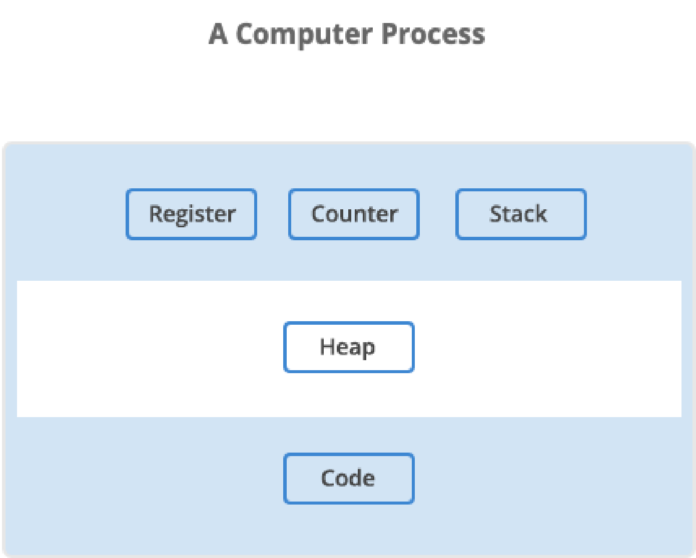
Multithreading with modern C++

Jui-Hung Hung

Memory model of a process

- Process is a running program
- Every process has its virtual memory allocation



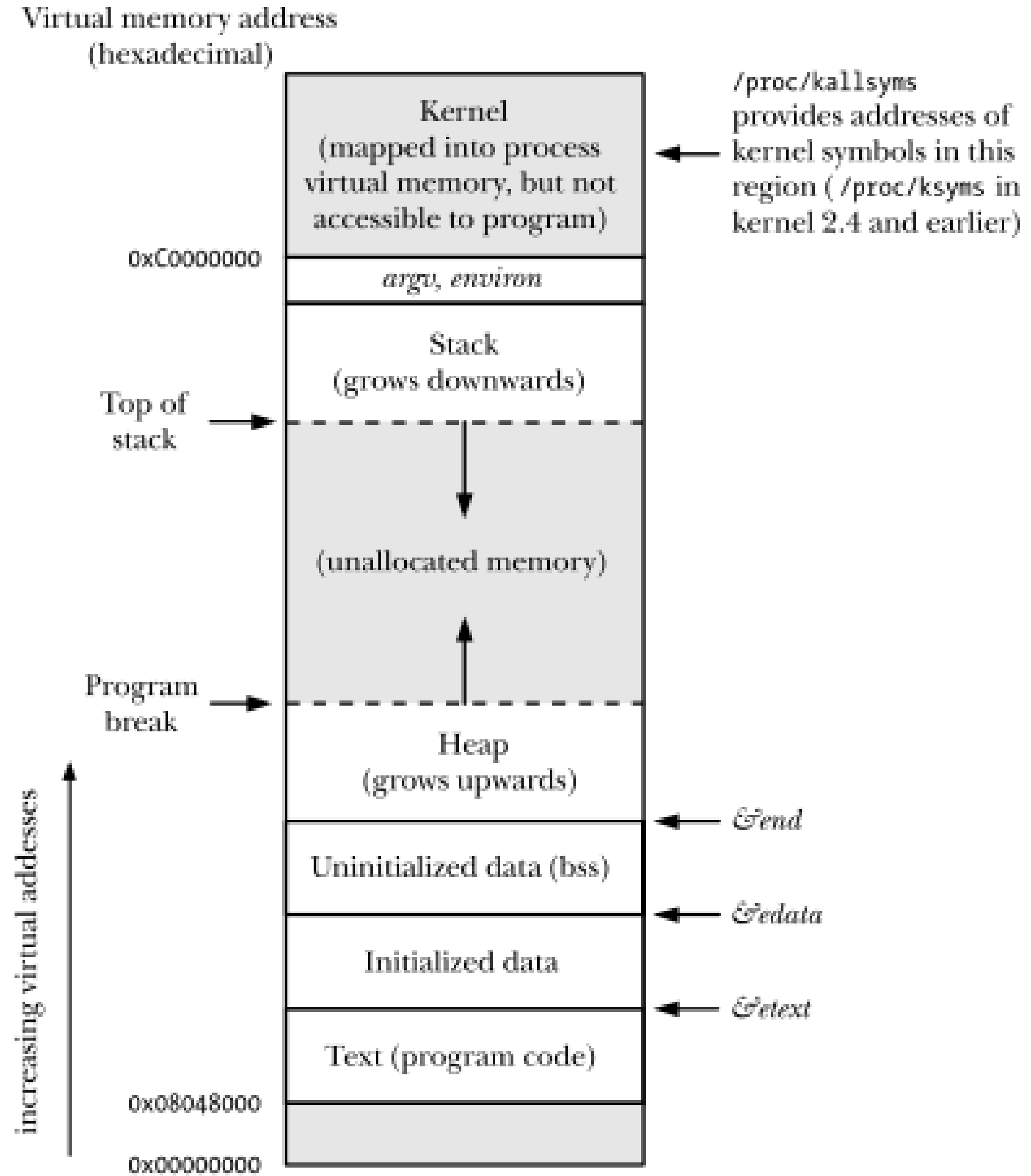
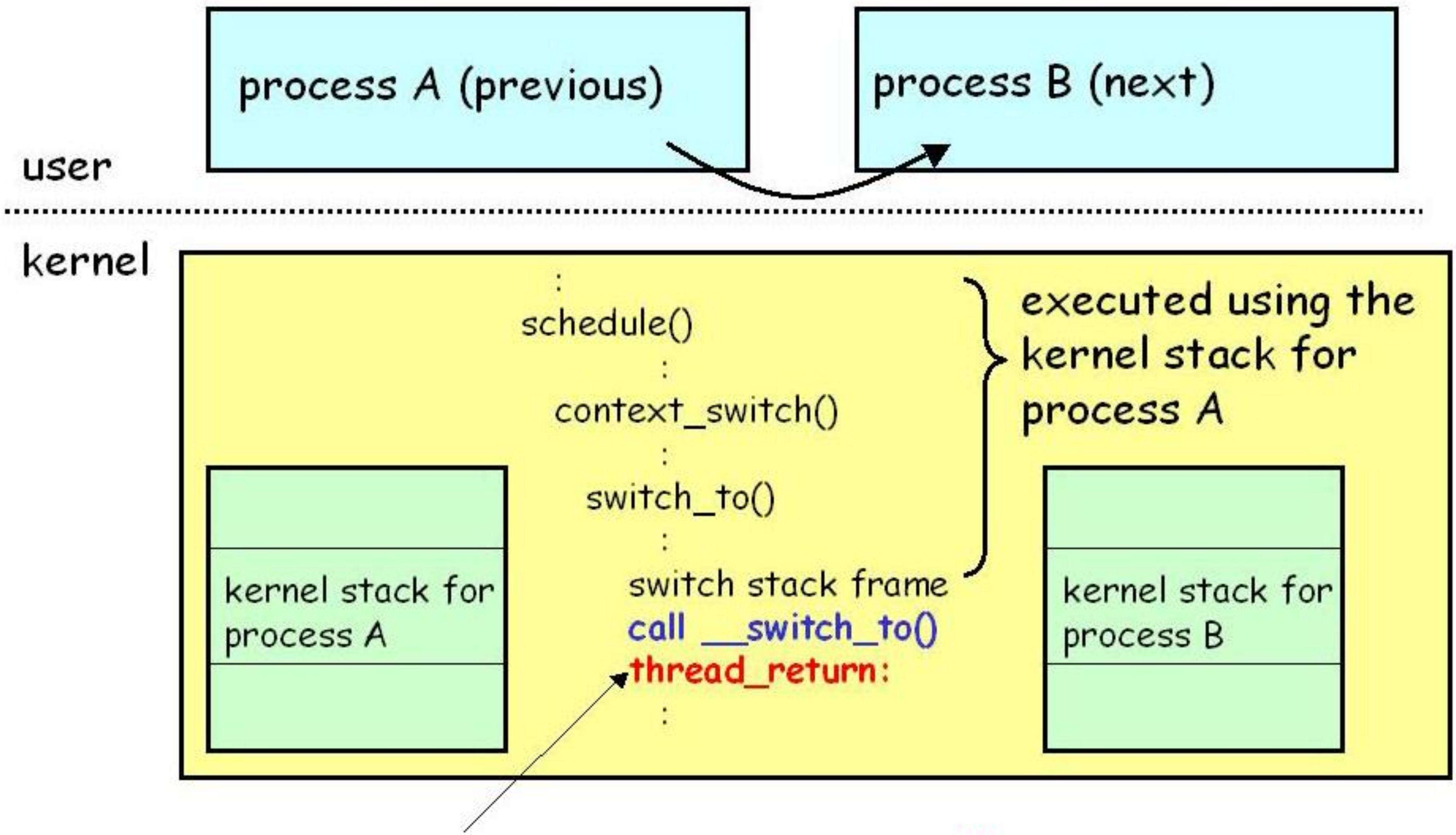


Figure 6-1: Typical memory layout of a process on Linux/x86-32

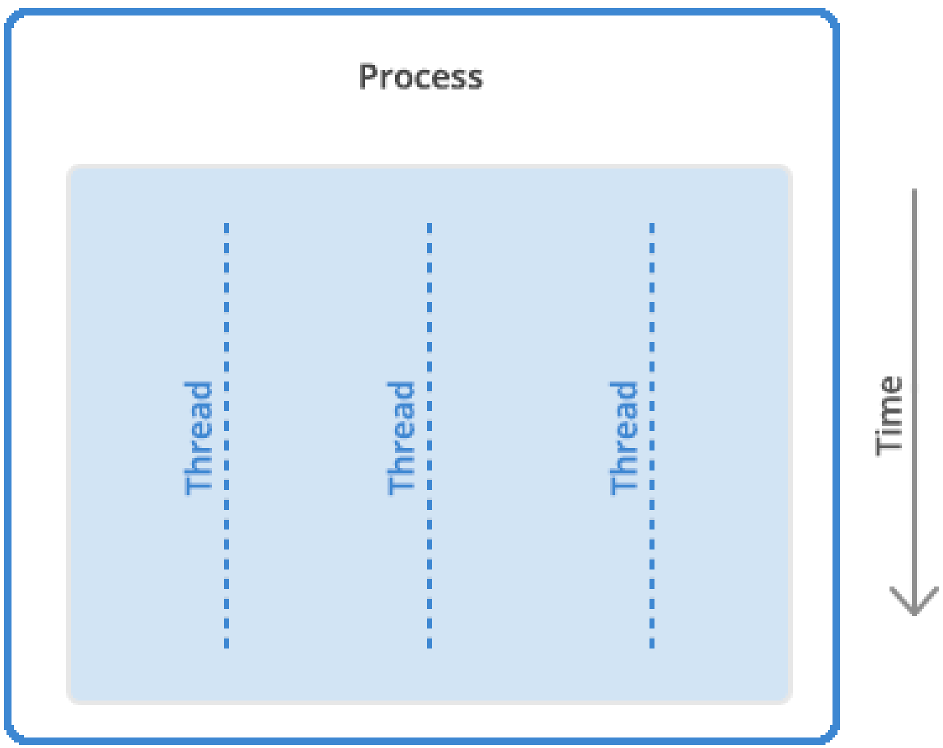
Context Switch of Linux(x86_64)

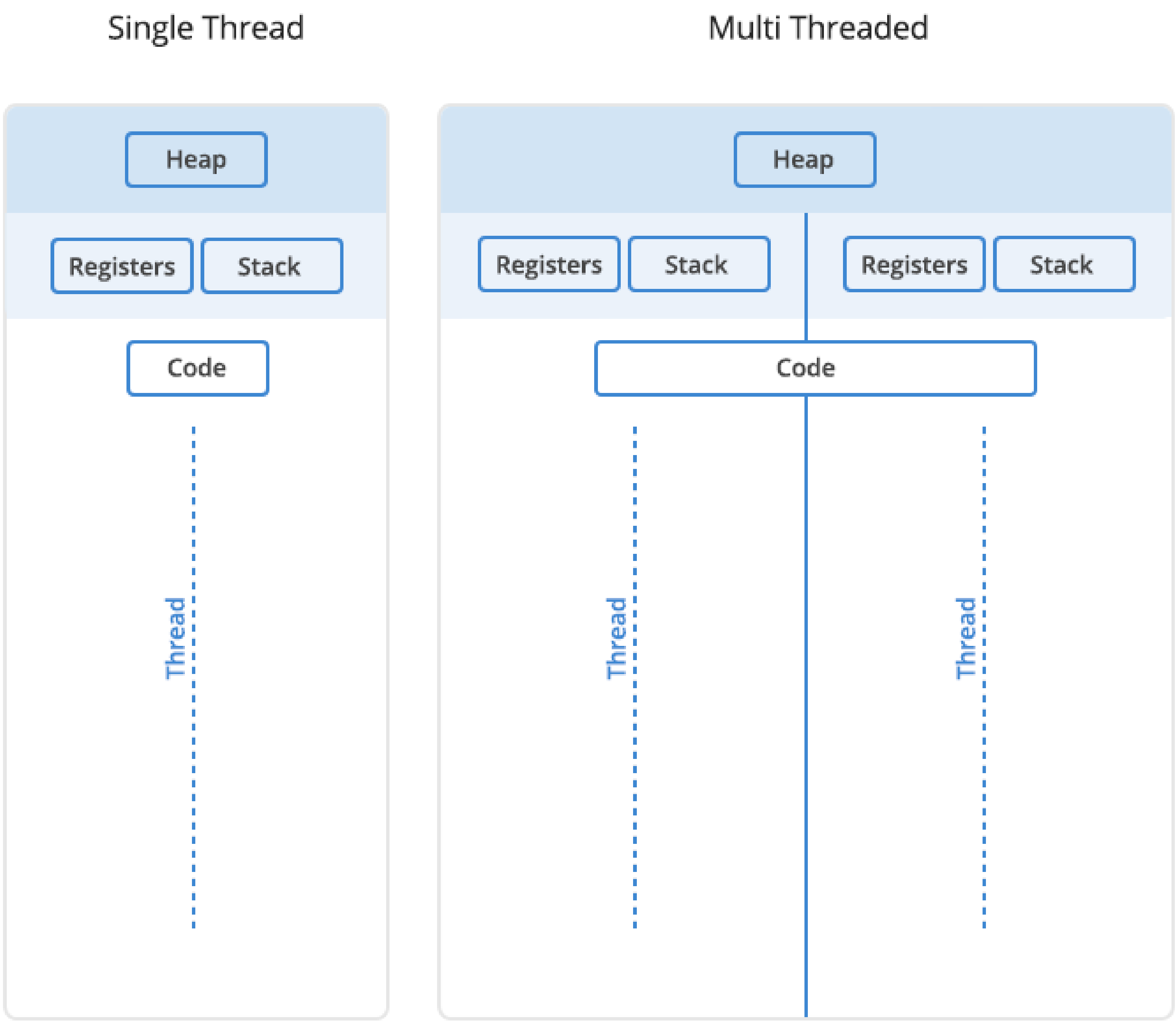


Stored as the return address by 'call __switch_to()' into the kernel stack for process B. Thus, on return of __switch_to(), process B begins to run from thread_return.

Thread (lightweight process)

- A thread is the unit of execution within a process
- spreading a single process across multiple threads



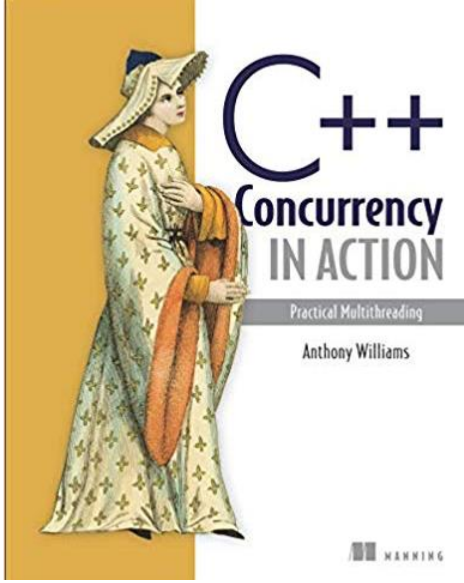

Single Thread
Multi Threaded
Heap
Registers
Stack
Code
Thread
Heap
Registers
Stack
Registers
Stack
Code
Thread
Thread

Life cycle

1. The program starts out as a text file of programming code,
2. The program is compiled or interpreted into binary form,
3. The program is loaded into memory,
4. The program becomes one or more running processes.
5. Processes are typically independent of each other,
6. While threads exist as the subset of a process.
    1. Threads can communicate with each other more easily than processes can,
    2. But threads are more vulnerable to problems caused by other threads in the same process.

PROCESS	THREAD
Processes are heavyweight operations	Threads are lighter weight operations
Each process has its own memory space	Threads use the memory of the process they belong to
Inter-process communication is slow as processes have different memory addresses	Inter-thread communication can be faster than inter-process communication because threads of the same process share memory with the process they belong to
Context switching between processes is more expensive	Context switching between threads of the same process is less expensive
Processes don’t share memory with other processes	Threads share memory with other threads of the same process

Hello concurrent world!




```
#include <iostream>
#include <thread>                    ❶

void hello()                                ❷
{
    std::cout<<"Hello Concurrent World\n";
}

int main()
{
    std::thread t(hello);            ❸
    t.join();                               ❹
}
```

Ways of invoking a thread

```
void do_some_work();
std::thread my_thread(do_some_work);
```

```
std::thread my_thread([](
    do_something();
    do_something_else();
});
```

```
class X
{
public:
    void do_lengthy_work();
};

X my_x;
std::thread t(&X::do_lengthy_work,&my_x);
```

```
class background_task
{
public:
    void operator()() const
    {
        do_something();
        do_something_else();
    }
};
background_task f;
std::thread my_thread(f);
```

Std::bind

```
#include <random>
#include <iostream>
#include <memory>
#include <functional>

void f(int n1, int n2, int n3, const int& n4, int n5)
{
    std::cout << n1 << ' ' << n2 << ' ' << n3 << ' ' << n4 << ' ' << n5 << '\n';
}

int g(int n1)
{
    return n1;
}

struct Foo {
    void print_sum(int n1, int n2)
    {
        std::cout << n1+n2 << '\n';
    }
    int data = 10;
};
```

The arguments to bind are copied or moved, and are never passed by reference unless wrapped in [std::ref](https://en.cppreference.com/w/cpp/utility/functional/ref) or [std::cref](https://en.cppreference.com/w/cpp/utility/functional/ref).

```cpp
int main()
{
    using namespace std::placeholders;  // for _1, _2, _3...

    // demonstrates argument reordering and pass-by-reference
    int n = 7;
    // (_1 and _2 are from std::placeholders, and represent future
    // arguments that will be passed to f1)
    auto f1 = std::bind(f, _2, 42, _1, std::cref(n), n);
    n = 10;
    f1(1, 2, 1001); // 1 is bound by _1, 2 is bound by _2, 1001 is unused
                    // makes a call to f(2, 42, 1, n, 7)

    // nested bind subexpressions share the placeholders
    auto f2 = std::bind(f, _3, std::bind(g, _3), _3, 4, 5);
    f2(10, 11, 12); // makes a call to f(12, g(12), 12, 4, 5);

    // common use case: binding a RNG with a distribution
    std::default_random_engine e;
    std::uniform_int_distribution<> d(0, 10);
    auto rnd = std::bind(d, e); // a copy of e is stored in rnd
    for(int n=0; n<10; ++n)
        std::cout << rnd() << ' ';
    std::cout << '\n';

    // bind to a pointer to member function
    Foo foo;
    auto f3 = std::bind(&Foo::print_sum, &foo, 95, _1);
    f3(5);

    // bind to a pointer to data member
    auto f4 = std::bind(&Foo::data, _1);
    std::cout << f4(foo) << '\n';

    // smart pointers can be used to call members of the referenced objects, too
    std::cout << f4(std::make_shared<Foo>(foo)) << '\n'
              << f4(std::make_unique<Foo>(foo)) << '\n';
}
```

Join or detach a thread

- Join
  - Wait for the thread to complete
  - cleans up any storage associated with the thread
- Detach
  - leaves the thread to run in the background, with no direct means of communicating with it.
  - It’s no longer possible to wait for that thread to complete
  - The thread can no longer be joined.

Thread guard (auto joining at destruction)

- local objects are destroyed in reverse order of construction.
- thread_guard object g is destroyed first, and the thread is joined with in the destructor

```
class thread_guard
{
    std::thread& t;
public:
    explicit thread_guard(std::thread& t_):
        t(t_)
    {}
    ~thread_guard()
    {
        if(t.joinable())
        {
            t.join();
        }
    }
    thread_guard(thread_guard const&)=delete;
    thread_guard& operator=(thread_guard const&)=delete;
};

struct func;

void f()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread t(my_func);
    thread_guard g(t);

    do_something_in_current_thread();
}
```

See definition in listing 2.1

Passing parameter into a thread can be tricky

```
struct func
{
    int& i;

    func(int& i_):i(i_){}

    void operator()()
    {
        for(unsigned j=0;j<1000000;++j)
        {
            do_something(i);
        }
    }
};

void oops()
{
    int some_local_state=0;
    func my_func(some_local_state);
    std::thread my_thread(my_func);
    my_thread.detach();
}
```

1 Potential access to dangling reference

```
void f(int i,std::string const& s);

void not_oops(int some_param)
{
    char buffer[1024];
    sprintf(buffer,"%i",some_param);
    std::thread t(f,3,std::string(buffer));
    t.detach();
}
```

by default the arguments are *copied* into internal storage

Use std::ref or cref to pass by reference

```
void update_data_for_widget(widget_id w,widget_data& data);      1

void oops_again(widget_id w)
{
    widget_data data;
    std::thread t(update_data_for_widget,w,data);      2
    display_status();
    t.join();
    process_widget_data(data);      3
}
```

```
std::thread t(update_data_for_widget,w,std::ref(data));
```

Thread is only movable not copyable

```
class scoped_thread
{
    std::thread t;
public:
    explicit scoped_thread(std::thread t_):                    1
        t(std::move(t_))
    {
        if(!t.joinable())                                      2
            throw std::logic_error("No thread");
    }
    ~scoped_thread()
    {
        t.join();                  3
    }
    scoped_thread(scoped_thread const&)=delete;
    scoped_thread& operator=(scoped_thread const&)=delete;
};

struct func;                              See
                                          listing 2.1
void f()
{
    int some_local_state;
    scoped_thread t(std::thread(func(some_local_state)));      4

    do_something_in_current_thread();
}                                                 5
```

Hardware concurrency

```
unsigned long const hardware_threads=
    std::thread::hardware_concurrency();
```

```
std::thread::id master_thread;
void some_core_part_of_algorithm()
{
    if(std::this_thread::get_id()==master_thread)
    {
        do_master_thread_work();
    }
    do_common_work();
}
```

Sharing data between threads

- *Care-free data/function call*
  - *read-only, there’s no problem, because the data read by one thread is unaffected by whether or not another thread is reading the same data.*
  - *invariants*—statements that are always true about a particular data structure, such as “this variable contains the number of items in the list”
- Data race
  - the outcome depends on the relative ordering of execution of operations on two or more threads
  - cause the dreaded *undefined behavior*
  - can be benign or *problematic*

Two solutions for data race

- Finest granularity
  - modify the design of your data structure and its invariants so that modifications are done as a series of indivisible changes, each of which preserves the invariants. This is generally referred to as *lock-free programming*
- Protect critical sections
  - wrap your data structure with a protection mechanism, to ensure that only the thread actually performing a modification can see the intermediate states where the invariants are broken

Data race

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;


void add_to_list(int new_value)
{

    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{

    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

Mutex and unique_lock

Critical section; access shared data / data race
(define by the life cycle of a lock)
Mutex (door to the critical section)
Lock (own the mutex)

```
#include <list>
#include <mutex>
#include <algorithm>

std::list<int> some_list;
std::mutex some_mutex;

void add_to_list(int new_value)
{
    std::lock_guard<std::mutex> guard(some_mutex);      3
    some_list.push_back(new_value);
}
bool list_contains(int value_to_find)
{                                                            4
    std::lock_guard<std::mutex> guard(some_mutex);
    return std::find(some_list.begin(),some_list.end(),value_to_find)
        != some_list.end();
}
```

AVOID NESTED LOCKS

don't wait for another thread if there's a chance it's waiting for you

Jailbreak

```
class some_data
{
    int a;
    std::string b;
public:
    void do_something();
};

class data_wrapper
{
private:
    some_data data;
    std::mutex m;
public:
    template<typename Function>
    void process_data(Function func)
    {
        std::lock_guard<std::mutex> l(m);
        func(data);
    }
};
```

```
some_data* unprotected;

void malicious_function(some_data& protected_data)
{
    unprotected=&protected_data;
}

data_wrapper x;

void foo()
{
    x.process_data(malicious_function);
    unprotected->do_something();
}
```

2 Pass i
malici
functi

Thread-safe data structure

Thread A	Thread B
`if(!s.empty())`	
	`if(!s.empty())`
`    int const value=s.top();`	
	`    int const value=s.top();`
`    s.pop();`	
`    do_something(value);`	`    s.pop();`
	`    do_something(value);`

A thread-safe stack

```
#include <exception>
#include <memory>                                  For std::shared_ptr< >

struct empty_stack: std::exception
{
    const char* what() const throw();
};

template<typename T>
class threadsafe_stack                                   Assignment
{                                                        operator is
public:                                                  deleted 1
    threadsafe_stack();
    threadsafe_stack(const threadsafe_stack&);
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value);
    std::shared_ptr<T> pop();
    void pop(T& value);
    bool empty() const;
};
```

```
private:
    std::stack<T> data;
    mutable std::mutex m;
public:
    threadsafe_stack(){}
    threadsafe_stack(const threadsafe_stack& other)
    {
        std::lock_guard<std::mutex> lock(other.m);
        data=other.data;
    }
    threadsafe_stack& operator=(const threadsafe_stack&) = delete;

    void push(T new_value)
    {
        std::lock_guard<std::mutex> lock(m);
        data.push(new_value);
    }
    std::shared_ptr<T> pop()
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        std::shared_ptr<T> const res(std::make_shared<T>(data.top()));
        data.pop();
        return res;
    }
    void pop(T& value)
    {
        std::lock_guard<std::mutex> lock(m);
        if(data.empty()) throw empty_stack();
        value=data.top();
        data.pop();
    }
    bool empty() const
    {
        std::lock_guard<std::mutex> lock(m);
        return data.empty();
    }
```

1 **Copy performed in constructor body**

Check for empty before trying to pop value

Allocate return value before modifying stack

Deadlock

- This is almost the opposite of a race condition: rather than two threads racing to be first, each one is waiting for the other, so neither makes any progress
- The common advice for avoiding deadlock is to always lock the two mutexes in the same order: if you always lock mutex A before mutex B, then you’ll never deadlock.

Avoid deadlock

std::defer_lock, std::try_to_lock, std::adopt_lock

Defined in header <mutex>

`constexpr std::defer_lock_t defer_lock {};`	(since C++11) (until C++17)
`inline constexpr std::defer_lock_t defer_lock {};`	(since C++17)
`constexpr std::try_to_lock_t try_to_lock {};`	(since C++11) (until C++17)
`inline constexpr std::try_to_lock_t try_to_lock {};`	(since C++17)
`constexpr std::adopt_lock_t adopt_lock {};`	(since C++11) (until C++17)
`inline constexpr std::adopt_lock_t adopt_lock {};`	(since C++17)

```
class some_big_object;
void swap(some_big_object& lhs,some_big_object& rhs);

class X
{
private:
    some_big_object some_detail;
    std::mutex m;
public:
    X(some_big_object const& sd):some_detail(sd){}

    friend void swap(X& lhs, X& rhs)
    {
        if(&lhs==&rhs)
            return;
        std::lock(lhs.m,rhs.m);                                    1
        std::lock_guard<std::mutex> lock_a(lhs.m,std::adopt_lock);   2
        std::lock_guard<std::mutex> lock_b(rhs.m,std::adopt_lock);   3
        swap(lhs.some_detail,rhs.some_detail);
    }
};
```

Alternative

```
void transfer(bank_account &from, bank_account &to, int amount)
{
    // lock both mutexes without deadlock
    std::lock(from.m, to.m);
    // make sure both already-locked mutexes are unlocked at the end of scope
    std::lock_guard<std::mutex> lock1(from.m, std::adopt_lock);
    std::lock_guard<std::mutex> lock2(to.m, std::adopt_lock);

// equivalent approach:
//    std::unique_lock<std::mutex> lock1(from.m, std::defer_lock);
//    std::unique_lock<std::mutex> lock2(to.m, std::defer_lock);
//    std::lock(lock1, lock2);

    from.balance -= amount;
    to.balance += amount;
}
```

Shared mutex for read only data

- Std::shared_mutex + std::shared_lock

```
class ThreadSafeCounter {
 public:
  ThreadSafeCounter() = default;

  // Multiple threads/readers can read the counter's value at the same time.
  unsigned int get() const {
    std::shared_lock lock(mutex_);
    return value_;
  }

  // Only one thread/writer can increment/write the counter's value.
  void increment() {
    std::unique_lock lock(mutex_);
    value_++;
  }

  // Only one thread/writer can reset/write the counter's value.
  void reset() {
    std::unique_lock lock(mutex_);
    value_ = 0;
  }

 private:
  mutable std::shared_mutex mutex_;
  unsigned int value_ = 0;
};
```

synchronization

- Conditional variable
- block a thread until another thread both modifies a shared variable (the *condition*), and notifies the condition_ variable

std::condition_variable

Member functions

condition_variable::condition_variable
condition_variable::~condition_variable

Notification

condition_variable::notify_one
condition_variable::notify_all

Waiting

condition_variable::wait
condition_variable::wait_for
condition_variable::wait_until

Native handle

condition_variable::native_handle

```cpp
#include <iostream>
#include <condition_variable>
#include <thread>
#include <chrono>

std::condition_variable_any cv;
std::mutex cv_m;
int i = 0;
bool done = false;

void waits()
{
    std::unique_lock<std::mutex> lk(cv_m);
    std::cout << "Waiting... \n";
    cv.wait(lk, [](){return i == 1;});
    std::cout << "...finished waiting. i == 1\n";
    done = true;
}

void signals()
{
    std::this_thread::sleep_for(std::chrono::seconds(1));
    std::cout << "Notifying...\n";
    cv.notify_one();

    std::unique_lock<std::mutex> lk(cv_m);
    i = 1;
    while (!done) {
        lk.unlock();
        std::this_thread::sleep_for(std::chrono::seconds(1));
        lk.lock();
        std::cerr << "Notifying again...\n";
        cv.notify_one();
    }
}

int main()
{
    std::thread t1(waits), t2(signals);
    t1.join(); t2.join();
}
```

Condition_variable

The thread that intends to modify the variable has to

1. acquire a `std::mutex` (typically via `std::lock_guard`)
2. perform the modification while the lock is held
3. execute `notify_one` or `notify_all` on the `std::condition_variable` (the lock does not need to be held for notification)

Even if the shared variable is atomic, it must be modified under the mutex in order to correctly publish the modification to the waiting thread.

Any thread that intends to wait on `std::condition_variable` has to

1. acquire a `std::unique_lock<std::mutex>`, on the same mutex as used to protect the shared variable
2. execute `wait`, `wait_for`, or `wait_until`. The wait operations atomically release the mutex and suspend the execution of the thread.
3. When the condition variable is notified, a timeout expires, or a spurious wakeup occurs, the thread is awakened, and the mutex is atomically reacquired. The thread should then check the condition and resume waiting if the wake up was spurious.

Spurious wakeup

- Is a hardware related issue

According to David R. Butenhof's Programming with POSIX Threads ISBN 0-201-63392-2:

> "This means that when you wait on a condition variable, the wait may (occasionally) return when no thread specifically broadcast or signaled that condition variable. Spurious wakeups may sound strange, but on some multiprocessor systems, making condition wakeup completely predictable might substantially slow all condition variable operations. The race conditions that cause spurious wakeups should be considered rare."

`wait` causes the current thread to block until the condition variable is notified or a spurious wakeup occurs, optionally looping until some predicate is satisfied.

1) Atomically unlocks `lock`, blocks the current executing thread, and adds it to the list of threads waiting on `*this`. The thread will be unblocked when `notify_all()` or `notify_one()` is executed. It may also be unblocked spuriously. When unblocked, regardless of the reason, `lock` is reacquired and `wait` exits. If this function exits via exception, `lock` is also reacquired. (until C++14)

2) Equivalent to

```
while (!pred()) {
    wait(lock);
}
```

this_thread namespace

get_id (C++11) sleep_for (C++11)
yield (C++11) sleep_until (C++11)

This_thread::sleep and yield

- They both causes the current thread to block
- sleep (until a condition)
- Yield: busy sleep
- Yield is better for fast turn-over model but waste CPU

```
void little_sleep(std::chrono::microseconds us)
{
    auto start = std::chrono::high_resolution_clock::now();
    auto end = start + us;
    do {
        std::this_thread::yield();
    } while (std::chrono::high_resolution_clock::now() < end);
}
```

Package task/promise/future

```cpp
#include <iostream>
#include <future>
#include <thread>

int main()
{
    // future from a packaged_task
    std::packaged_task<int()> task([](){ return 7; }); // wrap the function
    std::future<int> f1 = task.get_future();  // get a future
    std::thread(std::move(task)).detach(); // launch on a thread

    // future from an async()
    std::future<int> f2 = std::async(std::launch::async, [](){ return 8; });

    // future from a promise
    std::promise<int> p;
    std::future<int> f3 = p.get_future();
    std::thread( [](std::promise<int>& p){ p.set_value(9); },
                 std::ref(p) ).detach();

    std::cout << "Waiting..." << std::flush;
    f1.wait();
    f2.wait();
    f3.wait();
    std::cout << "Done!\nResults are: "
              << f1.get() << ' ' << f2.get() << ' ' << f3.get() << '\n';
}
```

async (lazy threading)

```
#include <iostream>
#include <vector>
#include <algorithm>
#include <numeric>
#include <future>

template <typename RAIter>
int parallel_sum(RAIter beg, RAIter end)
{
    typename RAIter::difference_type len = end-beg;
    if(len < 1000)
        return std::accumulate(beg, end, 0);

    RAIter mid = beg + len/2;
    auto handle = std::async(std::launch::async,
                             parallel_sum<RAIter>, mid, end);
    int sum = parallel_sum(beg, mid);
    return sum + handle.get();
}

int main()
{
    std::vector<int> v(10000, 1);
    std::cout << "The sum is " << parallel_sum(v.begin(), v.end()) << '\n';
}
```

Jthread (C++20)

- The class jthread represents [a single thread of execution](). It has the same general behavior as [std::thread](), except that jthread automatically rejoins on destruction, and can be cancelled/stopped in certain situations.

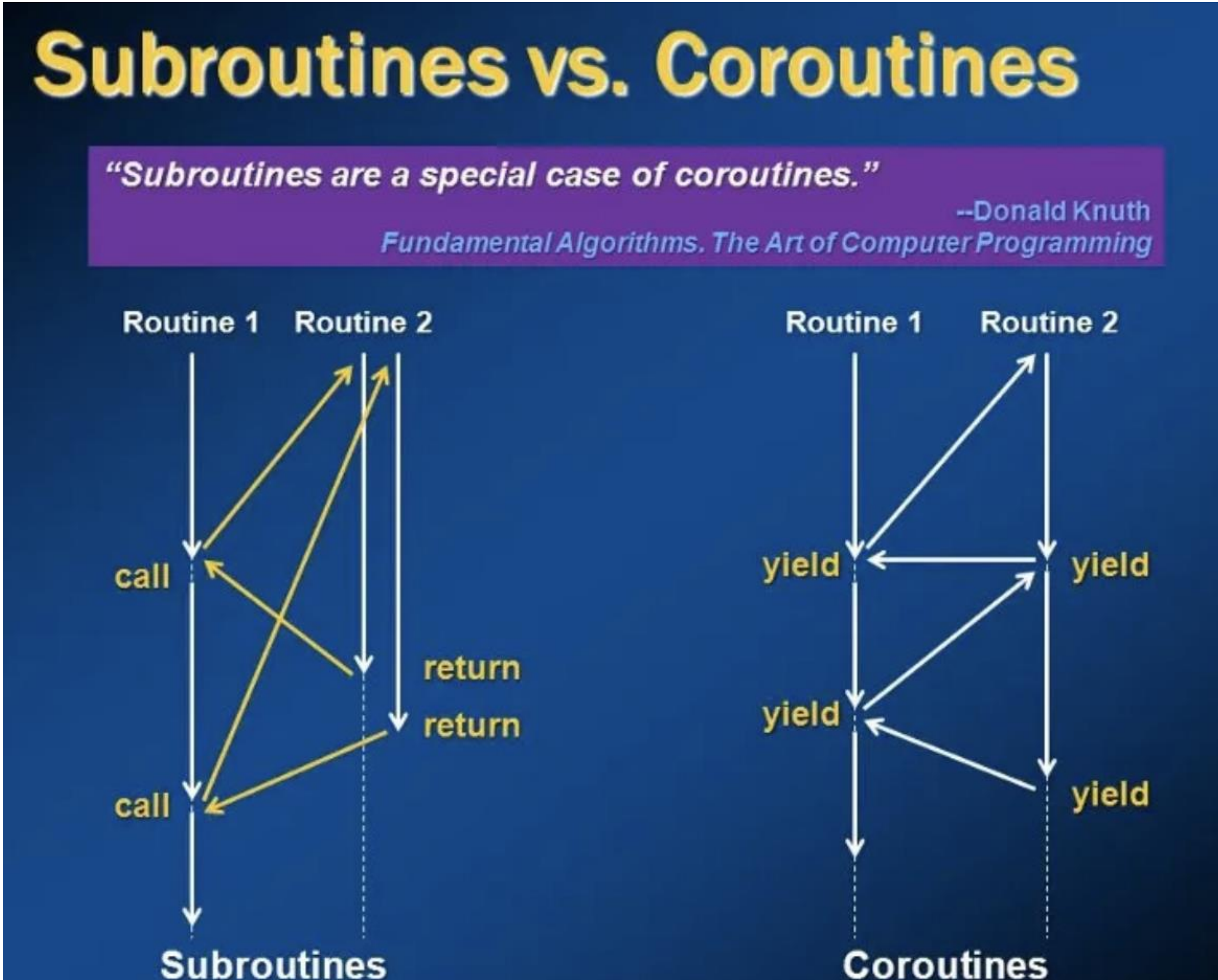
```cpp
#include "jthread.hpp"
#include <chrono>
#include <iostream>

void sleep(const int seconds)
{
    std::this_thread::sleep_for(std::chrono::seconds(seconds));
}

int main()
{
    std::jthread jt{ [] (std::stop_token st) {
        while (!st.requested_stop()) {
            std::cout << "Doing work\n";
            sleep(1);
        }
    }};
    sleep(5);
    jt.request_stop();
    jt.join();
}
```

Coroutines: lightweight threads

- **Coroutines** are stackless functions designed for enabling co-operative Multitasking, by allowing execution to be suspended and resumed.
  - **Cooperative Multitasking—**If multitasking participant process or thread voluntarily let go of control periodically or when idle or logically blocked.
- **Coroutines** suspend execution by returning to the caller and the data that is required to resume execution is stored separately from the stack. This allows for sequential code that executes asynchronously
  - well-suited for implementing familiar program components such as cooperative tasks, exceptions, event loops, state machines, and pipes.


Subroutines vs. Coroutines
"Subroutines are a special case of coroutines."
--Donald Knuth
Fundamental Algorithms. The Art of Computer Programming
Routine 1
Routine 2
call
return
return
call
Subroutines
Routine 1
Routine 2
yield
yield
yield
yield
Coroutines

Coroutines vs Threads

- **Coroutines provide concurrency but not parallelism**
- Switching between coroutines need not involve any system/blocking calls so no need for synchronization primitives such as mutexes, semaphores.
- Coroutines provide asynchronicity and resource locking isn’t needed.

Coroutines in C++20

- A function is a coroutine if its definition does any of the following:
  - uses the co_await operator to suspend execution until resumed.
  - uses the keyword co_yield to suspend execution returning a value.
  - uses the keyword co_return to complete execution. Let’s take a similar example to get a range.

```cpp
#include <iostream>

#include <vector> // Coroutine gets called on need

generator <int> generateNumbers(int begin, int inc = 1) {

  for (int i = begin;; i += inc) {
    co_yield i;
  }

}

int main() {

  std::cout << std::endl;

  const auto numbers = generateNumbers(-10);

  for (int i = 1; i<= 20; ++i)
    std::cout << numbers << " "; // Runs finite = 20 times***

  for (auto n:generateNumbers(0, 5)) // Runs infinite times**
    std::cout << n << " "; // (3)

  std::cout << "\n\n";

}
```

CppCoro - A coroutine library for C++

The 'cppcoro' library provides a large set of general-purpose primitives for m
proposal described in N4680.

These include:

- Coroutine Types
  - `task<T>`
  - `shared_task<T>`
  - `generator<T>`
  - `recursive_generator<T>`
  - `async_generator<T>`

Atomic

```
//long multiThreadedSum = 0;
std::atomic<long> multiThreadedSum( 0 );
```

```
long multiThreadedSum = 0;

void SumNumbers( const std::vector<int>& toBeSummed, int idxStart, int idxEnd )
{
    for (int i = idxStart; i <= idxEnd; ++i)
    {
        multiThreadedSum += toBeSummed[i];
    }
}
```

```
int main()
{
    volatile int val = 0;
    val += 1;
    return val;
}
```

```
main:
        mov     DWORD PTR [rsp-4], 0
        mov     eax, DWORD PTR [rsp-4]
        add     eax, 1
        mov     DWORD PTR [rsp-4], eax
        mov     eax, DWORD PTR [rsp-4]
        ret
```

Memory order

- Relaxed ordering
- Release-Acquire ordering
- Release-Consume ordering
- Sequentially-consistent ordering

```
           int x = 0;     // global variable
           int y = 0;     // global variable

Thread-1:                  Thread-2:
x = 100;                   while (y != 200)
y = 200;                       ;
                           std::cout << x;
```

```
Thread-1:
y = 200;
x = 100;
```

```
           int x = 0;     // global variable

Thread-1:                          Thread-2:
x = 100;    // A                   std::cout << x;    // B
```

Relaxed ordering

- No control at all

```
void f()
{
    for (int n = 0; n < 1000; ++n) {
        cnt.fetch_add(1, std::memory_order_relaxed);
    }
}
```

Atomic operations library

The atomic library provides components for fine-grained programming. Each atomic operation is indivisible with r object. Atomic objects are free of data races.

Defined in header <atomic>

Atomic types

atomic	(C++11)
atomic_ref	(C++20)

Operations on atomic types

atomic_is_lock_free	(C++11)
atomic_store **atomic_store_explicit**	(C++11) (C++11)
atomic_load **atomic_load_explicit**	(C++11) (C++11)
atomic_exchange **atomic_exchange_explicit**	(C++11) (C++11)
atomic_compare_exchange_weak **atomic_compare_exchange_weak_explicit** **atomic_compare_exchange_strong** **atomic_compare_exchange_strong_explicit**	(C++11) (C++11) (C++11) (C++11)
atomic_fetch_add **atomic_fetch_add_explicit**	(C++11) (C++11)
atomic_fetch_sub **atomic_fetch_sub_explicit**	(C++11) (C++11)
atomic_fetch_and **atomic_fetch_and_explicit**	(C++11) (C++11)
atomic_fetch_or **atomic_fetch_or_explicit**	(C++11) (C++11)

Release-Acquire ordering

`memory_order_acquire`	A load operation with this memory order performs the *acquire operation* on the affected memory location: no reads or writes in the current thread can be reordered before this load. All writes in other threads that release the same atomic variable are visible in the current thread (see Release-Acquire ordering below)
`memory_order_release`	A store operation with this memory order performs the *release operation*: no reads or writes in the current thread can be reordered after this store. All writes in the current thread are visible in other threads that acquire the same atomic variable (see Release-Acquire ordering below) and writes that carry a dependency into the atomic variable become visible in other threads that consume the same atomic (see Release-Consume ordering below).

```
std::atomic<bool> ready{ false };
int data = 0;

void producer()
{
    data = 100;                                        // A
    ready.store(true, std::memory_order_release);      // B
}

void consumer()
{
    while (!ready.load(std::memory_order_acquire))     // C
        ;
    assert(data == 100); // never failed               // D
}
```