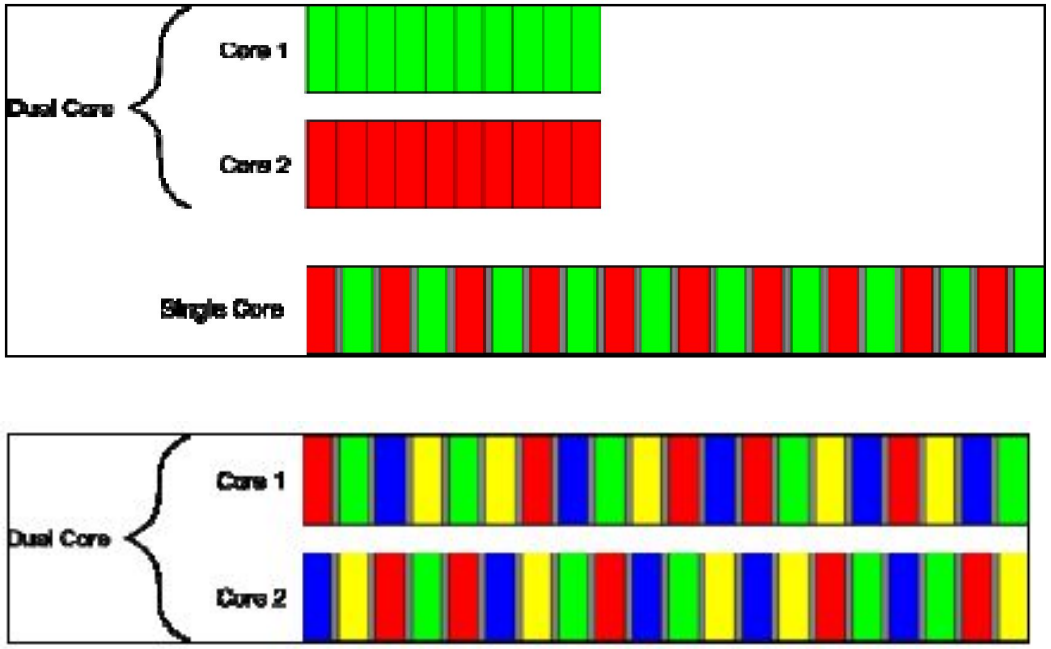
Многопоточные приложения

Concurrency

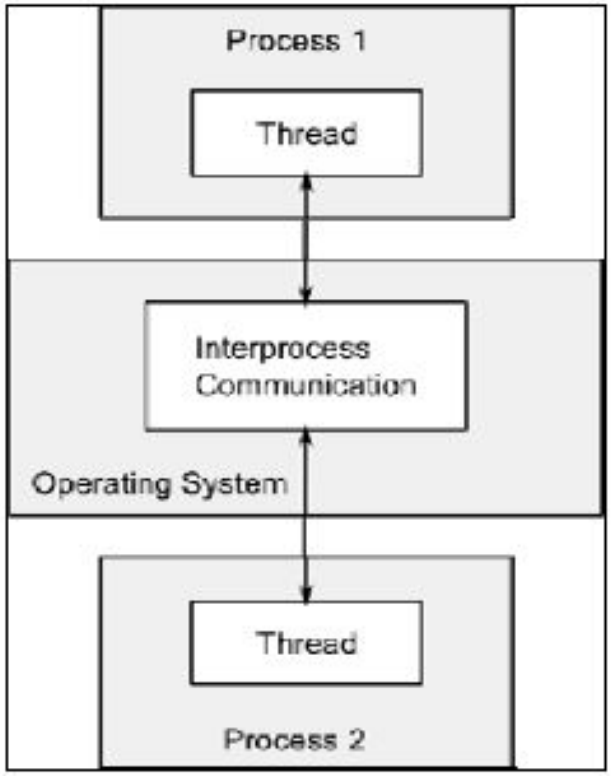


Несколько вычислительных ядер процессора позволяют выполнять несколько задач одновременно.

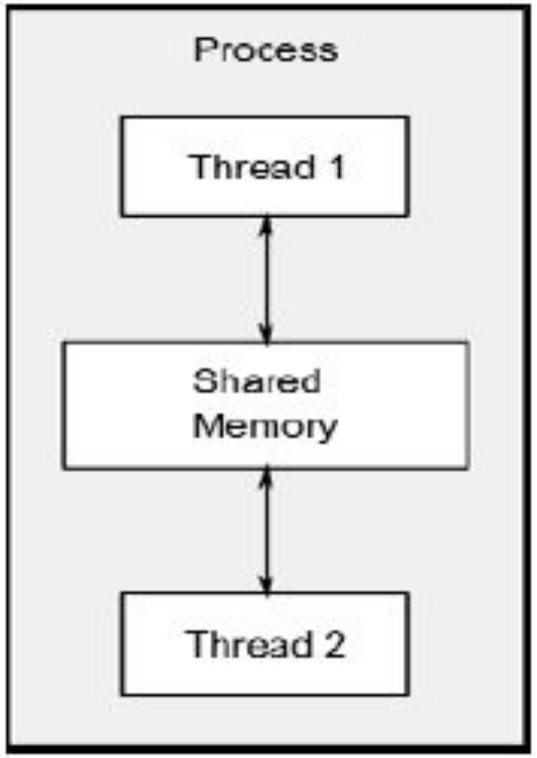
Одно ядро процессора может выполнять несколько задач, только переключаясь между ними.

Два вида многозадачности

Multiple processes



Multiple threads



За чем применять многозадачность?

1. Разделение программы на независимые части. Один процесс выполняет одну задачу (например, взаимодействие с пользователем), а другой – другую (например, вычисления).
2. Для увеличения производительности.

Увеличение числа параллельных процессов не всегда приводит к ускорению программы.

Hello World!

CPP_Examples18

```
#include <iostream>
#include <thread>
void hello() {
std::cout<<"Hello Concurrent World\n";
}

int main(int argc,char * argv[]){
std::thread t(hello); // launch new thread
t.join(); //wait for finish of t

cin.get();
return 0;
}
```

Передаем объекты в поток
CPP_Examples19

std::unique_ptr – позволяет иметь только одну ссылку на объект. Его нельзя копировать.

std::move – позволяет перемещать содержимое unique_ptr;

```
void clearP(std::unique_ptr<BigObject>& ptr) {…}
std::unique_ptr<BigObject> p(new BigObject(4096));
std::thread t(clearP,std::move(p));
```

Полезные функции

std::thread::hardware_concurrency()

Возвращает количество Thread которые могут выполняться параллельно для данного приложения.

std::this_thread::get_id()

Возвращает идентификатор потока.

std::this_thread::sleep_for(std::chrono::milliseconds)

Позволяет усыпить поток на время

Как дождаться завершения потока красиво?

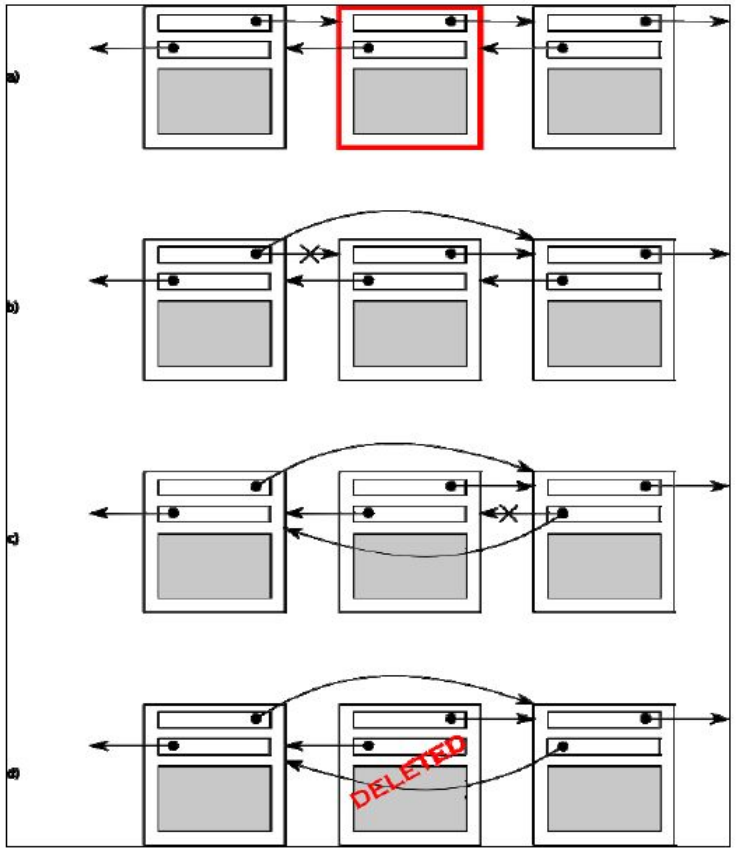
CPP_Examples20

std::for_each – позволяет применять функцию к элементам коллекции

```
template<class InputIterator, class Function>
  Function for_each(InputIterator first, InputIterator last, Function fn)
{
  while (first!=last) {
    fn (*first);
    ++first;
  }
  return fn;       // or, since C++11: return move(fn);
}
```

std::mem_fn – делает из метода класса функцию, первый параметр которой объект класса

Проблемы работы с динамическими структурами данных в многопоточной среде



При удалении элемента из связанного списка производится несколько операций:

- удаление связи с предыдущим элементом

- удаление связи со следующим элементом

- удаление самого элемента списка

Во время выполнения этих операций к этими элементами обращаться из других потоков нельзя!

Mutex
CPP_Examples23

Мьютекс — базовый элемент синхронизации и в C++11 представлен в 4 формах в заголовочном файле <mutex>:

mutex
обеспечивает базовые функции lock() и unlock() и не блокируемый метод try_lock()

recursive_mutex
может войти «сам в себя»

timed_mutex
в отличие от обычного мьютекса, имеет еще два метода: try_lock_for() и try_lock_until()

recursive_timed_mutex
это комбинация timed_mutex и recursive_mutex

Потоко-безопасный Stack
CPP_Examples21

Классы «обертки» позволяют непротиворечиво использовать мьютекс в RAII-стиле с автоматической блокировкой и разблокировкой в рамках одного блока. Эти классы:

lock_guard
когда объект создан, он пытается получить мьютекс (вызывая lock()), а когда объект уничтожен, он автоматически освобождает мьютекс (вызывая unlock())

unique_lock
в отличие от lock_guard, также поддерживает отложенную блокировку, временную блокировку, рекурсивную блокировку и использование условных переменных

Deadlock

```
std::lock_guard<std::mutex> lock(a);

std::lock_guard<std::mutex> lock(b);
```

```
std::lock_guard<std::mutex> lock(b);

std::lock_guard<std::mutex> lock(a);
```

Возникает когда несколько потоков пытаются получить доступ к нескольким ресурсам в разной последовательности.

Exceptions в многопоточной среде

CPP_Examples_22

1. Исключения между потоками не передаются!
2. Нужно устроить хранилище исключений, для того что бы их потом обработать!

Условные переменные <condition_variable>

CPP_Examples24,25

condition_variable

требует от любого потока перед ожиданием сначала выполнить std::unique_lock

condition_variable_any

более общая реализация, которая работает с любым типом, который можно заблокировать

1. Должен быть хотя бы один поток, ожидающий, пока какое-то условие станет истинным. Ожидающий поток должен сначала выполнить unique_lock.
2. Должен быть хотя бы один поток, сигнализирующий о том, что условие стало истинным. Сигнал может быть послан с помощью notify_one(), при этом будет разблокирован один (любой) поток из ожидающих, или notify_all(), что разблокирует все ожидающие потоки.
3. В виду некоторых сложностей при создании пробуждающего условия, которое может быть предсказуемых в многопроцессорных системах, могут происходить ложные пробуждения (spurious wakeup). Это означает, что поток может быть пробужден, даже если никто не сигнализировал условной переменной. Поэтому необходимо еще проверять, верно ли условие пробуждение уже после то, как поток был пробужден.

Lambda
CPP_Examples26

Лямбда-выражения в C++ — это краткая форма записи анонимных функторов.

Например:

```
[](int _n)   {    cout << _n << " ";}
```

Соответствует:

```
class MyLambda
{
    public: void operator ()(int _x) const { cout << _x << " "; }
};
```

Лямбда функции могут возвращать значения

CPP_Examples27

В случае, если в лямбда-функции только один оператор return то тип значения можно не указывать. Если несколько, то нужно явно указать.

```
[] (int i) -> double
    {
        if (i < 5)
            return i + 1.0;
        else if (i % 2 == 0)
            return i / 2.0;
        else
            return i * i;
    }
```

Захват переменных из внешнего контекста

CPP_Examples28

```
[]                      // без захвата переменных из внешней области видимости
[=]                     // все переменные захватываются по значению
[&]                     // все переменные захватываются по ссылке
[this]            // захват текущего класса
[x, y]                  // захват x и y по значению
[&x, &y]                // захват x и y по ссылке
[in, &out]              // захват in по значению, а out — по ссылке
[=, &out1, &out2]       // захват всех переменных по значению, кроме out1 и out2,
                        // которые захватываются по ссылке
[&, x, &y]              // захват всех переменных по ссылке, кроме x…
```

Генерация лямбда-выражений

CPP_Examples29

Начиная со стандарта C++11 шаблонный класс std::function является полиморфной оберткой функций для общего использования. Объекты класса std::function могут хранить, копировать и вызывать произвольные вызываемые объекты - функции, лямбда-выражения, выражения связывания и другие функциональный объекты. Говоря в общем, в любом месте, где необходимо использовать указатель на функцию для её отложенного вызова, или для создания функции обратного вызова, вместо него может быть использован std::function, который предоставляет пользователю большую гибкость в реализации.

Впервые данный класс появился в библиотеке Function в версии Boost 1.23.0[7]. После его дальнейшей разработки, он был включен в стандарт расширения C++ TR1 и окончательно утвержден в C++11.

Определение класса

```
template<class> class function; // undefined

template<class R, class... ArgTypes> class function<R(ArgTypes...)>;
```

Атомарные операции

Атомарность означает неделимость операции. Это значит, что ни один поток не может увидеть промежуточное состояние операции, она либо выполняется, либо нет.

Например операция «++» не является атомарной:

```
int x = 0;

++x;
```

Транслируется в ассемблерный код, примерно так:

```
013C5595  mov         eax,dword ptr [x]
013C5598  add         eax,1
013C559B  mov         dword ptr [x],eax
```

Атомарные типы C++

#include<atomic>

```
std::atomic_bool //bool

std::atomic_char //char

std::atomic_schar //signed char

std::atomic_uchar //unsigned char

std::atomic_int //int

std::atomic_uint //unsigned int

std::atomic_short //short

std::atomic_ushort //unsigned short

std::atomic_long //long

std::atomic_ulong //unsigned long

std::atomic_llong //long long

std::atomic_ullong //unsigned long long

std::atomic_char16_t //char16_t
```

Основные операции

```
load() //Прочитать текущее значение

store() //Установить новое значение

exchange() //Установить новое значение и вернуть предыдущее

compare_exchange_weak() // см. следующий слайд

compare_exchange_strong() // compare_exchange_weak в цикле

fetch_add() //Аналог оператора ++

fetch_or() //Аналог оператора --

is_lock_free() //Возвращает true, если операции на данном типе
неблокирующие
```

Метод atomic::compare_exchange_weak

```
bool compare_exchange_weak( Ty& Exp, Ty Value)
```

Сравнивает значения которые хранится в *this с Exp.

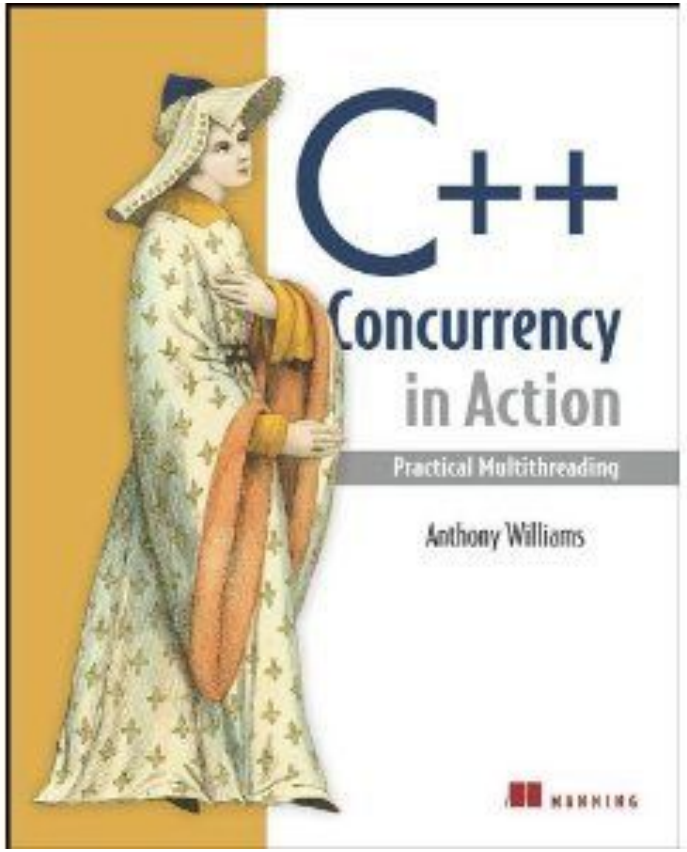
- Если значения равны то операция заменяет значение, которая хранится в *this на Val (*this=val) , с помощью операции read-modify-write.
- Если значения не равны, то операция использует значение, которая хранится в *this, чтобы заменить Exp (exp=this).

Потокобезопасный Stack

CPP_Examples30

```
void push(const T& data)
{
node* new_node = new node(data, head.load());
while (!head.compare_exchange_weak(
new_node->next,
new_node));
}
```

Что еще почитать?



C++ Concurrency in Action
Practical Multithreading
Anthony Williams

February, 2012 | 528 pages
ISBN: 9781933988771

Разные блоги, например:

http://habrahabr.ru/post/182610/