// -> success state.
bool reset(TPtr * p_ptr) noexcept;
// Set managed object to null and free previous one
// -> success state.
bool reset(void) noexcept;

// Release the ownership of the managed object if any.
// Return the previously owned pointer and set the current to null.
// -> success state.
bool release(TPtr ** p_out) noexcept;

// Set managed object and return previous one.
// -> success state.
bool exchange(TPtr ** p_out, TPtr * p_ptr) noexcept;
```

Implementation

Atomic

Compare And Swap (CAS)

compares the contents of a memory location to a given value and, if and only if they are the same, modifies the contents of that memory location to a given new value.

`std::atomic::compare_exchange_strong`

bitwise comparison
read-modify-write **and** load operations
no spurious failing guarantee

Implementation

Atomic

sync_ptr workhorse

```
class ref_count_ptr
{
...
    std::atomic<TPtr *> ptr_;
...

    inline bool release_ptr_cas(TPtr * p_ptr) noexcept
    {
        auto ptr = ptr_.load();
        if (ptr_.compare_exchange_strong(ptr, p_ptr))
        {
            if (ptr)
            {
                free(ptr);
            }
            return true;
        }
        return false;
    }
}
```

Implementation

Atomic

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{ /* success ! */ }
```

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{  
    /* success ! */  
}  
else  
{  
    /* failure ! */  
}
```

< thread A
< thread B

Implementation

Atomic

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{ /* success ! */ }
```

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{  
    /* success ! Now contains B value. */  
}  
else  
{  
    /* failure ! */  
}
```

< thread C
< thread A
< thread B

Implementation

Atomic

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{ /* success ! */ }
```

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))    < thread C  
{  
    /* success ! */  
}  
else  
{  
    /* failure ! Still contains B value. */    < thread A  
}
```

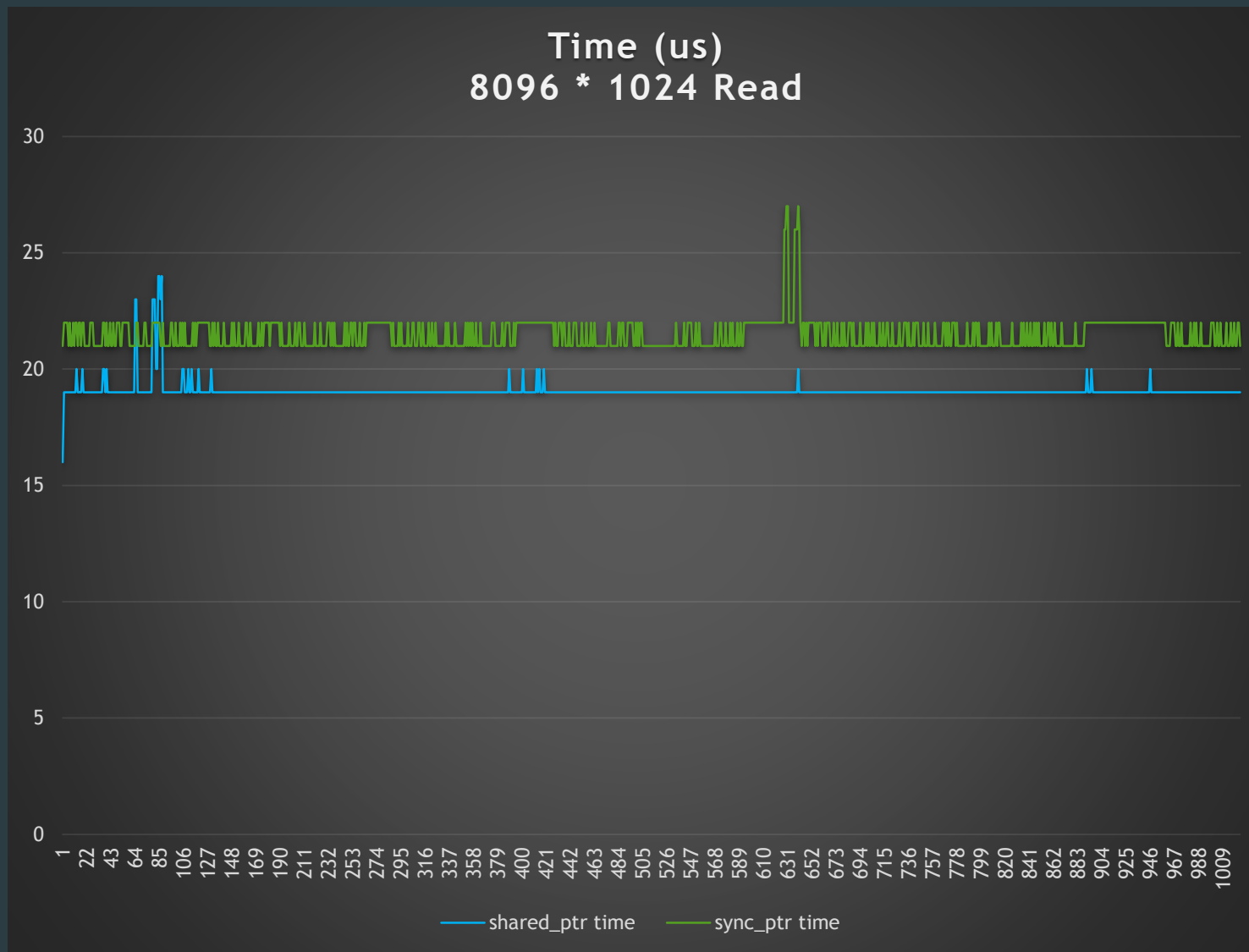
Implementation

Atomic

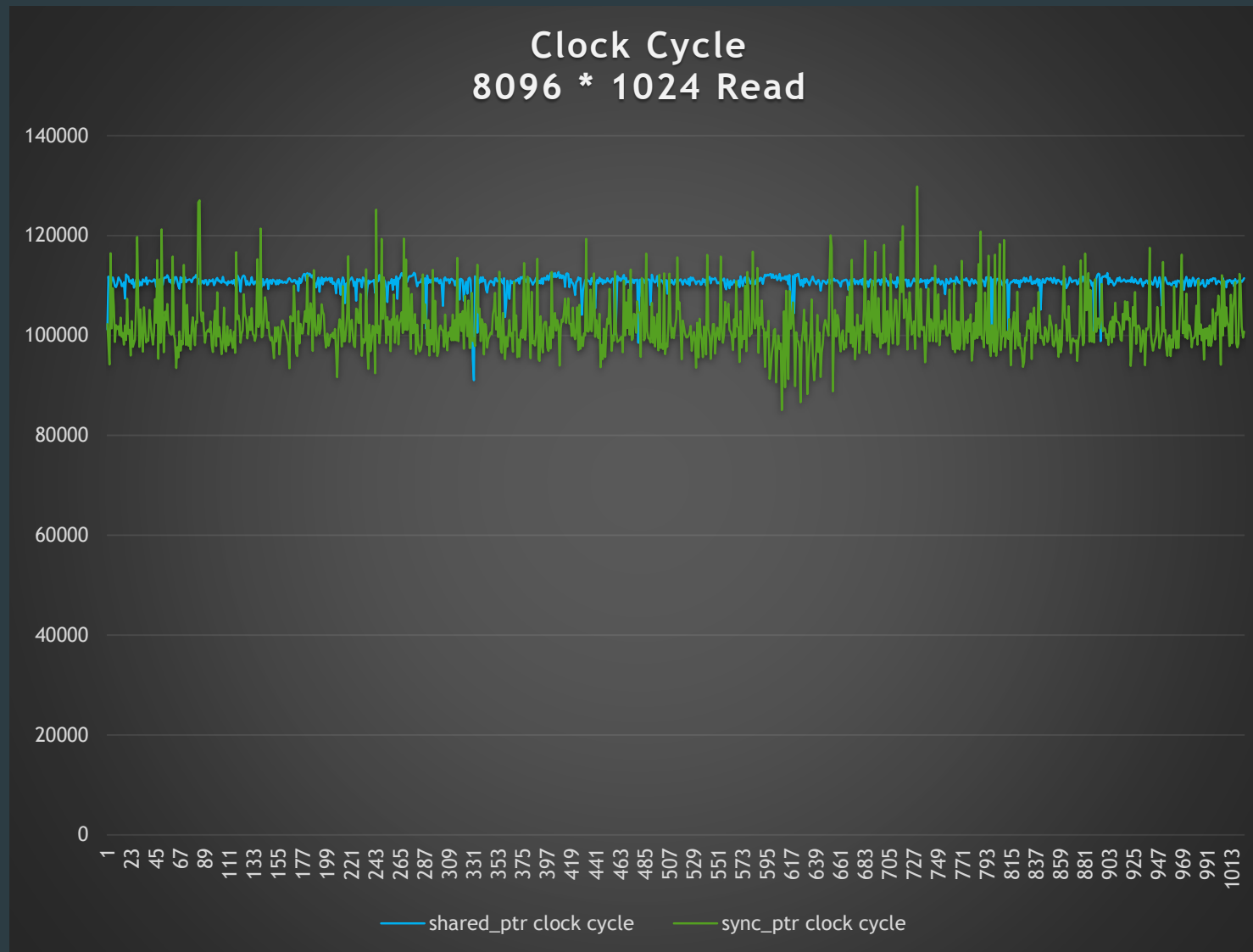
```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{ /* success ! */ }
```

```
auto ptr = ptr_.load();  
if (ptr_.compare_exchange_strong(ptr, p_ptr))  
{  
    /* success ! Now contains C value. */    < thread C  
}  
else  
{  
    /* failure ! */  
}
```

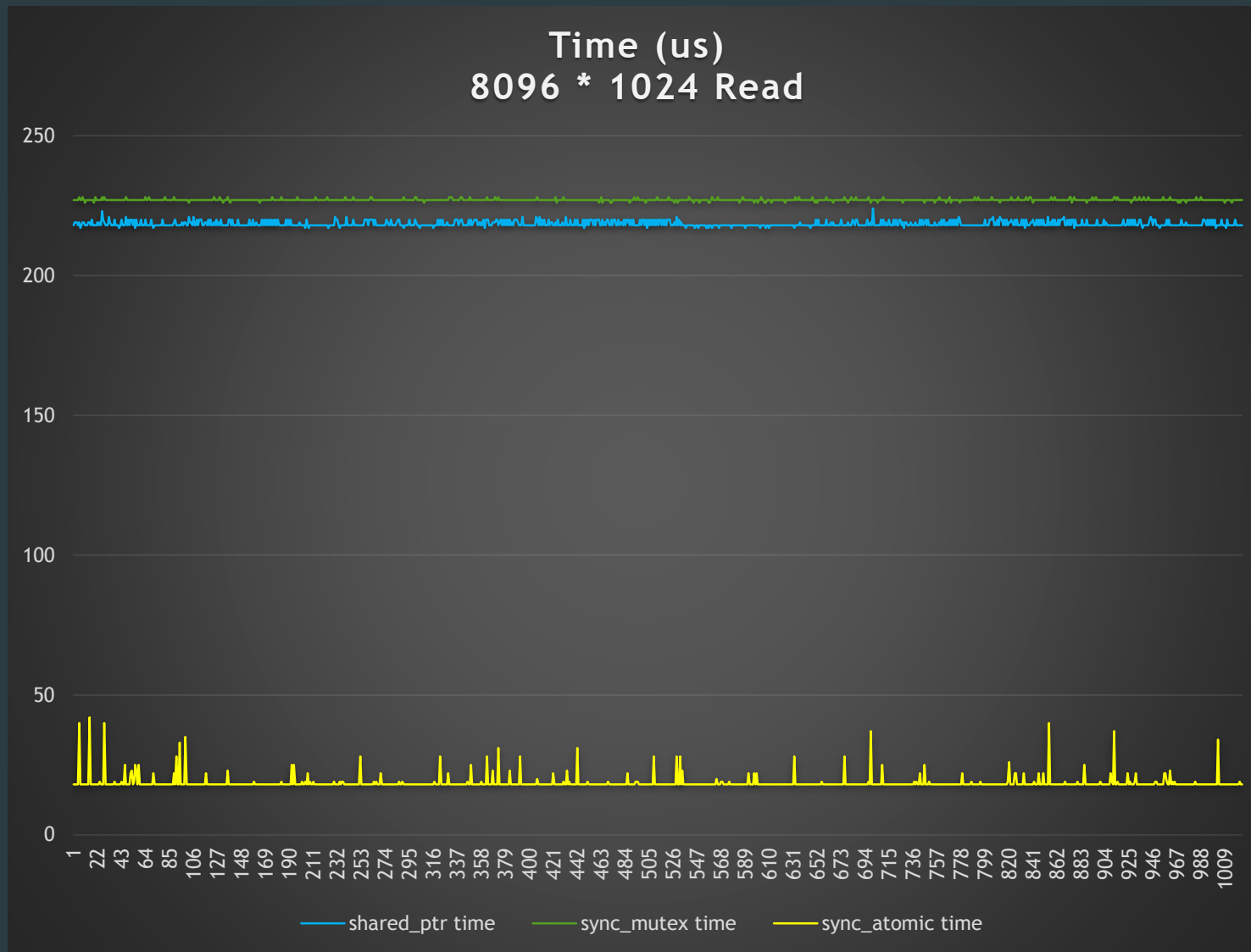
Benchmark: shared_ptr / sync_ptr



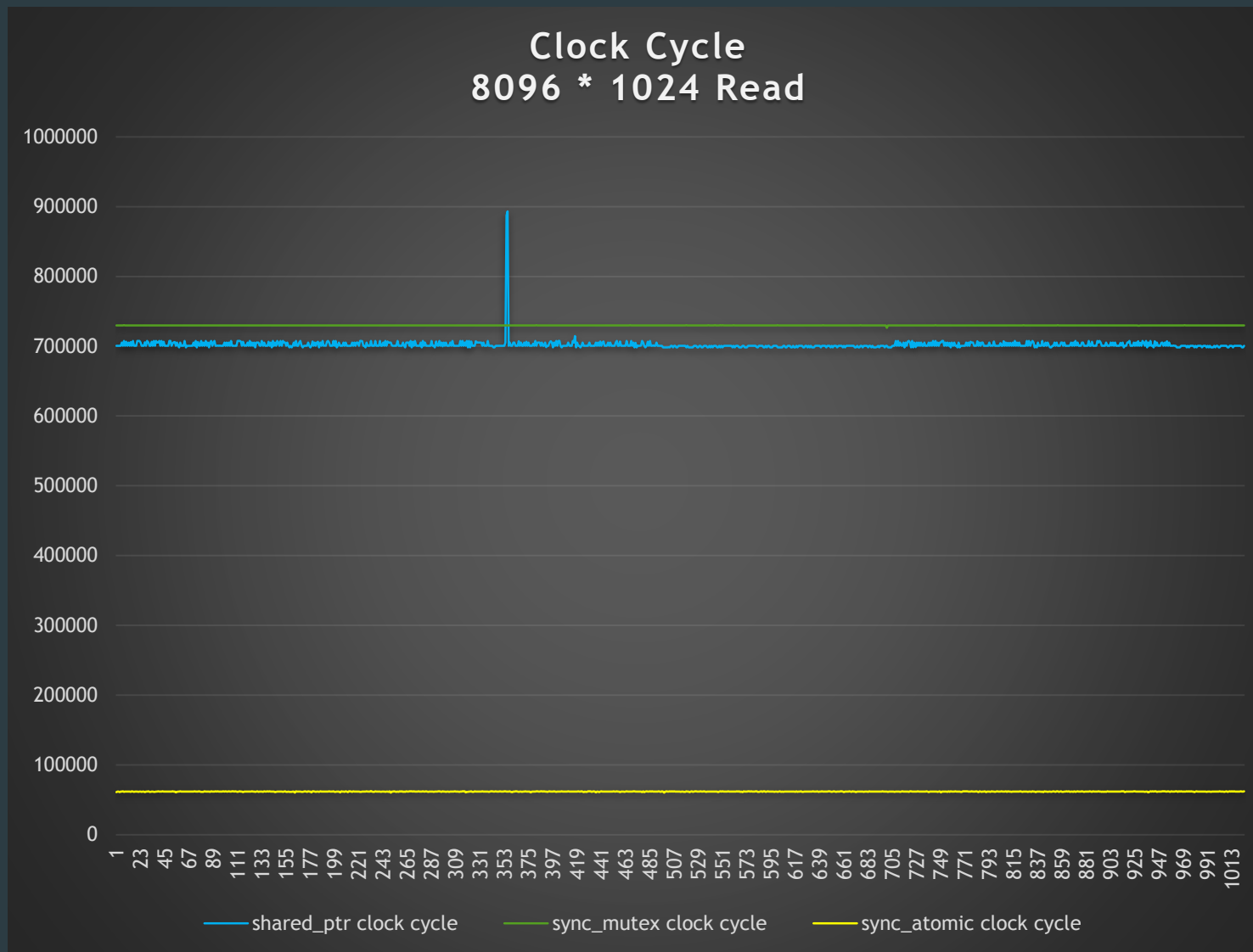
Benchmark : shared_ptr / sync_ptr



Benchmark : mutex / atomic



Benchmark : mutex / atomic



When to use it

Shared ownership / owning cycles.

Modification requires complex / costly operation.

Data race modification is time critical.

“Data path switcher”

Summary

- ✓ Chain using copy construction / copy assignment.
- ✓ Point to the same “body” (handle-body idiom).
- ✓ No need for traversal.
- ✓ Single call modification of all chained `sync_ptr`.
- ✓ 2 implementations, Policy and Atomic.
- ✓ Use when data race modification is time critical.

github.com/romaincheminade/sync_ptr

BSD License

What is next

- ✓ Faster implementation
(as fast as `std::shared_ptr` on read and write).
- ✓ Stronger execution guarantee of Atomic implementation.
- ✓ Safer release (deferred ?)
- ✓ Suggestions and participation are welcome !!!

Q & A