Содержание

1. Модель памяти C/C++ программ

Модель памяти описывает взаимодействие параллельно исполняющихся процессов (нитей) посредством использования данных в общей памяти.

В случае параллельных процессов общая память организуется средствами операционной системы. Например, это может быть файл, отображенный в память с помощью системного вызова `mmap`, либо разделяемая память POSIX (POSIX shared memory — `shm_overview(7)`), либо разделяемая память System V IPC — `shmget(2)`.

В случае нитей вся память является формально общей, так как нити разделяют единое адресное пространство своего процесса. Тем не менее, как правило, мы можем предполагать, что стек каждой нити и thread-local переменные нити используются только этой нитью. Общая память в случае многонитевых программ — это главным образом глобальные переменные независимо от их видимости и данные, расположенные в динамической памяти.

Например:

```
extern int var1;
static int var2;
int var3;
_Thread_local int var4 = 10; // в C++ используем thread_local
void func(int par1)
{
    static int var5 = 15;
    double var6 = 0.0;
```

```
}
```

Переменные `var1`, `var2` и `var3` являются общими для всех нитей, несмотря на то, что переменая `var1` определена (то есть под нее выделена память) в какой-то другой точке программы (возможно, в другой единице компиляции). Переменная `var2` недоступна извне данной единицы компиляции, но тоже является общей для всех нитей.

Переменная `var5` тоже является общей, хотя она видима только внутри функции `func`. Переменная `var5` существует от момента запуска программы, когда она получит начальное значение 15, до момента завершения программы. Если функция `func` будет вызываться из нескольких нитей одновременно, возможен одновременный доступ к одной и той же переменной `var5`.

Переменная `var4` — локальная для нити. То есть память под переменную будет выделена, и переменная будет проинициализирована значением 10 в момент создания нити (например, при `pthread_create`). Память будет освобождена в момент завершения нити. Хотя адрес переменной `var4` можно передать в другие нити, и другие нити могут одновременно обратиться к этой переменной, такие ситуации мы рассматривать не будем. Переменная `var4` не будет общей для нитей, и мы можем работать с этой переменной по правилам работы с глобальными переменными последовательных программ.

Параметр функции `par1` и локальная переменная `var6` не будут считаться общими для нитей, хотя, как и в предыдущем случае, их адрес может быть передан за пределы текущей нити.

В языке C++ инициализация глобальных объектов может потребовать вызова конструктора и исполнения произвольного кода. В случае глобальных и статических переменных, объявленных в глобальной области данных, конструкторы объектов вызываются до вызова функции `main`. Деструкторы глобальных объектов вызываются в порядке, обратном к порядку вызова конструкторов после завершения функции `main`.

```
std::vector<int> var1;
static std::vector<int> var2(var1);
namespace
{
    std::vector<int> var3{1, 2, 3, 4};
}
```

В этом примере конструкторы переменных `var1`, `var2`, `var3` (в этом порядке) будут вызваны до функции `main`. Поскольку выполнение программы начинается в одном главном потоке, при работе конструкторов глобальных объектов нет проблемы одновременного вызова конструкторов в нескольких потоках.

Но если переменная объявлена как статическая в теле функции, ситуация меняется.

```
void func()
{
    static std::vector<int> var4(100);
}
```

Конструктор для переменной `var4` будет вызван при первом вызове функции `func` в коде программы. Возможно, что функция `func` будет в первый раз одновременно вызвана сразу в нескольких нитях. В этом случае гарантируется, что конструктор выполнится только в одной нити, а остальные нити будут ждать завершения инициализации переменной `var4`.

1.1. Последовательные синхронные программы

По умолчанию в стандартах C/C++ предполагается, что программа выполняется последовательно и синхронно, то есть при выполнении программы не наступают события, не предусмотренные последовательно выполняющейся программой. Компилятор C/C++ может предполагать, что вся память принадлежит только данной программе, состояние ячеек памяти модифицируется только в результате описанных в программе последовательности действий.

Компилятор может преобразовывать программу, например, модифицируя порядок действий в программе, или вообще удаляя операторы, если компилятор может гарантировать, что результат выполнения программы не изменится.

Например, компилятор может оптимизировать функцию

```
int counter;
void *thread_func(void *ptr)
{
    for (int i = 0; i < 100000; ++i) {
        ++counter;
    }
    return 0;
}
```

заменив цикл на одно присваивание `counter += 100000;`, так как компилятор имеет право считать, что при выполнении функции `thread_func` только она модифицирует переменную `counter`.

1.2. Асинхронное исполнение

Функции обработки прерываний, функции обработки сигналов или даже функции обратного вызова (callbacks) вносят в программу асинхронность. В случае обработки сигналов основной поток выполнения может быть в произвольный момент времени прерван, выполнится некоторая функция-обработчик, затем выполнение программы продолжится с точки, в которой оно было прервано. Функция-обработчик сигнала может изменить значения глобальных переменных.

Однако компилятор программы не делает никаких предположений о возможных асинхронных событиях и предполагает, что программа исполняется строго последовательно и синхронно. Программист сам должен отметить переменные, которые могут быть изменены асинхронно, так как в противном случае компилятор имеет право считать, что переменные изменяются только в основной программе.

Для этого используется ключевое слово `volatile`. Оно указывает компилятору, что значение переменной может быть прочитано или модифицировано в асинхронно-выполняющемся фрагменте программы. Это означает, что, если значение `volatile`-переменной модифицируется, то оно должно быть записано в память немедленно, то есть компилятор не имеет права оставить значение переменной на регистре, чтобы записать его в память позднее. Если значение `volatile`-переменной считывается, компилятор должен загрузить значение из памяти.

Для `volatile`-переменных гарантируется относительный порядок записи в память или чтения из памяти, если обращения разделены точками последовательных вычислений (sequence points), например,

```
volatile int var1;
volatile int var2;
```

```
void func()
{
    // ...
    var1 = 1;
    var2 = 2; // var2 будет модифицирована строго после var1
}
```

Но не в следующей ситуации:

```
volatile int var1;
volatile int var2;
volatile int var3;

void func()
{
    // ...
    var3 = (var1 = 1) + (var2 = 2);
    func2(var1, var2, var3);
}
```

Здесь, поскольку модификации переменных находятся внутри одного выражения (не разделены точками последовательных вычислений — sequence points), относительный порядок модификации переменных `var1`, `var2` и `var3` может быть произвольным. Относительный порядок чтения значений переменных при вызове функции `func2` также произволен.

Доступ к `volatile`-переменным никак не упорядочивает доступ к обычным переменным. Например,

```
int var1;
volatile int var2;
int var3;

void func()
{
    // ...
    var1 = 1;
    var2 = 2;
    var3 = 3;
}
```

Переменные `var1` и `var3` могут быть сохранены в память в произвольном порядке как до сохранения `var2`, так и после. Другими словами, доступ к `volatile`-переменной не является барьером для обычных переменных.

В случае указателей `volatile` трактуется аналогично квалификатору `const`:

```
int *p1;                    // обычный указатель на обычный int
volatile int *p2;           // обычный указатель на volatile int
int * volatile p3;          // volatile указатель на обычный int
volatile int * volatile p4; // volatile указатель на volatile int
```

1.3. Модель памяти

Современные процессоры даже при исполнении последовательной программы используют разные средства для ускорения исполнения. Например, кеши позволяют со-

кратить количество обращений в основную память, суперскалярное исполнение позволяет выполнять более одной инструкции за такт, предсказание переходов позволяет снизить накладные расходы на перезагрузку конвейера при условных переходах и т. д. Тем не менее, с точки зрения программы она исполняется строго последовательно в том смысле, что результат ее выполнения совпадает с тем, какой был бы, если бы никаких средств для ускорения работы не использовалось.

Но при параллельном исполнении сложные механизмы исполнения программы на каждом из ядер дают в совокупности еще более сложные для понимания особенности выполнения параллельной программы в целом.

Рассмотрим фрагмент кода, выполняющийся в одной из нитей.

```
var1 = 1;
var2 = 2;
var3 = 3;
```

В той нити, которая выполняет фрагмент кода, значения переменных будут изменены в порядке `var1`, `var2`, `var3`. То есть для этой нити справедливы импликации:

```
if (var2 == 2) {
    assert(var1 == 1);
}
if (var3 == 3) {
    assert(var2 == 2);
}
```

Однако такой порядок в общем случае не гарантирован для других нитей. Вторая нить может «увидеть» изменения значений переменных в порядке `var2`, `var1`, `var3`, а третья — в порядке `var3`, `var2`, `var1`.

Минимальные требования к перестановкам операций с памятью между разными нитями задаются *моделью памяти*. Модель памяти — это свойство процессорной архитектуры.

Сильная модель памяти (strong memory model) требует, что если одно процессорное ядро выполняет запись в память в определенной последовательности, все другие процессорные ядра видят изменения значений в памяти в той же самой последовательности. Процессоры x86/x64 практически всегда реализуют сильную модель памяти. Процессоры x86/x64 не могут переставлять записи в память друг относительно друга, но могут менять местами записи и чтения. Каждое процессорное ядро может отложить запись в память и выполнить чтение из независимой ячейки памяти перед этим.

Рассмотрим пример:

```
// thread 1                    // thread 2
movl    $1, X                  movl    $1, Y
movl    Y, %eax                movl    X, %eax
```

Предполагая начальные значения переменных X и Y равными 0, мы можем ожидать, что в зависимости от порядка выполнения нитей либо в первой, либо во второй, либо в обеих нитях в регистре `%eax` окажется значение 1. Однако на самом деле в обеих нитях в регистре `%eax` может оказаться значение 0. Это произойдет, когда операции будут выполнены в следующем порядке:

```
(1) movl Y, %eax
(2) movl X, %eax
(1) movl $1, X
(2) movl $1, Y
```

В *слабой модели памяти* процессор может переставлять порядок и операций чтения, и операций записи при условии, что такие перестановки не изменяют результат вычисления программы состоящей из одной синхронно выполняющейся нити. Соответственно, и другие нити могут видеть изменения в памяти в произвольном порядке. Такая модель памяти была реализована в процессоре Alpha.

В более современных процессорах (ARM, PPC, Itanium) используется чуть более строгая модель, чем у Alpha за счет того, что учитываются зависимости по данным. Например, если выполняется операция `p->f`, то есть из памяти считывается значение поля `f` по указателю `p`, то гарантируется, что процессорное ядро, выполняющее обращение по указателю `p` увидит изменение в поле `f`, которое было выполнено в другом ядре раньше чем, изменение `p`.

То есть, если одно процессорное ядро выполняет операции

```
q->f = 1;
p = q;
```

то другое процессорное ядро, выполняющее обращение `p->f` по новому значению `p`, гарантированно прочитает новое значение поля `f`. Конечно, другое процессорное ядро может выполнить обращение `p->f` по старому значению `p`.

Слабая модель памяти дает гораздо больший простор для компилятора и процессора в оптимизации порядка доступа к памяти в параллельно выполняющихся нитях.

Однако, программа, которая подразумевает для своей корректной работы процессор с сильной моделью памяти (x86/x64) может работать некорректно на процессоре со слабой моделью памяти (ARM)!

Стандарты языков C и C++ задают слабую модель памяти. Программы, неявно предполагающие сильную модель памяти, скорее всего, будут с точки зрения стандарта демострировать неопределенное поведение.

1.4. Data race

Чтобы гарантировать корректное выполнение программы на архитектурах с разными моделями памяти от Alpha до x86 необходимо в программе на C/C++ явно обозначить конструкции, выполнение которых может зависить от модели памяти.

Если несколько нитей одновременно обращаются к одной и той же ячейке памяти, причем хотя бы одна из нитей выполняет запись в эту ячейку, такая ситуация называется *data race*. Если при выполнении программы возникает data race, дальнейшее поведение программы не определено (undefined behavior). Программы, при выполнении которых возможны data race, являются некорректными программами на C или C++.

Если в программе отсутствуют data race, такая программа будет выполняться корректно на процессоре с любой моделью памяти.

Если в программе присутствуют data race, они могут проявляться как ошибки типа race condition, то есть программа будет давать различные результаты при разных запусках, либо программа будет работать правильно на одних архитектурах и неправильно на других архитектурах.

Чтобы в программе не возникали data races, все обращения к разделяемым данным должны быть корректно выделены в программе. Например, обращение к разделяемым данным может быть заключено в критическую секцию, охраняемую мьютексом. Это сделает одновременную запись в разделяемую переменную невозможной.

Естественно, реализации стандартных мьютексов, условных переменных, семафоров и других примитивов синхронизации в стандартной библиотеке или библиотеке pthread не содержат data race.

Но мьютекс может быть слишком дорогим для организации корректного доступа к разделяемой переменной. Возможно, что свойства модели памяти позволят получить требуемый результат свободный от data race и без использования «тяжелых» синхронизационных операций.

1.5. Atomic-переменные

Ключевое слово `_Atomic` в языке C или шаблонный класс `std::atomic<T>` используются для описания переменных, к которым возможен одновременный доступ из нескольких нитей без возникновения data race. Например,

```
_Atomic int var1;
struct Item * _Atomic head;
```

Если подключить заголовочный файл `<stdatomic.h>` становятся доступными типы `atomic_bool`, `atomic_char`, `atomic_int` и т. д. Их можно использовать для совместимости с C++.

Операции с atomic переменными атомарны. Например, в функции

```
_Atomic int counter;
void *thread_func(void *ptr)
{
    for (int i = 0; i < 100000; ++i)
        ++counter;
    return ptr;
}
```

гарантируется, что чтение-модификация-запись переменной `counter` в цикле будет выполнена атомарно, то есть независимо от других нитей, выполняющих данную функцию, значение переменной `counter` будет 100000 раз гарантированно увеличено на 1.

Хотя отдельная операция с atomic-переменной и будет атомарна, последовательность атомарных операций в совокупности может и не быть атомарной.

```
_Atomic int value;

    value += 5;             // атомарная операция
    value = value * 2 + 5; // не атомарная операция
```

Ключевое слово `_Atomic` применимо ко всем типам данных C, в том числе массивам и структурам. Не гарантируется, однако, что для любого типа данных атомарная работа с ним может быть реализована на уровне инструкций процессора. В этом случае компилятор может генерировать код, который будет использовать другие примитивы для синхронизации, например, мьютексы. В любом случае, для `_Atomic` переменной гаранируется атомарная работа, свободная от data race.

Узнать, требует ли atomic-тип «тяжелой» синхронизации можно с помощью макросов `ATOMIC_<T>_LOCK_FREE`, которые могут принимать значение 0, означающее что атомарная работа с типом `T` всегда требует «тяжелой» синхронизации, 1, означающее, что атомарная работа с типом `T` не во всех случаях требует «тяжелой» синхронизации,

и значение 2, если тип T синхронизуется средствами системы команд процессора. Например, макрос ATOMIC_LLONG_LOCK_FREE позволяет получить информацию о статусе поддержки типа _Atomic long long.

Работа с самой _Atomic переменной атомарна, но влияние на операции работы с обычными переменными, расположенными около операции с атомарной переменной, может быть разным. При работе с атомарной переменной можно указать, как она влияет на операции с неатормарными переменными рядом с с атомарной.

1.6. Порядок работы с памятью (memory order)

Порядок работы с памятью определяет, как неатомарные обращения к памятью могут быть переупорядочены вокруг атомарной операции. Как было показано выше, в слабой модели памяти принятой в стандартах C и C++, когда несколько нитей одновременно считывают из памяти и модифицируют несколько переменных, наблюдаемый в некоторой нити порядок записи в память может отличаться от действительного порядка записи. Даже в однопроцессорной системе возможны аналогичные эффекты из-за разрешенных слабой моделью памяти перестановок операций чтения и записи.

По умолчанию для всех атомарных операций используется последовательно консистентный (sequentially consistent) порядок работы с памятью как самый строгий. Библиотечные функции работы с атомарными переменными могут принимать дополнительный аргумент типа memory_order для указания дополнительных к атомарности требований к упорядоченности обращений к памяти.

Перечислимый тип memory_order определен в заголовочном файле <stdatomic.h> следующим образом:

```
enum memory_order
{
    memory_order_relaxed,
    memory_order_consume,
    memory_order_acquire,
    memory_order_release,
    memory_order_acq_rel,
    memory_order_seq_cst
};
```

1.6.1. memory_order_relaxed

memory_order_relaxed - самый слабый порядок работы с памятью. Операция с атомарной переменной в этом режиме предполагает только атомарность и не дает никаких гарантий сохранения порядка записей в память или чтения из памяти для операций вокруг данной, если только они не выполняются с более сильным порядком работы с памятью.

```
atomic_int counter;
void *thread_func(void *ptr)
{
    for (int i = 0; i < 100000; ++i)
        atomic_fetch_add_explicit(&counter, 1, memory_order_relaxed);
    return ptr;
}
```

гарантируется, что чтение-модификация-запись переменной `counter` в цикле будет выполнена атомарно, то есть независимо от других нитей, выполняющих данную функцию, значение переменной `counter` будет 100000 раз гарантированно увеличено на 1. На платформах со слабой моделью памяти операция `atomic_fetch_add_explicit` может быть более быстрой, чем операция инкремента `++counter`.

С другой стороны, не гарантируется сохранение порядка независимых записей и чтения в параллельных нитях.

```
// Thread 1:
r1 = atomic_load_explicit(y, memory_order_relaxed); // A
atomic_store_explicit(x, r1, memory_order_relaxed); // B

// Thread 2:
r2 = atomic_load_explicit(x, memory_order_relaxed); // C
atomic_store_explicit(y, 42, memory_order_relaxed); // D
```

Допускается, что в результате выполнения этого фрагмента значения переменных `r1 == r2 == 42`, если операции будут выполнены в порядке D, A, B, C.

`memory_order_relaxed` может применяться для увеличения значения счетчика ссылок (но для уменьшения значения должен применяться более строгий режим), для доступа на чтение к переменной состояния нити и т. д.

1.6.2. memory_order_consume

Режим работы `memory_order_consume` объявлен устаревшим в стандарте C++17. На настоящее время режим работы `memory_order_consume` так и не был поддержан ни одним компилятором, которые реализуют вместо этого режима работы режим работы `memory_order_acquire`.

1.6.3. memory_order_acquire и memory_order_release

Парная операция в режимах acquire и release образуется, когда в нити A выполняется сохранение в память в режиме `memory_order_release`, а в нити B выполняется чтение из той же самой переменной в режиме `memory_order_acquire`. В этом случае все записи в память (и обычные неатомарные, и в режиме `memory_order_relaxed`), которые с точки зрения нити A выполняются раньше атомарнй операции сохранения в память гарантированно становятся видимыми в нити B. Таким образом, после выполнения атомарной загрузки из памяти в нити B она гарантированно увидит все, что нить A записала в память.

Эта синхронизация затрагивает только нити, выполняющие операции release и acquire над одной и той же атомарной операцией. Другие нити могут видеть записи в память в порядке, отличном от видимого порядка в нитях A или B.

На платформах с сильной моделью памяти (например, x86/x64) порядок release-acquire выполняется для большинства операций с памятью. То есть, инструкция процессора чтения из памяти работает в режиме acquire, а инструкция записи в память работает в режиме release. На таких платформах не требуется дополнительных инструкций процессора для этого режима синхронизации.

На платформах со слабой моделью памяти (ARM, PPC) для обеспечения release-acquire синхронизации требуются специальные инструкции процессора для записи и чтения или барьеров памяти.

В любом случае при использовании записи в память в режиме release компилятору запрещено переставлять неатомарные или relaxed записи, которые в тексте программы находились до атомарной записи в память, после нее. И наоборот, неатомарные и relaxed операции чтения, которые находились в тексте программы до атомарной загрузки из памяти в режиме acquire запрещено менять местами с атомарной загрузкой.

Синхронизация release-acquire обычно используется для реализации спинлоков. Например:

```
atomic_flag m = ATOMIC_FLAG_INIT;
void lock()
{
    while (atomic_flag_test_and_set_explicit(&m, memory_order_acquire)) {}
}
void unlock()
{
    atomic_flag_clear_explicit(&m, memory_order_release);
}
```

1.6.4. memory_order_acq_rel

Режим работы `memory_order_acq_rel` применим к операциям типа чтение-модификация-запись (`atomic_exchange`, `atomic_fetch_add` и т. п.). Ни операции чтения, ни операции записи в данной нити не могут быть переставлены местами с данной атомарной операцией. Все операции записи в другой нити, которые выполняются до операции release над той же самой переменной будут видимы в данной нити после данной атомарной операции. Наоборот, все операции записи выполненные до данной атомарной операции в текущей нити будут видны в других нитях, которые выполняют операцию acquire.

1.6.5. memory_order_seq_cst

Последовательно консистентный режим работы — самый строгий. Атомарная операция, выполняющаяся в режиме `memory_order_seq_cst`, не только является и release, и acquire операцией, но кроме того все нити видят все операции, выполненные в этом режиме, в одном и том же порядке. Этот режим действует по умолчанию для атомарных операций. Например,

```
atomic_int var1;
atomic_int var2;

// thread
var1 += 10; // (A)
++var2;     // (B)
```

Гарантируется, что все нити увидят модификации в порядке A, B.

В примере

```
// thread 1
var1 += 10; // (A)
++var2;     // (B)

// thread 2
var2 —= 2;  // (C)
```

```
--var1;     // (D)
```

операции могут выполнится в любом из порядков: ABCD, ACBD, ACDB, CABD, CADB, CDAB. Но в каком бы порядке операции не выполнились, все остальные нити увидят модификации переменных `var1` и `var2` именно в таком порядке.

В противоположность этому режим работы acquire-release не гарантирует фиксированного порядка видимости обновлений для нитей, не участвующих в синхронизации.

1.7. Атомарные операции

В этом разделе дается краткое описание атомарных операций стандарта C11.

1.7.1. Операции с atomic_flag

```
struct atomic_flag;    // непрозрачный тип

#define ATOMIC_FLAG_INIT ...

_Bool atomic_flag_test_and_set(volatile atomic_flag* obj);
_Bool atomic_flag_test_and_set_explicit(volatile atomic_flag* obj, memory_order
    order);

void atomic_flag_clear(volatile atomic_flag* obj);
void atomic_flag_clear_explicit(volatile atomic_flag* obj, memory_order order);
```

Тип `atomic_flag` — это атомарный булевский тип. Только для него стандартом C гарантируется реализация lock-free — без «тяжелой» синхронизации (мьютексов). Для других типов, даже для `atomic_bool` такой гарантии не предоставляется, хотя, конечно, все современные платформы поддерживают lock-free синхронизацию для типов размера до машинного слова включительно (то есть до 32 бит для 32-битных платформ и 64 бит для 64-битных платформ).

Для типа `atomic_flag` определены только операции `test_and_set` и `clear`. По умолчанию они выполняются в режиме `memory_order_seq_cst`, но с помощью функций `*_explicit` можно явно указать требуемый режим синхронизации. Переменная типа `atomic_flag` должна быть явно проинициализирована с помощью `ATOMIC_FLAG_INIT`, в противном случае ее значение не определено.

```
atomic_flag m = ATOMIC_FLAG_INIT;   // инициализируем значением _False (0)
```

Операция `atomic_flag_test_and_set` атомарно устанавливает переменную, на которое указывает параметр `obj`, в `_True` (1) и возвращает старое значение. Операция `atomic_flag_clear` атомарно устанавливает переменную, на которую указывает параметр `obj`, в `_False` (0). Если использовать переменную типа `atomic_flag` в качестве мьютекса, то значение 0 (`_False`) означает, что мьютекс открыт, а значение 1 (`_True`) — что мьютекс закрыт.

Обратите внимание, что тип `atomic_flag` определен как структура, чтобы попытка применения к нему операций, доступных для других атомарных типов, приводило к ошибке компиляции. Кроме того, стандарт не специфицирует, как значения `atomic_flag` хранятся в памяти. Например, `atomic_flag` может храниться в памяти как один байт, в котором значение 1 представляет сброшенный `atomic_flag`, а значение 2 — установленный.

1.7.2. Операции загрузки из памяти и сохранения в память

```
void atomic_store(volatile A* obj, C desired);
void atomic_store_explicit(volatile A* obj, C desired, memory_order order);

C atomic_load(const volatile A* obj);
C atomic_load_explicit(const volatile A* obj, memory_order order);
```

Здесь A — это некоторый атомарный тип, например, atomic_long, а C — соответствующий ему неатомарный тип (long).

Функция atomic_store сохраняет значение desired по адресу, переданному в параметре obj. Если используется explicit функция, то параметр order задает режим работы. По умолчанию используется режим memory_order_seq_cst.

Если atomic-переменной присваивается значение с помощью операции присваивания, например, var = 0;, то подразумевается операция atomic_store.

Функция atomic_load атомарно считывает значение из памяти по адресу obj и возвращает его в качестве результата. Если используется explicit функция, то параметр order задает режим работы. По умолчанию используется режим memory_order_seq_cst.

Если имя atomic-переменной используется в контексте, в котором требуется ее значение, подразумевается операция atomic_load. Например,

```
atomic_int var;

...
printf("%d\n", var);
printf("%d\n", atomic_load(&var)); // то же самое
```

1.7.3. Операции обмена

```
C atomic_exchange(volatile A* obj, C desired);
C atomic_exchange_explicit(volatile A* obj, C desired, memory_order order);

_Bool atomic_compare_exchange_strong(volatile A* obj, C* expected, C desired);
_Bool atomic_compare_exchange_weak(volatile A *obj, C* expected, C desired);
_Bool atomic_compare_exchange_strong_explicit(volatile A* obj,
                                              C* expected, C desired,
                                              memory_order succ,
                                              memory_order fail);
_Bool atomic_compare_exchange_weak_explicit(volatile A *obj,
                                            C* expected, C desired,
                                            memory_order succ,
                                            memory_order fail);
```

Здесь A — это некоторый атомарный тип, например, atomic_long, а C — соответствующий ему неатомарный тип (long).

Функции семейства atomic_exchange сохраняют по адресу obj значение desired, а возвращают в качестве результата старое значение, которое находилось по адресу obj. Эта операция считается операцией типа read-modify-write.

Функции семейства atomic_compare_exchange сравнивают значение по адресу obj со значением по адресу expected. Если значения равны, то значение по адресу obj заменяется на desired (выполняется read-modify-update). В противном случае по адресу expected записывается текущее значение по адресу obj. Функции возвращают результат сравнения *obj == *expected (выполняется только read).

Отличие weak-функций от strong в том, что weak-функции иногда могут вернуть `_False`, даже если значения равны. Они могут выполняться быстрее, чем их strong-аналоги, если используются в цикле:

```
atomic_int var;

// ждем значения 10 в переменной var
int exp = 10;
while (!atomic_compare_exchange_weak(&var, &exp, 0)) {
    exp = 10;
}
```

У explicit-вариантов функций параметр `succ` задает режим работы в случае равенства значений, а `fail` — в случае неравенства. Параметр `succ` может быть любым, а параметр `fail` не может быть `memory_order_release` или `memory_order_acq_rel` и не может быть более строгим, чем `succ`.

1.7.4. Операции модификации

```
// атомарное сложение
// *obj += arg;
C atomic_fetch_add(volatile A* obj, M arg);
C atomic_fetch_add_explicit(volatile A* obj, M arg, memory_order order);

// атомарное вычитание
// *obj -= arg;
C atomic_fetch_sub(volatile A* obj, M arg);
C atomic_fetch_sub_explicit(volatile A* obj, M arg, memory_order order);

// атомарное побитовое или
// *obj |= arg;
C atomic_fetch_or(volatile A* obj, M arg);
C atomic_fetch_or_explicit(volatile A* obj, M arg, memory_order order);

// атомарное побитовое исключающее или
// *obj ^= arg;
C atomic_fetch_xor(volatile A* obj, M arg);
C atomic_fetch_xor_explicit(volatile A* obj, M arg, memory_order order);

// атомарное побитовое и
// *obj &= arg;
C atomic_fetch_and(volatile A* obj, M arg);
C atomic_fetch_and_explicit(volatile A* obj, M arg, memory_order order);
```

Здесь `A` — это некоторый атомарный тип, например, `atomic_long`, а `C` — соответствующий ему неатомарный тип (`long`).

Эти операции относятся к типу read-modify-update.

1.7.5. Барьер

```
void atomic_thread_fence(memory_order order);
```

Устанавливает порядок доступа к памяти для неатомарных и relaxed операций в соответствии с режимом `order`. Например, если нить A выполняет барьер в режиме `memory_order_release`, а нить `B` — барьер в режиме `memory_order_acquire`, все записи в памяти в нити A, выполненные до барьера, будут видимы в нити B после барьера.

2. Lock-free структуры данных

Под lock-free структурами данных понимают структуры данных, оптимизированные для использования в многонитевых программах на многопроцессорных/многоядерных системах таким образом, чтобы минимизировать необходимые для корректной работы операции синхронизации. Некоторые операции могут требовать полной блокировки других нитей, но такие операции в типичном потоке операций встречаются достаточно редко.

2.1. Ленивая инициализация (double checked locking)

Иногда инициализация какой-либо структуры данных откладывается на момент первого использования. Это может быть единственным вариантом, если при запуске программы данных недостаточно, или инициализация данной структуры использует другие структуры, которые в свою очередь должны быть проинициализированы и т. д.

Такие структуры обычно в программе являются *синглтонами*, то есть существуют в единственном экземпляре. В C++ корректная инициализация синглтонов при первом их использовании выполняется очень просто:

```
class SinglClass; // синглтон

SinglClass &getInstance()
{
    static SinglClass instance;
    return instance;
}
```

Конструктор для `instance` будет вызван при первом вызове `getInstance`, при этом гарантируется, что если несколько нитей вызовут getInstance одновременно, конструктор будет вызван только один раз.

Аналогичный фрагмент на C может выглядеть так:

```
struct SinglData;

struct SinglData *init_data();

struct SinglData *data = NULL;

struct SinglData *get_data()
{
    // ???
}
```

Функция `init_data` выделяет память под структуру SinglData, инициализирует ее и возвращает указатель на корректно проинициализированную структуру. Тем не менее функция `init_data` не содержит никакой защиты от одновременного вызова из нескольких нитей. Поэтому функция `get_data` должна гарантировать, что функция `init_data` будет вызвана только один раз независимо от того, сколько нитей вызывают `get_data`.

Простой вариант может быть таким:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
struct SinglData *data = NULL;
```

```
struct SinglData *get_data()
{
    pthread_mutex_lock(&mutex);
    if (!data) {
        data = init_data();
    }
    pthread_mutex_unlock(&mutex);
    return data;
}
```

Поскольку модификация переменной `data` защищена мьютексом, нет необходимости делать эту переменную атомарной. Этот вариант плох тем, что при каждом обращении к `get_data` будет блокироваться мьютекс, что совершенно избыточно, так как все обращения, кроме первого, будут просто считывать и возвращать значение глобальной переменной `data`.

Необходимо избавиться от мьютекса на горячем пути выполнения этой функции, то есть тогда, когда значение переменной `data` не равно NULL. Нам все равно мьютекс потребуется, но он будет использоваться только для исключения ситуации одновременной инициализации из нескольких нитей.

Первый вариант:

```
struct SinglData *get_data()
{
    if (!data) {
        pthread_mutex_lock(&mutex);
        data = init_data();
        pthread_mutex_unlock(&mutex);
    }
    return data;
}
```

Он, во-первых, не гарантирует, что `init_data` будет вызван только один раз. Если несколько нитей одновременно проверят текущее значение переменной `data`, убедятся, что оно равно NULL, войдут в тело оператора `if`. Затем каждая из этих нитей по очереди выполнит `init_data`. Во-вторых, в программе присутствует data race: возможно обращение на чтение к переменной `data` одновременно с ее модификацией в нити, которая выполняет инициализацию.

Второй вариант:

```
struct SinglData *get_data()
{
    if (!data) {
        pthread_mutex_lock(&mutex);
        if (!data) {
            data = init_data();
        }
        pthread_mutex_unlock(&mutex);
    }
    return data;
}
```

Мы добавили проверку на то, что `data` все еще NULL внутри критической секции. Теперь даже если несколько нитей войдут внутрь оператора `if`, только одна из нитей,

проверив еще раз, что переменная data равна NULL, запустит инициализацию. Этот вариант по-прежнему содержит data race в случае, когда одновременно модифицируется значение data и она проверяется на NULL при входе в функцию, но на первый взгляд такая функция должна работать корректно.

Это действительно так для платформ с сильной моделью памяти. Но на системах со слабой моделью памяти все не так однозначно. От функции инициализации мы ожидаем, что она выделит память, затем как-то заполнит поля структуры, затем адрес структуры мы присвоим переменной data. То есть ожидаемая последовательность действий будет такой:

```
p = malloc(sizeof(*p));
p->a = 1;   // инициализируем поле a
p->b = 2;   // инициализируем поле b
data = p;
```

На системах с сильной моделью памяти все нити видят записи в память в одном и том же порядке, но на системах со слабой моделью памяти это не гарантируется! Например, какая-то нить может сначала увидеть запись в переменную data, затем запись в поле b а затем запись в поле a. Очевидно, это может привести к ошибке при выполнении нити. Таким образом, data race на переменной data действительно сигнализирует о проблеме в коде.

Поэтому нам требуется дополнительная синхронизация на переменной data. Проверка значения data должна выполняться в режиме acquire, а присваивание нового значения - в режиме release. Улучшенный фрагмент программы будет таким:

```
pthread_mutex_t mutex = PTHREAD_MUTEX_INITIALIZER;
struct SinglData * _Atomic data = NULL; // атомарный указатель

struct SinglData *get_data()
{
    struct SinglData *d = atomic_load_explicit(&data, memory_order_acquire);
    if (!d) {
        pthread_mutex_lock(&mutex);
        d = atomic_load_explicit(&data, memory_order_acquire);
        if (!d) {
            d = init_data();
            atomic_store_explicit(&data, d, memory_order_release);
        }
        pthread_mutex_unlock(&mutex);
    }
    return d;
}
```

В типичном пути выполнения программы, когда значение переменной data не равно NULL, выполняется одна атомарная загрузка из памяти.

На платформе x86 (сильная модель памяти) операции atomic_load_explicit и atomic_store_explicit преобразовываются в обычные инструкции выборки из памяти и сохранения в память:

```
        // d = atomic_load_explicit(&data, memory_order_acquire);
        movl    data, %eax
```

```
        // atomic_store_explicit(&data, d, memory_order_release);
        movl    %eax, data
```

На платформе ARMv7 будут сгенерированы дополнительные инструкции барьера памяти.

```
        ldr     r0, [r5]
        mcr     p15, 0, r0, c7, c10, 5
```

Описанная выше идиома потокобезопасной ленивой инициализации называется double checked locking.

2.2. Read-copy-update (RCU)

Рассмотрим структуру данных, например, двусвязный список. Список может быть определен примерно следующим образом:

```
struct Item
{
    struct Item *prev, *next;
    int data; // whatever...
};
struct List
{
    struct Item *first, *last;
};
```

Тип `struct List` хранит указатель на первый и на последний элемент списка. Если список пуст, эти указатели равны NULL. Попробуем адаптировать его к многонитевой программе. Понятно, что несколько нитей одновременно могут обращаться к списку в режиме чтения, но если мы хотим список модифицировать, только одна нить одновременно может модифицировать список, все остальные нити — и читатели, и писатели, должны ждать. Это типичная задача «читатели-писатели». Поэтому можем добавить в `struct List` соответствующий синхронизационный примитив библиотеки pthread.

```
struct List
{
    struct Item *first, *last;
    pthread_rwlock_t rwl;
};
```

Функция-читатель должна работать в критической секции, ограниченной функциями `pthread_rwlock_rdlock` и `pthread_rwlock_unlock`. Функция-писатель должна работать в критической секции, ограниченной функциями `pthread_rwlock_wrlock` в начале и `pthread_rwlock_unlock` в конце.

```
void reader(struct List *lst)
{
    pthread_rwlock_rdlock(&lst->rwl);
    // process the list
    pthread_rwlock_unlock(&lst->rwl);
}
void writer(struct List *lst)
```

```
{
    pthread_rwlock_wrlock(&lst->rwl);
    // update the list
    pthread_rwlock_unlock(&lst->rwl);
}
```

Это решение безусловно корректно. Его недостаток в том, что на время работы нити-писателя нити-читатели не имеют никакого доступа к списку. В совокупности с тем, что реализация этих примитивов в Linux в первую очередь дает доступ писателям это может привести к ситуации голодания читателей, если поток писателей будет достаточно плотным.

Чтобы избежать такой ситуации дадим читателям постоянный доступ к списку. Возможно читатель будет работать не с последней версией списка, но в любом случае он не будет ждать неограниченное время чтобы получить доступ. То, что читатель будет работать не с самой последней версией списка, нам не страшно. В многонитевой программе у нас и так нет способа гарантировать это, кроме как вводить барьеры читателей-писателей, что плохо с точки зрения параллельности работы.

Чтобы не возникали data race между читателями и писателем писатель сначала сделает копию списка, затем модифицирует его как нужно, затем заменят основной список на его копию. Введем соответствующие структуры данных.

```
struct Head
{
    struct Item *first, *last;
};
struct List
{
    struct Head *head;    // здесь указатель на актуальную версию списка
    pthread_mutex_t wm;   // мьютекс для писателей
};
```

Читатель просто берет указатель head и работает с ним, а писатель выполняет работу в критической секции:

```
void reader(struct List *lst)
{
    struct Head *head = lst->head;
    // дальше работаем со списком в head
}
void writer(struct List *lst)
{
    pthread_mutex_lock(&lst->wm);
    struct Head *new_head = copy_list(lst->head); // копируем список
    // модифицируем список new_head
    lst->head = new_head; // замещаем старый список новым
    pthread_mutex_unlock(&lst->wm);
}
```

Но есть проблемы: во-первых, непонятно когда освобождать память, занятую старой копией списка. Писатель это делать не может, так как в этот самый момент со старой копией списка могут работать один или несколько читателей. Во-вторых, data race на поле head, когда читатель берет текущий список для обработки, с одной стороны, и писателем, который записывает в поле head новый список, с другой стороны. Вторая проблема решается введением acquire-release операций над полем head.

Первую проблему решим введением счетчика читателей. Каждый читатель, входя в обработку списка, увеличивает счетчик, а выходя из обработчика уменьшает его. Писатель после замещения старого списка новым добавляет старый список в специальный список списков, подлежащих уничтожению. Дополнительная функция будет проходить по списку уничтожаемых списков и освобождать те из них, в которых не осталось читателей. Таким образом старый список рано или поздно будет удален.

```
struct Head
{
    struct Item *first, *last;
    atomic_int workers; // число читателей
};
struct List
{
    struct Head * _Atomic head; // актуальная версия списка
    pthread_mutex_t wm;   // мьютекс для писателей
};
void reader(struct List *lst)
{
    // берем текущую версию списка
    struct Head *head = atomic_load_explicit(&lst->head, memory_order_acquire);
    // увеличиваем счетчик читателей
    atomic_fetch_add_explicit(&head->workers, 1, memory_order_relaxed);
    // выполняем обработку списка
    DO_READ();
    // уменьшаем счетчик читателей
    atomic_fetch_add_explicit(&head->workers, -1, memory_order_relaxed);
}
void writer(struct List *lst)
{
    pthread_mutex_lock(&lst->wm);
    // берем текущую версию списка
    struct Head *old_head = atomic_load_explicit(&lst->head, memory_order_relaxed);
    struct Head *new_head = copy_list(old_head); // копируем список
    // модифицируем список new_head...
    DO_MODIFY();
    // замещаем текущий список новым
    atomic_store_explicit(&lst->head, new_head, memory_order_release);
    pthread_mutex_unlock(&lst->wm);
    // помещаем старый список в очередь на удаление
    add_to_reclaim_list(old_head);
    // пытаемся почистить старые списки
    reclaim_list();
}
struct ReclaimableListItem
{
    struct ReclaimableListItem *next;
    struct Head *head;
};
struct ReclaimableList
{
    pthread_mutex_t m;
```

```
    struct ReclaimableList *first;
};
// список всех списков на удаление
// правильнее его сделать синглтоном см(. выше)
// с этим списком работаем под мьютексом всегда
static struct ReclaimableList rcl = { PTHREAD_MUTEX_INITIALIZER, NULL };
void add_to_reclaim_list(struct Head *head)
{
    struct ReclaimableListItem *p;
    pthread_mutex_lock(&rcl.m);
    // если в списке есть пустой слот, вписываем head в него
    for (p = rcl.first; p; p = p->next) {
        if (!p->head) {
            p->head = head;
            break;
        }
    }
    // если пустых слотов нет, создаем
    if (!p) {
        p = calloc(1, sizeof(*p));
        p->next = rcl.first;
        p->head = head;
        rcl.first = p;
    }
    pthread_mutex_unlock(&rcl.m);
}
void reclaim_list(void)
{
    pthread_mutex_lock(&rcl.m);
    // просматриваем список
    // освобождаем слоты, в списках которых счетчик читателей равен 0
    for (struct ReclaimableListItem *p = rcl.first; p; p = p->next) {
        if (p->head && !atomic_fetch_explicit(&p->head->workers,
    memory_order_relaxed)) {
            free_list(p->head);
            p->head = NULL;
        }
    }
    pthread_mutex_unlock(&rcl.m);
}
```

Предложенную реализацию можно пооптимизировать в части обработки освобождения «отработанных» списков.

Как видно из рассмотренных примеров атомарные типы и неблокирующие операции для работы с ними не предназначены для полной замены «тяжелых» средств синхронизации, таких как мьютексы или условные переменные, а позволяеют оптимизировать «горячие» пути в типичных сценариях работы.