

Sincronização das Threads

Threads e Concorrência em C++

Hervé Yviquel

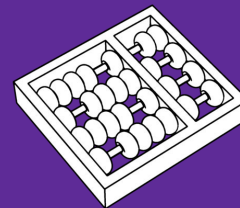
hyviquel@unicamp.br

Universidade Estadual de Campinas (Unicamp)

Instituto de Computação (IC)

Laboratório de Sistemas de Computação (LSC)

Programação em C++ • Out-Nov 2023



UNICAMP

Plano

- Sincronização
- Primeiro Exemplo
- Busy Waiting
- Exclusão Mútua
- Variável de Condição
- Barreiras
- Operações Atômicas
- Acesso a Estrutura

Sincronização



- A principal causa da ocorrência de erros na programação de threads está relacionada com o fato dos dados serem todos compartilhados
 - Apesar de este ser um aspetos mais poderosos da utilização de threads, também pode ser um dos mais problemáticos
- O problema existe normalmente quando dois ou mais threads tentam aceder/alterar as mesmas estruturas de dados
 - race conditions
 - o resultado depende da ordem em que os acessos ocorrem

Existem vários mecanismos de sincronização

- **Busy-wait**
 - Espera ativa
- **Mutex e lock**
 - Para situações de curta duração
 - Equivalente a um semáforo binário
- **Variável de Condição**
 - Para situações em que o tempo de espera não é previsível
 - Pode depender da ocorrência de um evento
- **Semáforo com contadores**
 - Útil quando várias unidades de um recurso estão disponíveis

Primeiro Exemplo



$$\pi = 4 \left(1 - \frac{1}{3} + \frac{1}{5} - \frac{1}{7} + \cdots + (-1)^n \frac{1}{2n+1} + \cdots \right)$$

```
double factor = 1.0;  
double sum = 0.0;  
for (i = 0; i < n; i++, factor = -factor) {  
    sum += factor/(2*i+1);  
}  
pi = 4.0*sum;
```

```
void* Thread_sum(void* rank) {  
    long my_rank = (long) rank;  
    double factor;  
    long long i;  
    long long my_n = n/thread_count;  
    long long my_first_i = my_n*my_rank;  
    long long my_last_i = my_first_i + my_n;  
  
    if (my_first_i % 2 == 0) /* my_first_i is even */  
        factor = 1.0;  
    else /* my_first_i is odd */  
        factor = -1.0;  
  
    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {  
        sum += factor/(2*i+1);  
    }  
  
    return NULL;  
} /* Thread_sum */
```


	n			
	10^5	10^6	10^7	10^8
π	3.14159	3.141593	3.1415927	3.14159265
1 Thread	3.14158	3.141592	3.1415926	3.14159264
2 Threads	3.14158	3.141480	3.1413692	3.14164686

- Note que à medida que aumentamos n , a estimativa com uma única thread vai melhorando
- Mas o que está ocorrendo com 2 threads?

Time	Thread 0	Thread 1
1	Started by main thread	
2	Call Compute ()	Started by main thread
3	Assign $y = 1$	Call Compute ()
4	Put $x=0$ and $y=1$ into registers	Assign $y = 2$
5	Add 0 and 1	Put $x=0$ and $y=2$ into registers
6	Store 1 in memory location x	Add 0 and 2
7		Store 2 in memory location x

```
y = Compute(my_rank);  
x = x + y;
```

y privada

x compartilhada

Busy Waiting



- Uma thread repetidamente testa uma condição, mas, efetivamente não faz qualquer trabalho útil até que a condição seja satisfeita

flag inicializada para 0 na thread principal

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```


o que pode ocorrer se eu usar -O3?

- Compiladores otimizantes podem ser um problema!!

- Antes e depois das otimizações

```
y = Compute(my_rank);  
while (flag != my_rank);  
x = x + y;  
flag++;
```

```
y = Compute(my_rank);  
x = x + y;  
while (flag != my_rank);  
flag++;
```



- Precisa reinicializar flag se quiser reusar várias vezes

```
flag = (flag+1) % thread_count;
```

```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor) {
        while (flag != my_rank);
        sum += factor/(2*i+1);
        flag = (flag+1) % thread_count;
    }

    return NULL;
} /* Thread_sum */
```

serial: 2.8s

2 threads: 19.5s

como melhorar?

qual o problema aqui?



```
void* Thread_sum(void* rank) {
    long my_rank = (long) rank;
    double factor, my_sum = 0.0;
    long long i;
    long long my_n = n/thread_count;
    long long my_first_i = my_n*my_rank;
    long long my_last_i = my_first_i + my_n;

    if (my_first_i % 2 == 0)
        factor = 1.0;
    else
        factor = -1.0;

    for (i = my_first_i; i < my_last_i; i++, factor = -factor)
        my_sum += factor/(2*i+1);

    while (flag != my_rank);
    sum += my_sum;
    flag = (flag+1) % thread_count;

    return NULL;
} /* Thread_sum */
```

Tempos de execução (em segundos) de programas para o cálculo do π usando $n = 10^8$ termos em um sistema com 2 processadores de 4 núcleos cada (8 núcleos)

Threads	Busy-Wait	Mutex
1	2.90	2.90
2	1.45	1.45
4	0.73	0.73
8	0.38	0.38
16	0.50	0.38
32	0.80	0.40
64	3.56	0.38

$$\frac{T_{\text{serial}}}{T_{\text{parallel}}} \approx \text{thread_count}$$

Exclusão Mútua



- A exclusão mútua é uma maneira direta de sincronizar múltiplas threads
 - As threads adquirem um lock em um objeto mutex antes de entrar em uma seção crítica
 - As threads liberam seu lock no mutex ao sair de uma seção crítica
- Modelo de programação de alto nível
 - O recurso (geralmente uma classe) que requer proteção contra condições de corrida possui um objeto mutex do tipo apropriado
 - Threads que pretendem acessar o recurso adquirem um lock adequado no mutex antes de realizar o acesso real
 - As threads liberam seu lock no mutex após completar o acesso
 - Geralmente, os locks são simplesmente adquiridos e liberados nas funções membro da classe

- A biblioteca padrão define várias classes úteis que implementam mutexes nos cabeçalhos `<mutex>` e `<shared_mutex>`
 - `std::mutex` – exclusão mútua regular
 - `std::recursive_mutex` – exclusão mútua recursiva
 - `std::shared_mutex` – exclusão mútua com bloqueios compartilhados
 - Usado para bloquear somente com escrita
- A biblioteca padrão fornece wrappers RAI para bloquear e desbloquear mutexes
 - `std::lock_guard` – wrapper RAI para bloqueio exclusivo
 - `std::scoped_lock` – wrapper RAI para bloqueio exclusivo (C++ 17)
 - `std::unique_lock` – wrapper RAI para bloqueio exclusivo
 - `std::shared_lock` – wrapper RAI para bloqueio compartilhado

- Os wrappers RAII devem sempre ser preferidos para bloquear e desbloquear mutexes
 - Torna bugs devido a bloqueio/desbloqueio inconsistente muito mais improváveis
 - Bloqueio e desbloqueio manual podem ser necessários em alguns casos raros
 - Ainda assim, devem ser realizados através das funções correspondentes dos wrappers RAII

- std::unique_lock pode ser usado para dar lock em um mutex no modo exclusivo
 - O construtor adquire um lock exclusivo no mutex
 - Sintaxe do construtor: unique_lock(mutex_type& m)
 - Bloqueia a thread chamadora até que o mutex esteja disponível
 - O destrutor libera o lock automaticamente
 - Pode ser usado com qualquer tipo de mutex da biblioteca padrão.

```
#include <mutex>
#include <iostream>

std::mutex printMutex;
void safe_print(int i) {
    std::unique_lock lock(printMutex); // lock is acquired
    std::cout << i;
} // lock is released
```

- `std::unique_lock` fornece construtores adicionais
 - `unique_lock(mutex_type& m, std::defer_lock_t t)` – Não dá lock no mutex imediatamente
 - `unique_lock(mutex_type& m, std::try_to_lock_t t)` – Não bloqueia quando o mutex não pode ser locked
- `std::unique_lock` fornece funções membro adicionais
 - `lock()` – Dá lock no mutex manualmente
 - `try_lock()` – Tenta dar lock no mutex, retorna true se for bem-sucedido
 - `operator bool()` – Verifica se o `std::unique_lock` possui um lock no mutex
 - `unlock()` – Libera o mutex manualmente

```
#include <mutex>

std::mutex mutex;

void foo() {
    std::unique_lock lock(mutex, std::try_to_lock);
    if (!lock) {
        doUnsyncronizedWork();

        // block until we can get the lock
        lock.lock();
    }

    doSynchroniziedWork();

    // release the lock early
    lock.unlock();

    doUnsyncronizedWork();
}
```

- std::unique_lock é movível para transferir a propriedade de um lock em um mutex

```
#include <mutex>

class MyContainer {
private:
    std::mutex mutex;

public:
    class iterator { /* ... */ };

    iterator begin() {
        std::unique_lock lock(mutex);

        // compute the begin iterator constructor args

        // keep the lock for iteration
        return iterator(std::move(lock), ...);
    }
};
```


O seguinte código resultará em deadlock, já que `std::mutex` pode ser bloqueado no máximo uma vez.

```
#include <mutex>

std::mutex mutex;

void bar() {
    std::unique_lock lock(mutex);

    // do some work...
}

void foo() {
    std::unique_lock lock(mutex);

    // do some work...

    bar(); // INTENTIONALLY BUGGY, will deadlock
}
```

- `std::recursive_mutex` implementa semânticas de propriedade recursiva
 - A mesma thread pode dar lock em um `std::recursive_mutex` várias vezes sem bloquear
 - Outras threads ainda serão bloqueadas se um `std::recursive_mutex` estiver atualmente locked
 - Pode ser usado com `std::unique_lock` assim como um `std::mutex` regular
 - Útil para funções que se chamam e usam o mesmo mutex

```
#include <mutex>

std::recursive_mutex mutex;
void bar() {
    std::unique_lock lock(mutex);
}
void foo() {
    std::unique_lock lock(mutex);
    bar(); // OK, will not deadlock
}
```

- `std::shared_lock` pode ser usado para dar lock em um mutex no modo compartilhado
 - Construtores e funções membro análogas ao `std::unique_lock`
 - Várias threads podem adquirir um lock compartilhado no mesmo mutex
 - Tentativas de lock compartilhado são bloqueadas se o mutex estiver locked no modo exclusivo
 - Usável apenas em conjunto com `std::shared_mutex`
- Temos que aderir a algum contrato para escrever programas bem-comportados
 - Mutexes compartilhados são usados principalmente para implementar locks de leitura/escrita
 - Apenas acessos de leitura são permitidos ao segurar um lock compartilhado
 - Acessos de escrita são permitidos apenas ao segurar um lock exclusivo

```
#include <shared_mutex>

class SafeCounter {
private:
    mutable std::shared_mutex mutex;
    size_t value = 0;

public:
    size_t getValue() const {
        std::shared_lock lock(mutex);
        return value; // read access
    }

    void incrementValue() {
        std::unique_lock lock(mutex);
        ++value; // write access
    }
};
```

- Normalmente, temos que tornar os mutexes mutáveis dentro de nossas estruturas de dados
 - Os wrappers RAII requerem referências mutáveis ao mutex
 - Funções membro const de nossa estrutura de dados geralmente também precisam usar o mutex
- Usar mutexes sem cuidado pode facilmente levar a deadlocks dentro do sistema
 - Normalmente ocorre quando uma thread tenta dar lock em outro mutex quando já possui um lock em algum mutex
 - Em alguns casos, pode ser evitado usando `std::recursive_mutex` (se estivermos dando lock no mesmo mutex várias vezes)
 - Requer técnicas de programação dedicadas quando vários mutexes estão envolvidos.

```
std::mutex m1, m2, m3;  
void threadA() {  
    // INTENTIONALLY BUGGY  
    std::unique_lock l1{m1}, l2{m2}, l3{m3};  
}  
void threadB() {  
    // INTENTIONALLY BUGGY  
    std::unique_lock l3{m3}, l2{m2}, l1{m1};  
}
```

- Cenário de deadlock possível:
 - threadA() adquire locks em m1 e m2
 - threadB() adquire lock em m3
 - threadA() espera que threadB() libere m3
 - threadB() espera que threadA() libere m2.

- Deadlocks podem ser evitados sempre travando os mutexes em uma ordem globalmente consistente:
 - Garante que uma thread sempre "vença"
 - Manter uma ordem de travamento globalmente consistente requer considerável disciplina do desenvolvedor
 - Manter uma ordem de travamento globalmente consistente pode não ser possível em alguns casos.

```
std::mutex m1, m2, m3;  
void threadA() {  
    // OK, will not deadlock  
    std::unique_lock l1{m1}, l2{m2}, l3{m3};  
}  
void threadB() {  
    // OK, will not deadlock  
    std::unique_lock l1{m1}, l2{m2}, l3{m3};  
}
```

- Às vezes, não é possível garantir uma ordem globalmente consistente:
 - O wrapper RAII `std::scoped_lock` pode ser usado para travar com segurança qualquer número de mutexes
 - Usa um algoritmo de prevenção de deadlock, se necessário
 - Geralmente é bastante ineficiente em comparação ao `std::unique_lock`
 - Deve ser usado apenas como último recurso!

```
std::mutex m1, m2, m3;  
void threadA() {  
    // OK, will not deadlock  
    std::scoped_lock l{m1, m2, m3};  
}  
void threadB() {  
    // OK, will not deadlock  
    std::scoped_lock l{m3, m2, m1};  
}
```


Variável de Condição



- Os mutexes permitem prevenir acessos simultâneos a variáveis compartilhadas mas, as vezes, pode ser bastante ineficiente
 - Se pretendermos realizar uma dada tarefa apenas quando uma dada variável tome um certo valor, temos que consultar sucessivamente a variável até que esta tome o valor pretendido
 - O ideal seria adormecer o thread enquanto a condição pretendida não sucede

```
bool flag;  
std::mutex m;  
void wait_for_flag()  
{
```

```
    std::unique_lock<std::mutex> lk(m);
```

```
    while(!flag)
```

```
    {
```

```
        lk.unlock();
```

```
        std::this_thread::sleep_for(std::chrono::milliseconds(100));
```

```
        lk.lock();
```

```
    }
```

```
}
```

1 Unlock the mutex.

Sleep for 100 ms. 2

3 Relock the mutex.

- Uma variável de condição é uma primitiva de sincronização que permite a várias threads esperar até que uma condição (arbitrária) se torne verdadeira.
 - Uma variável de condição usa um mutex para sincronizar threads
 - As threads podem esperar ou notificar a variável de condição
 - Quando uma thread espera na variável de condição, ela bloqueia até que outra thread a notifique
 - Se uma thread esperou na variável de condição e é notificada, ela detém o mutex
 - Uma thread notificada deve verificar a condição explicitamente porque despertares espúrios podem ocorrer

- A biblioteca padrão define a classe `std::condition_variable` no cabeçalho `<condition_variable>`, que possui as seguintes funções membro:
 - `wait()`: Recebe uma referência a um `std::unique_lock` que deve estar travado pelo chamador como argumento, destrava o mutex e espera pela variável de condição
 - `notify_one()`: Notifica uma única thread que está esperando, o mutex não precisa estar travado pelo chamador
 - `notify_all()`: Notifica todas as threads que estão esperando, o mutex não precisa estar travado pelo chamador

- Um caso de uso para variáveis de condição são filas de trabalhadores:
 - Tarefas são inseridas na fila
 - Em seguida, as threads de trabalho são notificadas para realizar a tarefa

```
std::mutex m;  
std::condition_variable cv;  
std::vector<int> taskQueue;  
  
void pushWork(int task) {  
    {  
        std::unique_lock l{m};  
        taskQueue.push_back(task);  
    }  
    cv.notify_one();  
}
```

```
void workerThread() {  
    std::unique_lock l{m};  
    while (true) {  
        while (!taskQueue.empty()) {  
            int task = taskQueue.back();  
            taskQueue.pop_back();  
            l.unlock();  
            // [...] do actual work here  
            l.lock();  
        }  
        cv.wait(l);  
    }  
}
```

- Variáveis de condição aceitam também
 - `wait_for()` – uma duração
 - `wait_until()` – um ponto no tempo

```
#include <condition_variable>
#include <mutex>
#include <chrono>
std::condition_variable cv;
bool done;
std::mutex m;
bool wait_loop()
{
    auto const timeout= std::chrono::steady_clock::now()+
        std::chrono::milliseconds(500);
    std::unique_lock<std::mutex> lk(m);
    while(!done)
    {
        if(cv.wait_until(lk,timeout)==std::cv_status::timeout)
            break;
    }
    return done;
}
```

Barreiras





- Sincronizar as threads de modo a garantir que todas elas chegam no mesmo ponto do programa é chamado barreira
- Nenhuma thread pode cruzar a barreira até que todas as threads tenham alcançado-a

- **Início Coordenado:** Fazer com que todas as threads comecem uma tarefa simultaneamente.
- **Progressão Sequencial:** Assegurar a conclusão de uma etapa de trabalho antes de começar a próxima.
- **Integridade de Dados:** Manter a consistência dos dados entre threads que acessam recursos compartilhados.
- **Sincronização de Timesteps:** Alinhar threads em simulações baseadas em tempo ou iterações.
- **Balanceamento de Trabalho:** Esperar que todas as threads terminem suas tarefas atuais antes de redistribuir o trabalho.
- **Consolidação de Resultados:** Aguardar a conclusão do processamento paralelo antes de combinar resultados

- Implementar barreiras usando busy-waiting e mutex é bem direto
 - No fundo, estamos usando um contador compartilhado e protegido pelo mutex
 - Quando o contador indicar que todas as threads entraram a seção crítica, as threads podem deixar a seção crítica

```
class Barrier {
private:
    int count;
    int original_count;
    std::mutex mtx;

public:
    explicit Barrier(int count) : count(count), original_count(count) {}

    void wait() {
        std::unique_lock<std::mutex> lock(mtx);
        count--;
        if (count == 0) {
            count = original_count;
        } else {
            // Busy wait
            lock.unlock();
            while (true) {
                lock.lock();
                if (count == original_count) {
                    lock.unlock();
                    break;
                }
                lock.unlock();
            }
        }
    }
};
```

```
class Barrier {
public:
    explicit Barrier(std::size_t count) : thread_count(count), counter(0),
                                         generation(0) {}

    void Wait() {
        std::unique_lock<std::mutex> lock(mutex_);
        auto gen = generation;
        if (++counter == thread_count) {
            generation++;
            counter = 0;
            cond_.notify_all();
        } else {
            cond_.wait(lock, [this, gen] { return gen != generation; });
        }
    }

private:
    std::mutex mutex_;
    std::condition_variable cond_;
    std::size_t thread_count;
    std::size_t counter;
    std::size_t generation;
};
```

- Contador que permite a várias threads aguardar até que um conjunto de operações seja concluído
 - Inicializado com um valor fixo
 - `wait()` para bloquear até que chegue a zero
 - `count_down()` para decrementar
 - `arrive_and_wait()`
 - Quando atinge zero, todas as threads bloqueadas são automaticamente liberadas

```
void DoWork(threadpool* pool) {  
    latch completion_latch(NTASKS);  
    for (int i = 0; i < NTASKS; ++i) {  
        pool->add_task([&] {  
            // perform work  
            ...  
            completion_latch.count_down();  
        }));  
    }  
    // Block until work is done  
    completion_latch.wait();  
}
```

- Barreiras

- Diferente do std::latch, pode ser usado repetidamente para sincronizar threads
- Função de Conclusão: Suporta a execução de uma função de callback quando todas as threads alcançam a barreira.
- Método de Espera: Threads utilizam arrive_and_wait() para bloquear até que a barreira seja completada por todas.

```
void DoWork() {
    Tasks& tasks;
    int n_threads;
    vector<thread*> workers;

    barrier task_barrier(n_threads);

    for (int i = 0; i < n_threads; ++i) {
        workers.push_back(new thread([&] {
            bool active = true;
            while(active) {
                Task task = tasks.get();
                // perform task
                ...
                task_barrier.arrive_and_wait();
            }
        }));
    }
    // Read each stage of the task until all stages are complete.
    while (!finished()) {
        GetNextStage(tasks);
    }
}
```

```
int main()
{
    const auto workers = {"Anil", "Busara", "Carl"};


    auto on_completion = []() noexcept
    {
        // locking not needed here
        static auto phase =
            "... done\n"
            "Cleaning up...\n";
        std::cout << phase;
        phase = "... done\n";
    };

    std::barrier sync_point(std::ssize(workers), on_completion);

    auto work = [&](std::string name)
    {
        std::string product = " " + name + " worked\n";
        std::osyncstream(std::cout) << product; // ok, op<< call is atomic
        sync_point.arrive_and_wait();

        product = " " + name + " cleaned\n";
        std::osyncstream(std::cout) << product;
        sync_point.arrive_and_wait();
    };

    std::cout << "Starting...\n";
    std::vector<std::jthread> threads;
    threads.reserve(std::size(workers));
    for (auto const& worker : workers)
        threads.emplace_back(work, worker);
}
```



```
Starting...
  Anil worked
  Carl worked
  Busara worked
... done
Cleaning up...
  Busara cleaned
  Carl cleaned
  Anil cleaned
... done
```

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 vários 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() { /* ... */ }
```

- 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() { /* ... */ }
```

- 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();
    }
}
```

Accesso a Estrutura




```
#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();      ← ❶
    }
    void wait_and_pop(T& value)      ← ❷
    {
        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

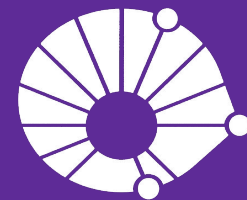
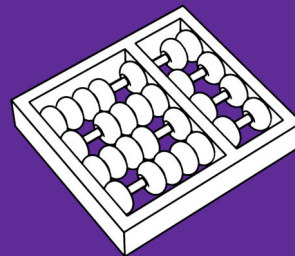
10% Delete

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

Resump

- Sincronização
 - Busy Waiting
 - Exclusão Mútua
 - Variável de Condição
 - Barreiras
 - Operações Atômicas
- Estruturas de dados *threadsafe*

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

