

Operações Atômicas e Concorrentes

Threads e Concorrência em C++

Hervé Yviquel

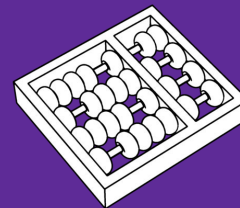
hyviquel@unicamp.br

Universidade Estadual de Campinas (Unicamp)

Instituto de Computação (IC)

Laboratório de Sistemas de Computação (LSC)

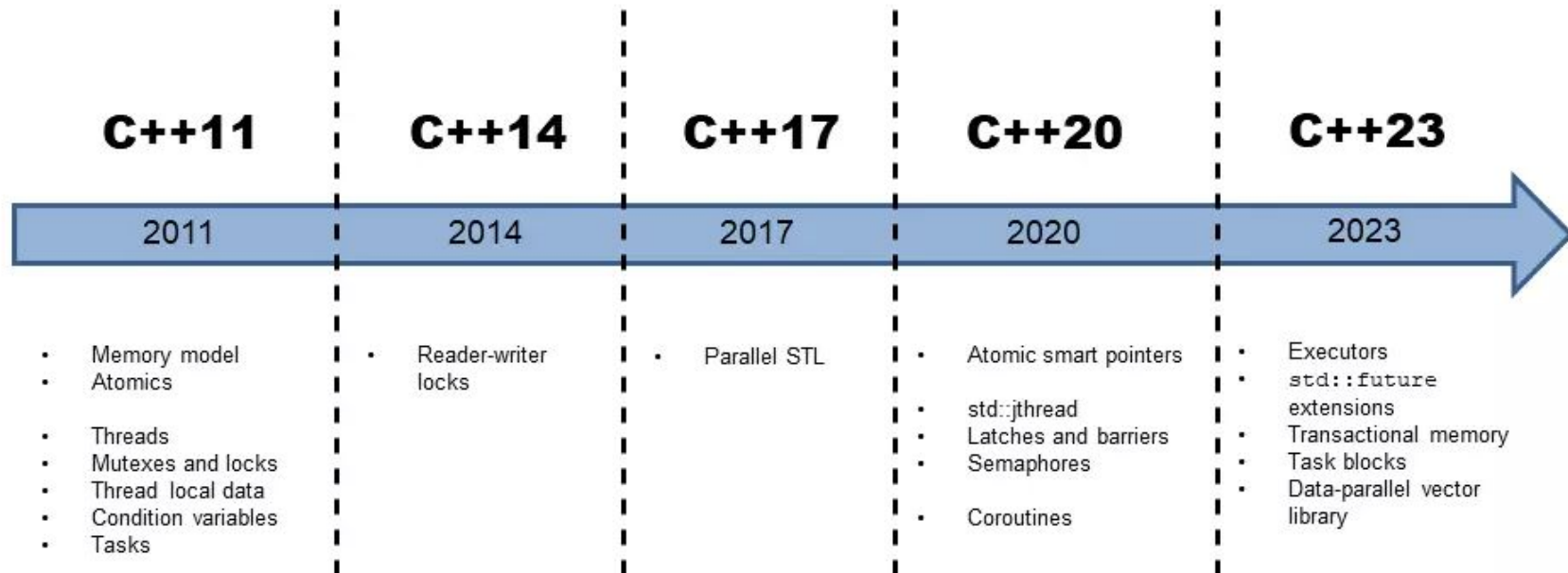
Programação em C++ • Out-Nov 2023



UNICAMP

Plano

- Estruturas de dados *threadsafe*
- Operações Atômicas
- Operações Assíncronas
- Future/Promise
- Tasks
- Async



Estruturas de dados *threadsafe*



```
#include <exception>
struct empty_stack: std::exception
{
    const char* what() const throw();
};
template<typename T>
class threadsafe_stack

{
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(std::move(new_value));    ← ❶
    }
}
```

```
std::shared_ptr<T> pop()
{
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();    ← 2
    std::shared_ptr<T> const res(
        std::make_shared<T>(std::move(data.top())));    ← 3
    data.pop();    ← 4
    return res;
}

void pop(T& value)
{
    std::lock_guard<std::mutex> lock(m);
    if(data.empty()) throw empty_stack();
    value=std::move(data.top());    ← 5
    data.pop();    ← 6
}

bool empty() const
{
    std::lock_guard<std::mutex> lock(m);
    return data.empty();
}

};
```

```
template<typename T>
class threadsafe_queue
{
private:
    mutable std::mutex mut;
    std::queue<T> data_queue;
    std::condition_variable data_cond;
public:
    threadsafe_queue()
    {}
    void push(T new_value)
    {
        std::lock_guard<std::mutex> lk(mut);
        data_queue.push(std::move(new_value));
        data_cond.notify_one();      ← 1
    }
    void wait_and_pop(T& value)      ← 2
    {
        std::unique_lock<std::mutex> lk(mut);
        data_cond.wait(lk, [this]{return !data_queue.empty();});
        value=std::move(data_queue.front());
        data_queue.pop();
    }
}
```

```
std::shared_ptr<T> wait_and_pop()    ← 3
{
    std::unique_lock<std::mutex> lk(mut);
    data_cond.wait(lk, [this]{return !data_queue.empty();});    ← 4
    std::shared_ptr<T> res(
        std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
}
bool try_pop(T& value)
{
    std::lock_guard<std::mutex> lk(mut);
    if(data_queue.empty())
        return false;
    value=std::move(data_queue.front());
    data_queue.pop();
    return true;
}
```



```
std::shared_ptr<T> try_pop()
{
    std::lock_guard<std::mutex> lk(mut);
    if (data_queue.empty())
        return std::shared_ptr<T>();    ← 5
    std::shared_ptr<T> res(
        std::make_shared<T>(std::move(data_queue.front())));
    data_queue.pop();
    return res;
}
bool empty() const
{
    std::lock_guard<std::mutex> lk(mut);
    return data_queue.empty();
}
};
```

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	0.213	0.123	0.098	0.115
One Mutex for Entire List	0.211	0.450	0.385	0.457
One Mutex per Node	1.680	5.700	3.450	2.700

100,000 ops/thread

99.9% Member

0.05% Insert

0.05% Delete

Implementation	Number of Threads			
	1	2	4	8
Read-Write Locks	2.48	4.97	4.69	4.71
One Mutex for Entire List	2.50	5.13	5.04	5.11
One Mutex per Node	12.00	29.60	17.00	12.00

100,000 ops/thread

80% Member

10% Insert

10% Delete

Pior que serial !! Para cada nó, uma chamada para lock outra para unlock

Operações Atômicas



- A exclusão mútua pode ser ineficiente para sincronização
 - Sincronização muito grosseira
 - Pode exigir comunicação com o sistema operacional
- O hardware moderno também suporta operações atômicas para sincronização
 - A ordem de memória de uma CPU determina como as operações de memória não atômicas podem ser reordenadas
 - Em C++, todas as operações conflitantes não atômicas têm comportamento indefinido, mesmo que a ordem de memória da CPU permita!
 - Há uma exceção: Funções atômicas especiais são permitidas para ter conflitos
 - O compilador geralmente conhece sua CPU e gera instruções atômicas “reais” apenas se necessário

- C++ fornece operações atômicas na biblioteca de operações atômicas
 - Implementado no cabeçalho `<atomic>`
 - `std::atomic<T>` é uma classe que representa uma versão atômica do tipo `T`
 - Pode ser usado (quase) de forma intercambiável com o tipo original `T`
 - Tem o mesmo tamanho e alinhamento que o tipo original `T`
 - Operações conflitantes são permitidas apenas em objetos `std::atomic<T>`
- `std::atomic` por si só não fornece nenhuma sincronização
 - Simplesmente torna as operações conflitantes possíveis e com comportamento definido
 - Expõe as garantias de modelos de memória específicos para o programador
 - Modelos de programação adequados devem ser usados para alcançar a sincronização adequada

- `std::atomic` possui várias funções membro que implementam operações atômicas
 - `T load()`: Carrega o valor
 - `void store(T desired)`: Armazena "desired" no objeto
 - `T exchange(T desired)`: Armazena "desired" no objeto e retorna o valor antigo
- Se `T` for um tipo integral (derivado de `integer`), as seguintes operações também existem
 - `T fetch_add(T arg)`: Adiciona "arg" ao valor e retorna o valor antigo
 - `T fetch_sub(T arg)`: O mesmo para subtração
 - `T fetch_and(T arg)`: O mesmo para a operação "and" bit a bit
 - `T fetch_or(T arg)`: O mesmo para a operação "or" bit a bit
 - `T fetch_xor(T arg)`: O mesmo para a operação "xor" bit a bit

```
#include <thread>

int main() {
    unsigned value = 0;
    thread t([]() {
        for (size_t i = 0; i < 10; ++i)
            ++value; // UNDEFINED BEHAVIOR, data race
    });

    for (size_t i = 0; i < 10; ++i)
        ++value; // UNDEFINED BEHAVIOR, data race

    t.join();

    // value will contain garbage
}
```



```
#include <atomic>
#include <thread>

int main() {
    std::atomic<unsigned> value = 0;
    thread t([]() {
        for (size_t i = 0; i < 10; ++i)
            value.fetch_add(1); // OK, atomic increment
    });

    for (size_t i = 0; i < 10; ++i)
        value.fetch_add(1); // OK, atomic increment

    t.join();

    // value will contain 20
}
```

- C++ pode suportar operações atômicas que não são suportadas pela CPU
 - `std::atomic<T>` pode ser usado com qualquer tipo trivialmente copiável
 - Em particular, também para tipos que são muito maiores do que uma linha de cache
 - Para garantir atomicidade, os compiladores têm permissão para recorrer a mutexes
- O padrão C++ define semânticas precisas para operações atômicas
 - Cada objeto atômico tem uma ordem de modificação totalmente ordenada
 - Existem várias ordens de memória que definem como operações em diferentes objetos atômicos podem ser reordenadas
 - As ordens de memória do C++ não mapeiam necessariamente de forma precisa para as ordens de memória definidas por uma CPU

- Todas as modificações de um único objeto atômico são totalmente ordenadas
 - Isso é chamado de ordem de modificação do objeto
 - Todas as threads têm garantia de observar modificações do objeto nesta ordem
- Modificações de diferentes objetos atômicos podem ser não ordenadas
 - Diferentes threads podem observar modificações de múltiplos objetos atômicos em uma ordem diferente.
 - Os detalhes dependem da ordem da memória que é usada para as operações atômicas

```
std::atomic<int> i = 0, j = 0;
void workerThread() {
    i.fetch_add(1); // (A)
    i.fetch_sub(1); // (B)
    j.fetch_add(1); // (C)
}
void readerThread() {
    int iLocal = i.load(), jLocal = j.load();
    assert(iLocal != -1); // always true
}
```

- Observações

- Threads leitoras nunca verão uma ordem de modificação com (B) antes de (A)
- Dependendo da ordem da memória, múltiplas threads leitoras podem ver qualquer uma das sequências (A),(B),(C), ou (A),(C),(B), ou (C),(A),(B).

- A biblioteca de atômicos define várias ordens de memória
 - Todas as funções atômicas recebem uma ordem de memória como seu último parâmetro
 - As duas ordens de memória mais importantes são `std::memory_order_relaxed` e `std::memory_order_seq_cst`
 - `std::memory_order_seq_cst` é usado por padrão se nenhuma ordem de memória for fornecida explicitamente
 - Devem manter esse padrão a menos que identifique a operação atômica como um gargalo de desempenho

```
std::atomic<int> i = 0;  
  
i.fetch_add(1); // uses std::memory_order_seq_cst  
i.fetch_add(1, std::memory_order_seq_cst);  
i.fetch_add(1, std::memory_order_relaxed);
```

```
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_relaxed); // (A)
        i.fetch_sub(1, std::memory_order_relaxed); // (B)
        j.fetch_add(1, std::memory_order_relaxed); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

- Mapeia aproximadamente para uma CPU com ordem de memória fraca
- Somente ordem de modificação consistente é garantida
- Operações atômicas de objetos diferentes podem ser reordenadas arbitrariamente
- Observações:
 - threadB() pode observar (A),(B),(C).
 - threadC() pode observar (C),(A),(B).

```
std::atomic<int> i = 0, j = 0;
void threadA() {
    while (true) {
        i.fetch_add(1, std::memory_order_seq_cst); // (A)
        i.fetch_sub(1, std::memory_order_seq_cst); // (B)
        j.fetch_add(1, std::memory_order_seq_cst); // (C)
    }
}
void threadB() { /* ... */ }
void threadC() { /* ... */ }
```

- Mapeia aproximadamente para uma CPU com ordem de memória forte
- Garante que todos os threads vejam todas as operações atômicas em uma ordem globalmente consistente
- Observações:
 - threadB() pode observar (C),(A),(B).
 - threadC() então também observará (C),(A),(B).

- As operações de comparação e troca (*compare-and-swap*) são uma das operações mais úteis em atômicos
 - Assinatura: `bool compare_exchange_weak(T& expected, T desired)`
 - Compara o valor atual do atômico com o esperado
 - Substitui o valor atual por "desired" se o atômico continha o valor esperado e retorna verdadeiro
 - Atualiza o "expected" para conter o valor atual do objeto atômico e retorna falso caso contrário.
- Frequentemente é o principal componente para sincronizar estruturas de dados sem mutexes:
 - Permite-nos verificar que nenhuma modificação ocorreu em um atômico durante algum período de tempo

Inserir em uma lista ligada simples lock-free

```
#include <atomic>

class SafeList {
private:
    struct Entry {
        T value;
        Entry* next;
    };

    std::atomic<Entry*> head;

    Entry* allocateEntry(const T& value);

public:
    void insert(const T& value) {
        auto* entry = allocateEntry(value);
        auto* currentHead = head.load();
        do {
            entry->next = currentHead;
        } while (!head.compare_exchange_weak(currentHead, entry));
    }
};
```

- O `std::atomic` realmente fornece duas versões de CAS com a mesma assinatura:
 - `compare_exchange_weak` – CAS fraco
 - `compare_exchange_strong` – CAS forte
- Semântica
 - A versão fraca tem permissão para retornar falso, mesmo quando nenhum outro thread modificou o valor.
 - Isso é chamado de "falha espúria".
 - A versão forte usa um loop internamente para evitar isso.
- Regra geral
 - Se você usa uma operação CAS em um loop, sempre use a versão fraca.

- O `std::atomic` pode ser difícil de manusear
 - `std::atomic` não é movível nem copiável
 - Como consequência, ele não pode ser usado facilmente em contêineres da biblioteca padrão
- `std::atomic_ref` nos permite aplicar operações atômicas em objetos não-atômicos
 - O construtor recebe uma referência a um objeto arbitrário do tipo T
 - O objeto referenciado é tratado como um objeto atômico durante a vida útil do `std::atomic_ref`
 - `std::atomic_ref` define funções membro similares ao `std::atomic`
- Corridas de dados (data races) entre acessos através de `std::atomic_ref` e acessos não-atômicos ainda são comportamentos indefinidos!

```
#include <atomic>
#include <thread>
#include <vector>

int main() {
    std::vector<int> localCounters(4);
    std::vector<std::thread> threads;

    for (size_t i = 0; i < 16; ++i) {
        threads.emplace_back([]() {
            for (size_t j = 0; j < 100; ++j) {
                std::atomic_ref ref(localCounters[i % 4]);
                ref.fetch_add(1);
            }
        });
    }

    for (auto& thread : threads) {
        thread.join();
    }
}
```

Operações Assíncronas



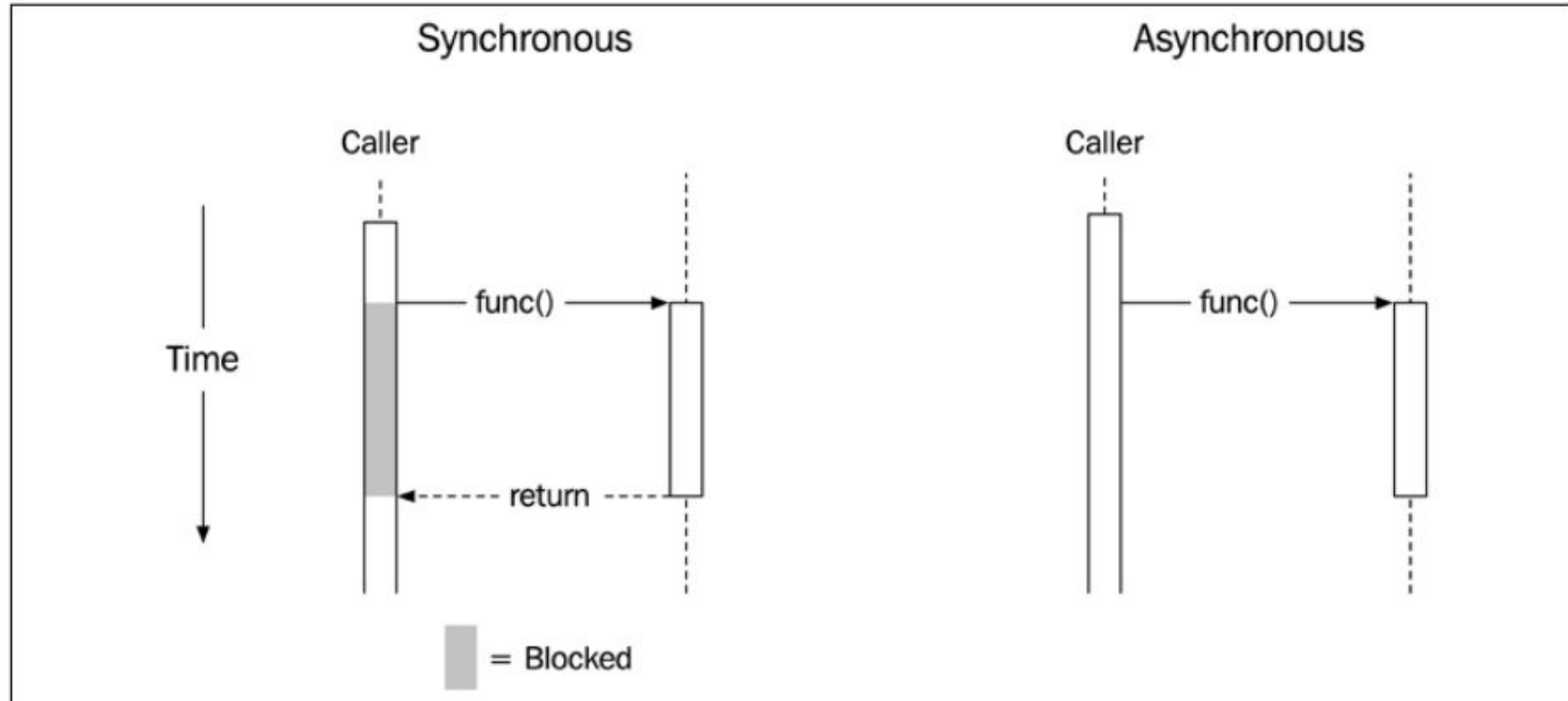


Figure 11.8: Synchronous versus asynchronous calls. The asynchronous task returns immediately but continues to work after the caller has regained control.

Promise/Future



- Um Promise é um objeto da biblioteca padrão de C++ que permite a uma thread prometer um valor ou um estado que estará disponível no futuro
- Ele é usado para passar valores entre threads ou para sinalizar a conclusão de uma tarefa

- **Passagem de Valor ou Estado**
 - Um Promise pode ser usado para armazenar um valor ou uma exceção que será recuperado posteriormente por um Future
- **Comunicação com Future**
 - Cada Promise está associado a um Future. Quando o valor é armazenado no Promise, o Future correspondente é notificado e pode acessar esse valor
- **Uso em Operações Assíncronas**
 - Promise é frequentemente usado em programação assíncrona para retornar resultados de uma thread para outra

- Um Future é um objeto que representa um valor ou estado que estará disponível em algum momento no futuro
- Future é usado para acessar o resultado de uma operação assíncrona que pode estar sendo executada em outra thread

- **Acesso ao Resultado de Tarefas Assíncronas**
 - Future permite que uma thread espere e recupere o valor fornecido por um Promise ou criado por `std::async`
- **Bloqueio até a Disponibilidade do Valor**
 - O acesso ao valor em um Future (por exemplo, usando `future.get()`) bloqueia a thread até que o valor esteja disponível
- **Uma Vez Só**
 - O valor em um Future pode ser recuperado apenas uma vez, após o qual o Future se torna vazio

```
void compute(std::promise<int>&& prom) {  
    int result = 42; // Algum cálculo ou operação  
    prom.set_value(result); // Armazena o resultado no Promise  
}  
  
int main() {  
    // Cria um Promise  
    std::promise<int> prom;  
  
    // Obtém um Future associado ao Promise  
    std::future<int> fut = prom.get_future();  
  
    // Executa a função 'compute' em uma nova thread, passando o Promise  
    std::thread t(compute, std::move(prom));  
  
    // Obtém o resultado do Future  
    int result = fut.get();  
    std::cout << "Resultado: " << result << std::endl;  
  
    // Aguarda a conclusão da thread  
    t.join();  
  
    return 0;  
}
```

- Comunicação entre Threads
 - Promise e Future são usados juntos para passar dados de forma segura entre threads.
- Tratamento de Exceções
 - Se uma exceção for lançada na thread produtora, ela pode ser passada para o Future e tratada na thread consumidora.
- Sincronização de Tarefas
 - `Future.get()` bloqueia a execução até que o resultado esteja disponível, garantindo a sincronização entre threads.
- Esses mecanismos são essenciais para a programação concorrente em C++, pois permitem a comunicação e sincronização entre threads de maneira eficiente e segura

```
auto divide(int a, int b, std::promise<int>& p) {
    if(b == 0) {
        auto e = std::runtime_error{"Divide by zero exception"};
        p.set_exception(std::make_exception_ptr(e));
    } else {
        const auto result = a / b;
        p.set_value(result);
    }
}

int main() {
    auto p = std::promise<int>{};
    std::thread(divide, 45, 5, std::ref(p)).detach();
    auto f = p.get_future();
    try {
        const auto& result = f.get(); // Blocks until ready
        std::cout << "Result: " << result << '\n';
    } catch(const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << '\n';
    }
}
```

- `std::shared_future` é uma variante do `std::future`
- **Acesso Compartilhado ao Resultado**
 - `std::shared_future` permite que várias partes do programa acessem o resultado de uma operação assíncrona
 - Isso é útil quando o mesmo resultado precisa ser usado em diferentes contextos ou threads
- **Transferência:**
 - Diferente de `std::future`, que é movido e se torna inválido após a movimentação, `std::shared_future` pode ser copiado, facilitando a passagem do mesmo resultado para várias threads

```
void worker(std::shared_future<int> fut) {  
    std::cout << "O valor é: " << fut.get() << std::endl;  
    // Pode chamar fut.get() novamente, se necessário  
}  
  
int main() {  
    std::promise<int> prom;  
    std::shared_future<int> fut = prom.get_future().share();  
  
    // Inicia várias threads, todas compartilhando o mesmo future  
    std::thread t1(worker, fut);  
    std::thread t2(worker, fut);  
  
    // Define o valor do promise  
    prom.set_value(10);  
  
    // Espera as threads terminarem  
    t1.join();  
    t2.join();  
  
    return 0;  
}
```


Task



- `std::packaged_task` é um recurso da biblioteca padrão de C++ que encapsula uma função ou uma tarefa para ser executada de maneira assíncrona
 - introduzido no C++11
- Ele permite associar um `future` a uma tarefa que pode ser executada posteriormente

1. Encapsulamento de Tarefa

- `packaged_task` pode encapsular qualquer função chamável, incluindo funções regulares, funções membro, funções lambda e objetos de função
- A tarefa encapsulada pode ser executada de forma assíncrona

2. Associação com future

- Ao criar um `packaged_task`, ele cria automaticamente um future associado
- Este future pode ser usado para acessar o resultado da tarefa uma vez que ela seja concluída

3. Execução Assíncrona

- Embora `packaged_task` não crie uma nova thread por si só, ele é frequentemente usado com threads para executar tarefas de forma assíncrona
- Ele permite que você separe a definição da tarefa de sua execução

- Recuperação de Resultados
 - Uma vez que a tarefa encapsulada é concluída, seu resultado (ou qualquer exceção lançada) é armazenado no future associado, de onde pode ser recuperado

```
// Uma função simples que calcula o quadrado de um número
int square(int x) {
    return x * x;
}

int main() {
    // Cria um packaged_task que encapsula a função 'square'
    std::packaged_task<int(int)> task(square);

    // Obtém o future associado ao packaged_task
    std::future<int> result = task.get_future();

    // Executa o packaged_task em uma thread separada
    std::thread t(std::move(task), 4); // Calcula o quadrado de 4

    // Aguarda o resultado
    int value = result.get(); // Bloqueia até que o resultado esteja disponível

    // Exibe o resultado
    std::cout << "O quadrado de 4 é: " << value << std::endl;

    // Junta a thread antes de sair
    t.join();

    return 0;
}
```

```
int divide(int a, int b) { // No need to pass a promise ref here!
    if(b == 0) {
        throw std::runtime_error{"Divide by zero exception"};
    }
    return a / b;
}

int main() {
    auto task = std::packaged_task<decltype(divide)>{divide};
    auto f = task.get_future();
    std::thread{std::move(task), 45, 5}.detach();

    try {
        const auto& result = f.get(); // Blocks until ready
        std::cout << "Result: " << result << '\n';
    } catch(const std::exception& e) {
        std::cout << "Caught exception: " << e.what() << '\n';
    }
    return 0;
}
```

std::async



- `std::async` é uma função da biblioteca padrão de C++ usada para executar uma tarefa (função ou expressão lambda) de forma assíncrona
 - introduzida no C++11
- Isso significa que a tarefa pode ser executada em uma thread separada, permitindo que o programa continue sua execução sem esperar que a tarefa seja concluída

- **Execução Assíncrona de Tarefas**
 - `std::async` permite que você execute uma função de forma assíncrona, o que é útil para operações demoradas ou bloqueantes, como IO ou cálculos intensivos.
- **Simplicidade e Facilidade de Uso**
 - Uma das principais vantagens de `std::async` é a sua simplicidade
 - Com apenas uma chamada de função, você pode iniciar uma tarefa assíncrona e obter um future para acessar seu resultado
- **Retorno através de `std::future`**
 - Quando uma tarefa é iniciada usando `std::async`, ela retorna um objeto `std::future`
 - Este future pode ser usado para obter o resultado da tarefa assíncrona

- `std::async` aceita uma política de execução como primeiro argumento
- Determina se a tarefa deve
 - ser executada imediatamente em uma nova thread (`std::launch::async`)
 - ser adiada até que o resultado seja necessário (`std::launch::deferred`)
- Se você não especificar a política de execução, o compilador escolherá uma
 - Isso pode levar a comportamentos diferentes, dependendo da implementação
 - Então é aconselhado sempre defini-la de forma explícita

```
// Uma função simples que calcula o quadrado de um número
int square(int x) {
    return x * x;
}

int main() {
    // Iniciando uma tarefa assíncrona para calcular o quadrado de 4
    std::future<int> result = std::async(square, 4);

    // Fazendo outras operações aqui se necessário...

    // Obtendo o resultado da tarefa assíncrona
    int value = result.get(); // Bloqueia até que o resultado esteja disponível

    // Exibindo o resultado
    std::cout << "O quadrado de 4 é: " << value << std::endl;

    return 0;
}
```

- Bloqueio na Chamada `.get()`
 - Assim como com outros `future`, a chamada `.get()` no `future` retornado por `std::async` bloqueia até que o resultado esteja disponível
- Gerenciamento de Recursos:
 - É importante estar ciente do gerenciamento de recursos ao usar `std::async`, especialmente em relação à criação e destruição de threads
- `std::async` é uma ferramenta extremamente útil para simplificar a execução de tarefas assíncronas em C++, permitindo que você escreva código concorrente de forma mais limpa e segura

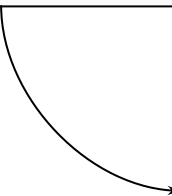
```
auto f = std::async(std::launch::async, do_something);

while(true)
{
    using namespace std::chrono_literals;
    auto status = f.wait_for(500ms);

    if(status == std::future_status::ready)
        break;

    std::cout << "waiting..." << '\n';
}

std::cout << "done!" << '\n';
```

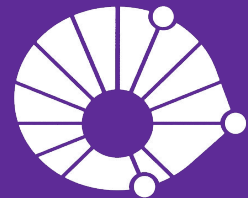
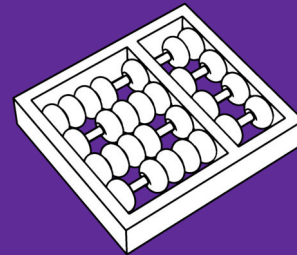


```
waiting...
waiting...
waiting...
operation 1 done
done!
```

Resumo

- Estruturas de dados threadsafe
- Operações Atômicas
- Operações Assíncronas
- Future/Promise
- Tasks
- Async

Obrigado!
Merci!



UNICAMP

Pallette



BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

BUBBLE

Lorem ipsum dolor
sit amet, consectetur
adipiscing elit.

DRACULA

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

Table Title	
Column 1	Column 2
One	Two
Three	Four

