Многопоточное программирование

Современные вычислительные системы содержат процессоры, имеющие несколько вычислительных ядер.

Каждое ядро представляет собой независимое арифметико-логическое устройство (АЛУ) и набор регистров. Ядро может выполнять последовательность из инструкций (программу) независимо от другого ядра.

Классическая программа подразумевает последовательное выполнение, когда результат предыдущей операции влияет на последующую, и поэтому не может выполняться одновременно на нескольких ядрах.

Использование нескольких ядер, а в перспективе — и всей вычислительной мощности процессора — требует специального подхода в разработке ПО.

Многопоточное программирование

Поток выполнения (thread) — это непрерывная последовательность команд, предназначенная для выполнения на одном ядре.

Процесс — это представление выполняемой программы на уровне операционной системы. Процесс описывается своей памятью, используемыми ресурсами (например, файлы) и содержит как минимум один поток выполнения.

Один процесс может содержать несколько потоков, выполняемых одновременно. При этом потоки живут в общем адресном пространстве процесса, но каждому из них соответствует свое состояние выполнения.

Современные операционные системы могут автоматически распределять разные потоки одного процесса на разные ядра процессора.

Многопоточное программирование

В многозадачной операционной системе может одновременно выполняться большое количество процессов, существенно большее, чем количество доступных ядер центрального процессора.

Операционная система применяет специальные правила работы процессов, которые позволяют им по очереди использовать одно ядро. При достаточно быстром переключении процессов создается иллюзия одновременности их работы.

Многопоточное программирование

Поскольку прикладное программное обеспечение пишется сторонними разработчиками, ОС применяет механизмы изоляции процессов, чтобы обеспечить стабильность работы системы. В современных ОС прикладное ПО:

- Работает через механизм виртуальной памяти. Каждый процесс «видит» непрерывное адресное пространство, в котором нет никого кроме него. Фактическое размещение страниц памяти процесса в физической памяти обеспечивает менеджер памяти операционной системы.
- Может быть в любой момент приостановлен средствами ОС и снят с ядра. Состояние процесса сохраняется, и через некоторое время ОС возвращает его на ядро, после чего процесс продолжает работу с того же места. Для процесса данная операция незаметна.

Многопоточное программирование

Потоки выполнения внутри одного процесса не изолированны друг от друга, и каждому потоку доступна вся виртуальная память процесса.

При этом потоки разных процессов изолированы друг от друга.

Потоки точно так же могут быть сняты с ядра в произвольный момент времени и возвращены обратно для продолжения работы с того же места. Это может нарушить работу приложения, если несколько потоков работают с одними и теми же данными в памяти процесса.

Все эти факты требуют специального подхода при разработке приложения, чей процесс будет содержать несколько потоков.

Поток выполнения

В любом языке программирования поток выполнения представляется как функция, запускаемая извне. Основной поток приложения стартует с функции **main**.

Допускается запускать несколько потоков, выполняющих одну и ту же функцию. При этом у каждой запущенной функции будет свой набор аргументов и локальные переменные, но все они могут пользоваться одними и теми же глобальными переменными.

Запущенная функция никак не ограничена. Она может вызывать другие функции, работать с динамической памятью, файлами и т.д.

Потоки в C++

Начиная со стандарта C++11, существует способ запуска потоков при помощи заголовочного файла <thread>

```
#include <thread>
```

Для запуска потока необходимо определить функцию без возвращаемого значения (**void**) и создать объект типа **std::thread**. Аргументами конструктора являются имя функции и значения передаваемых в нее аргументов. Поток начнет выполняться сразу после создания объекта.

```
void add(double a, double b, double* sum) {
    *sum = a + b;
}

int main() {
    double s;
    thread t(add, 2, 2, &s);
}
```

Потоки в C++

Основной поток программы немедленно продолжит выполнение и в приведенном примере завершится до того, как поток закончит свою работу. Необходимо вызвать у потока метод **join**, чтобы дождаться его завершения.

```
void add(double a, double b, double* sum) {
    *sum = a + b;
}

int main() {
    double s;
    thread t(add, 2, 2, &s);
    t.join();
    return 0;
}
```

Потоки в C++

```
#include <thread>

void add(int a, int b, int* sum)
{
  *sum = a + b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  int sum[4];
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      add, v1[a], v2[a], &sum[a]
    );
  for (thread t& : pool)
    t.join();
  return 0;
}
```

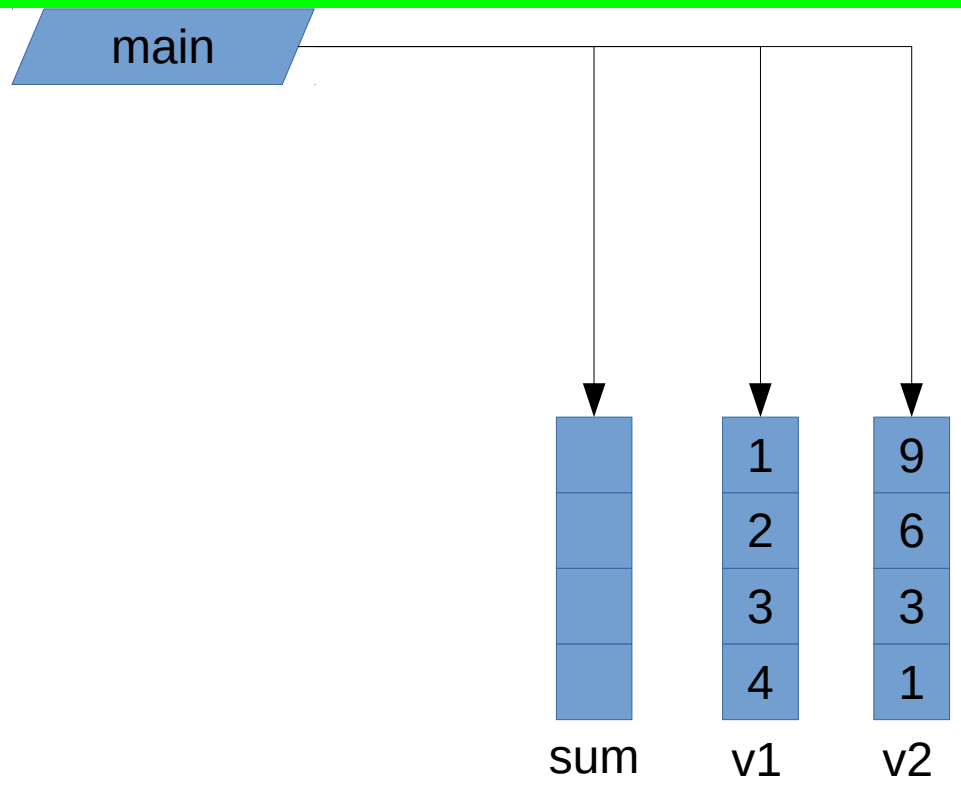
main

Потоки в C++

```
#include <thread>

void add(int a, int b, int* sum)
{
  *sum = a + b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  int sum[4];
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      add, v1[a], v2[a], &sum[a]
    );
  for (thread t& : pool)
    t.join();
  return 0;
}
```

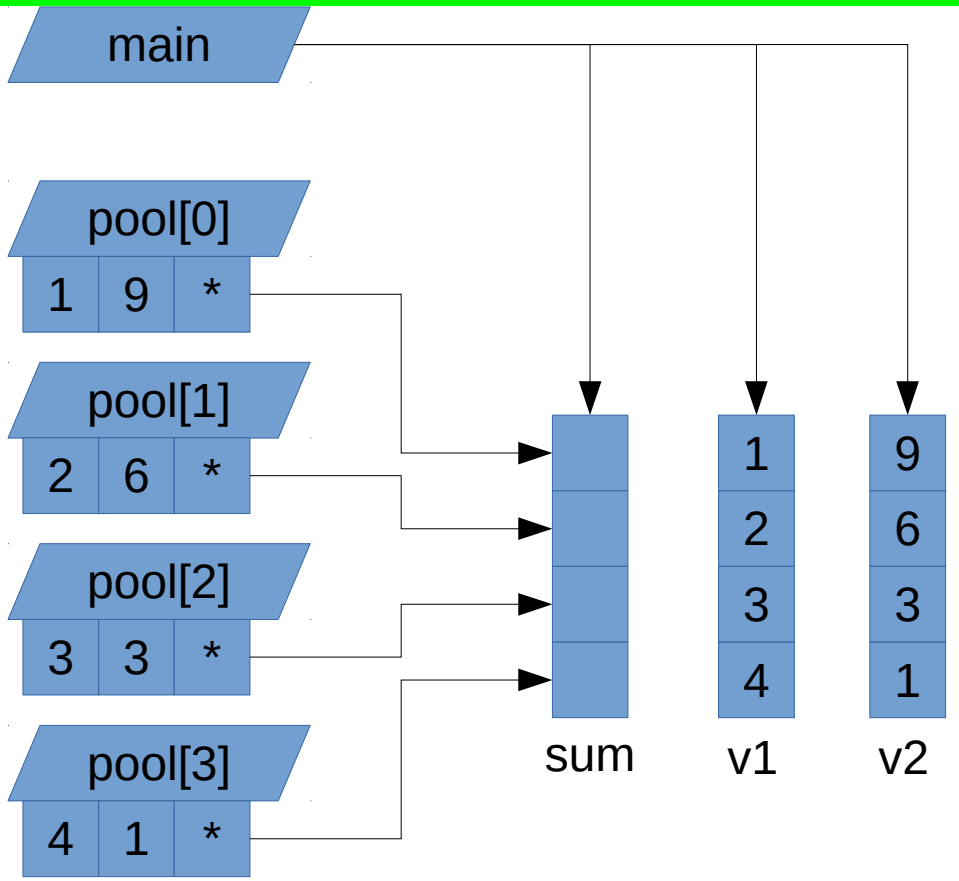


Потоки в C++

```
#include <thread>

void add(int a, int b, int* sum)
{
  *sum = a + b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  int sum[4];
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      add, v1[a], v2[a], &sum[a]
    );
  for (thread t& : pool)
    t.join();
  return 0;
}
```

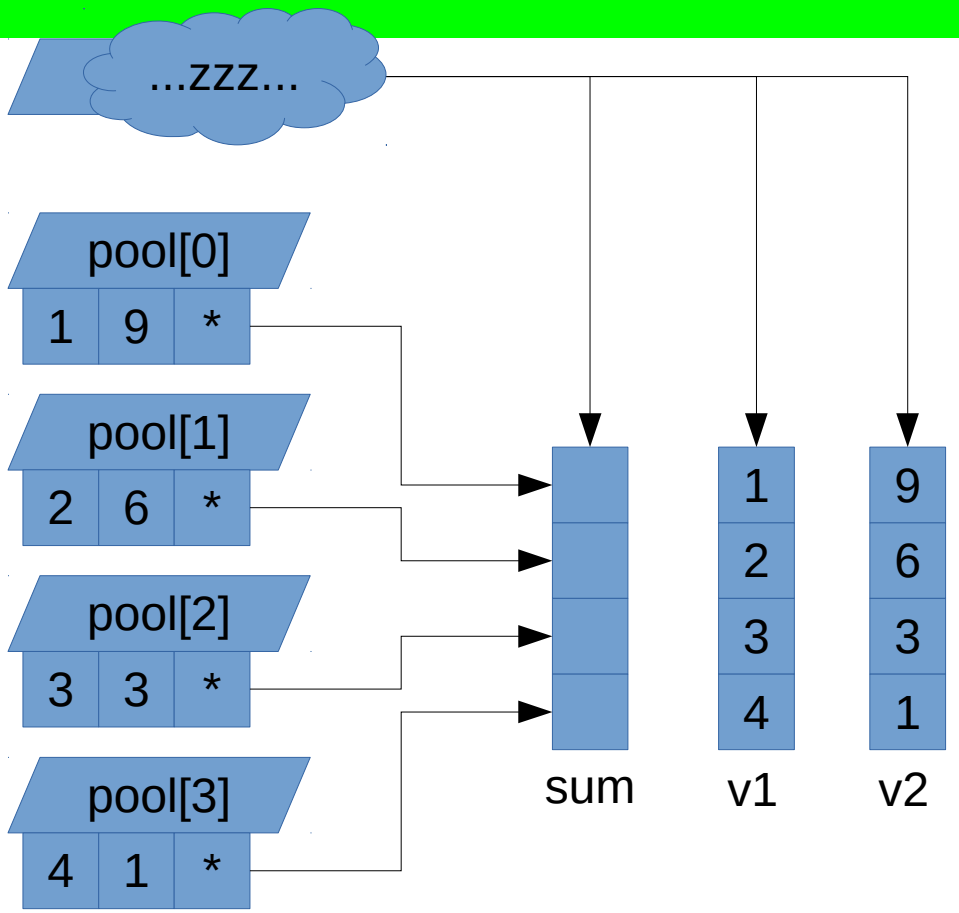


Потоки в C++

```
#include <thread>

void add(int a, int b, int* sum)
{
  *sum = a + b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  int sum[4];
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      add, v1[a], v2[a], &sum[a]
    );
  for (thread t& : pool)
    t.join();
  return 0;
}
```

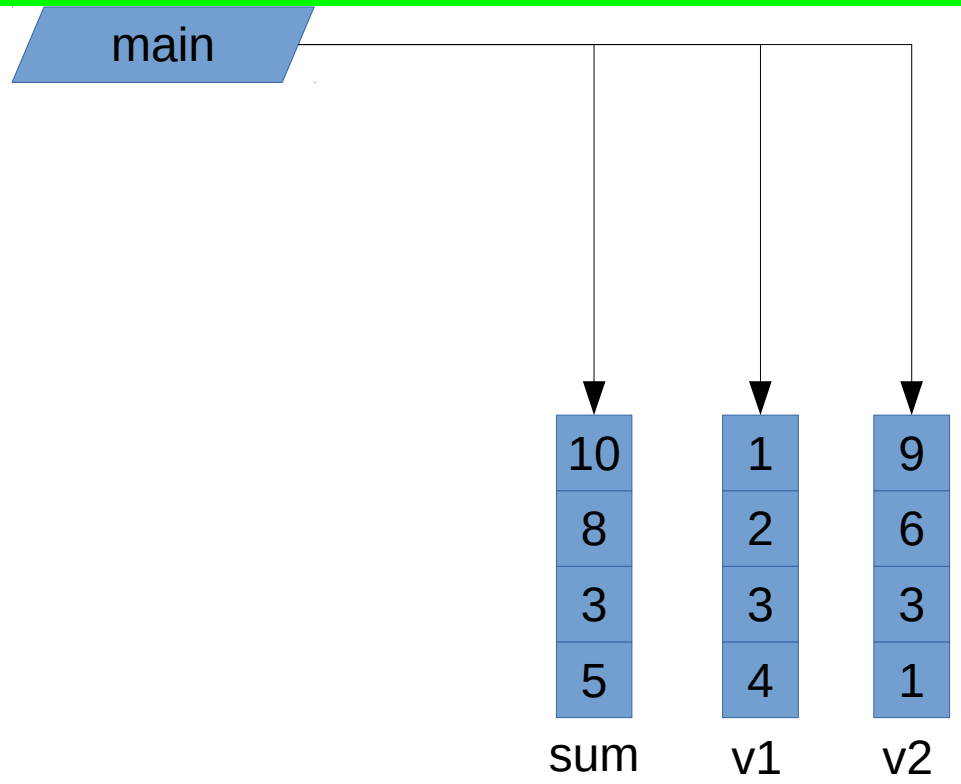


Потоки в C++

```
#include <thread>

void add(int a, int b, int* sum)
{
  *sum = a + b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  int sum[4];
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      add, v1[a], v2[a], &sum[a]
    );
  for (thread t& : pool)
    t.join();
  return 0;
}
```



Примитивы синхронизации

Далеко не всегда удается разделить задачу таким образом, чтобы потоки никак не пересекались при работе с общей памятью. При этом, нельзя предсказать, в какой момент времени поток будет прерван.

При доступе к общим данным из разных потоков применяют **мьютекс** (от англ. MUTually EXclusive).

Блокировка или разблокировка мьютекса — атомарная операция, то есть она либо выполнится целиком, либо не выполнится вообще.

Перед доступом к общему ресурсу поток блокирует мьютекс, а по окончании доступа — разблокирует. Другой поток, пытающийся получить тот же мьютекс, будет ждать его разблокировки.

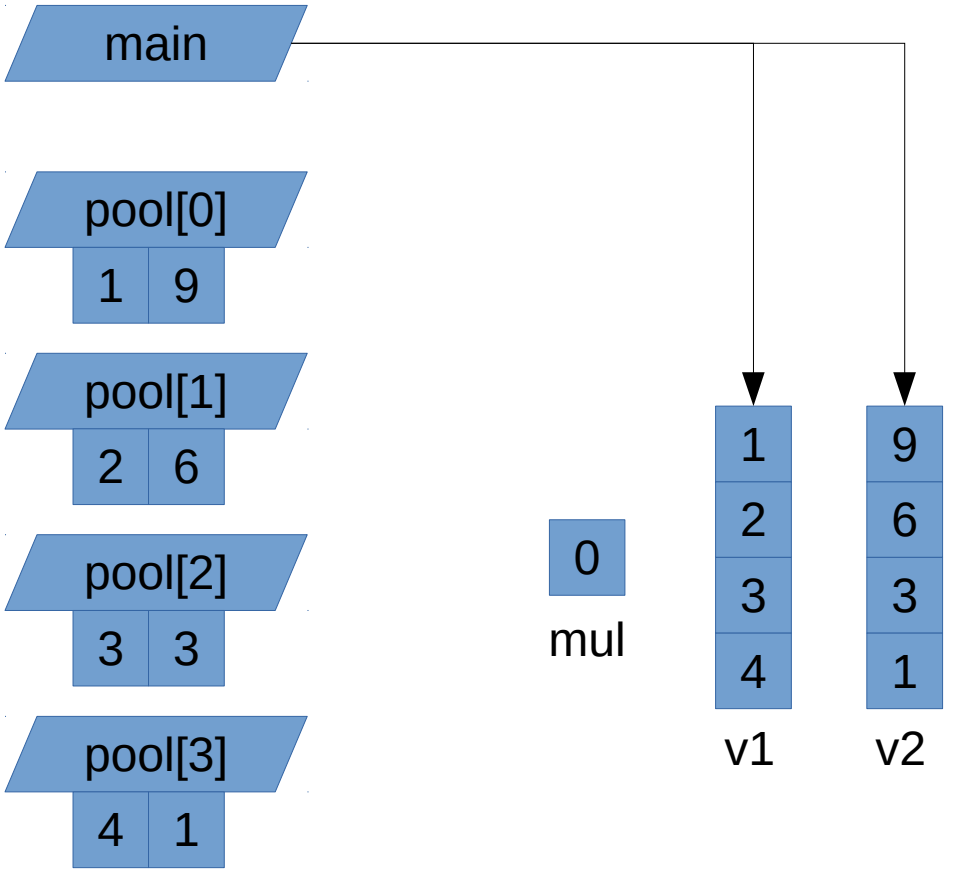
Мьютексы являются частью операционной системы.

Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

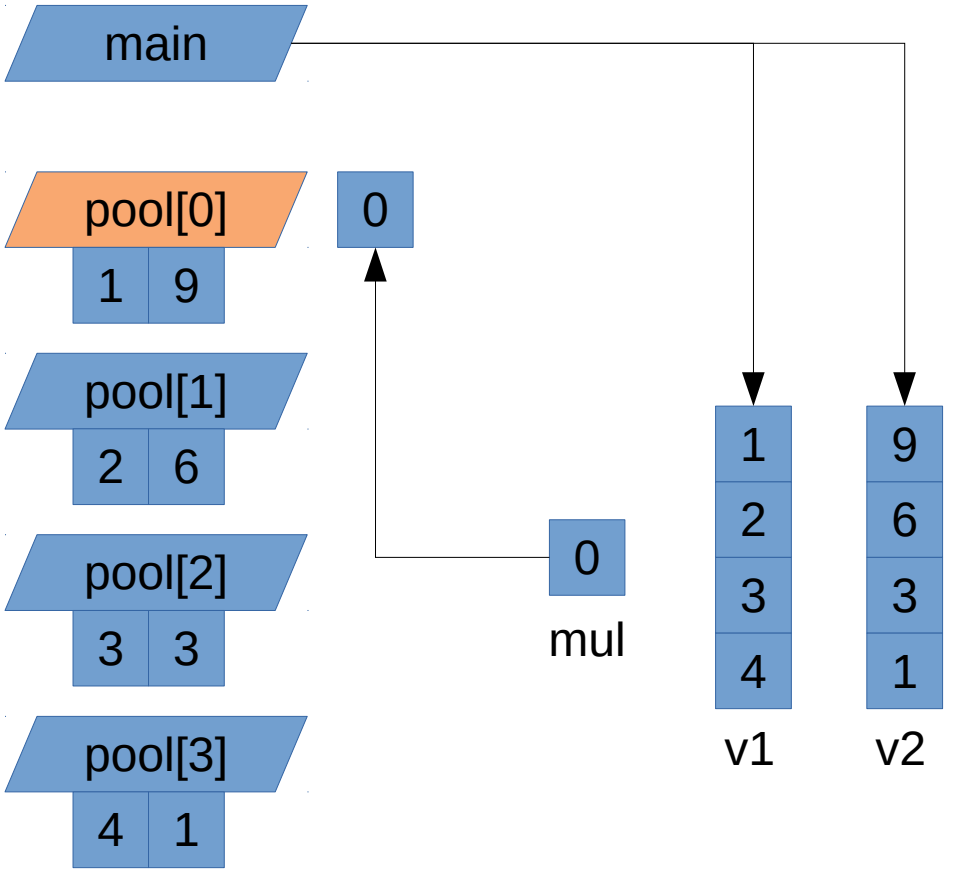


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

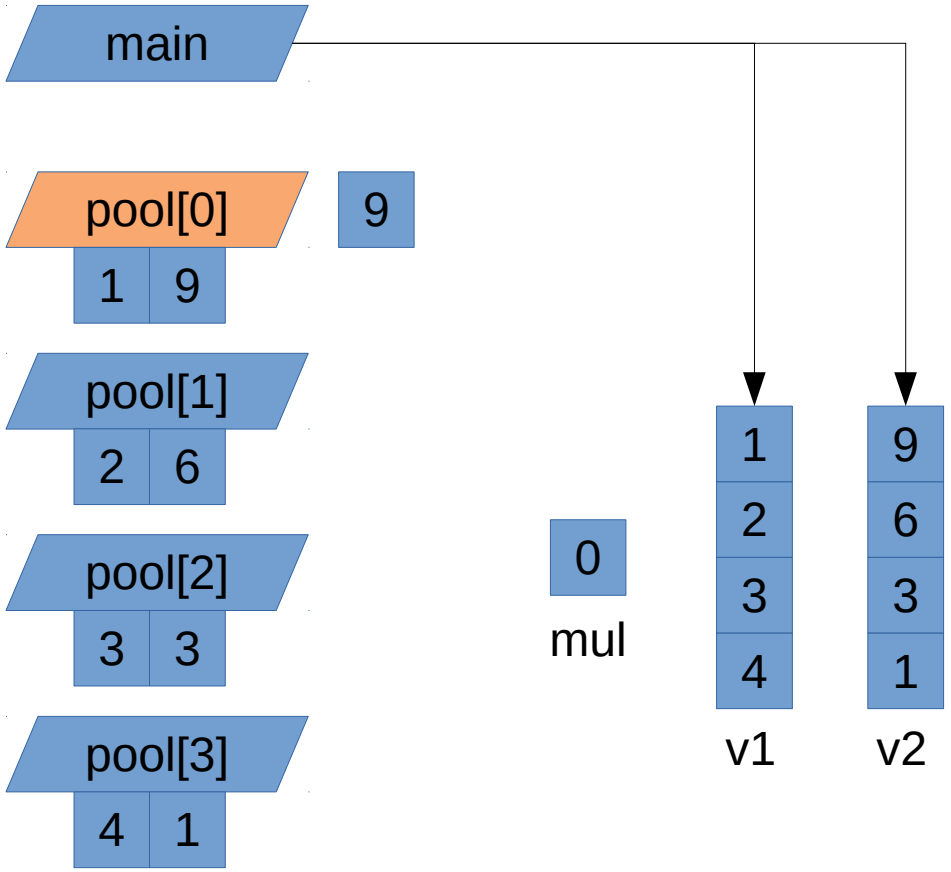


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

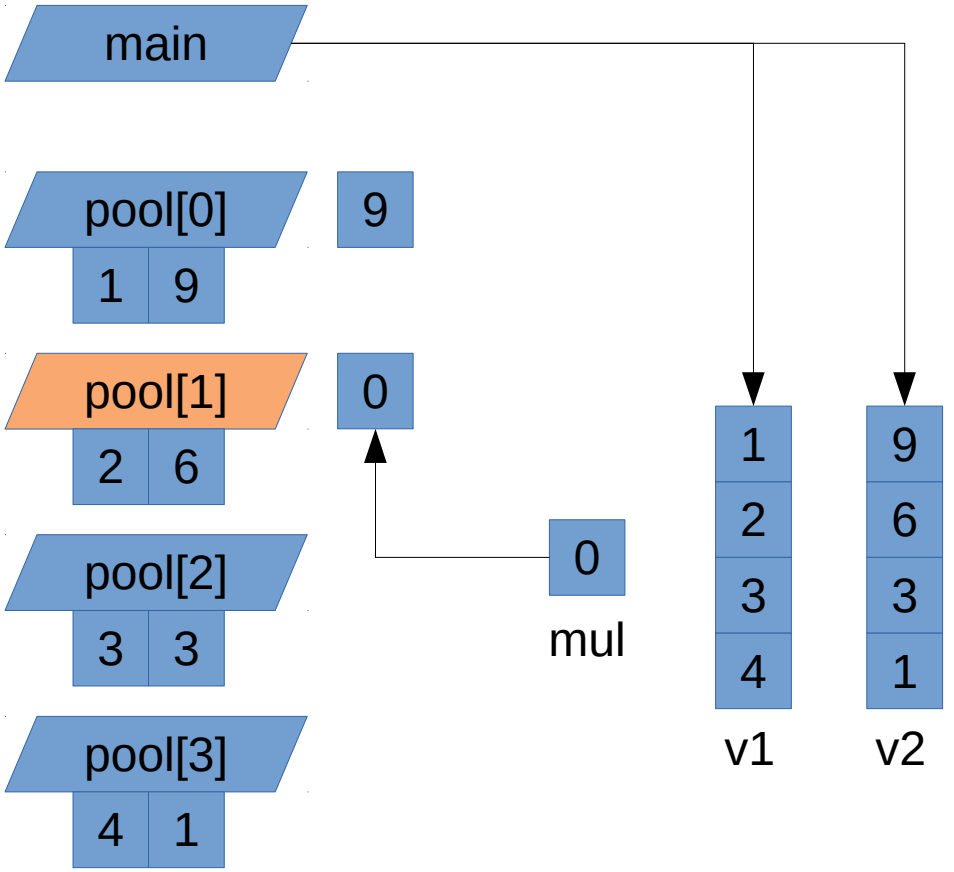


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

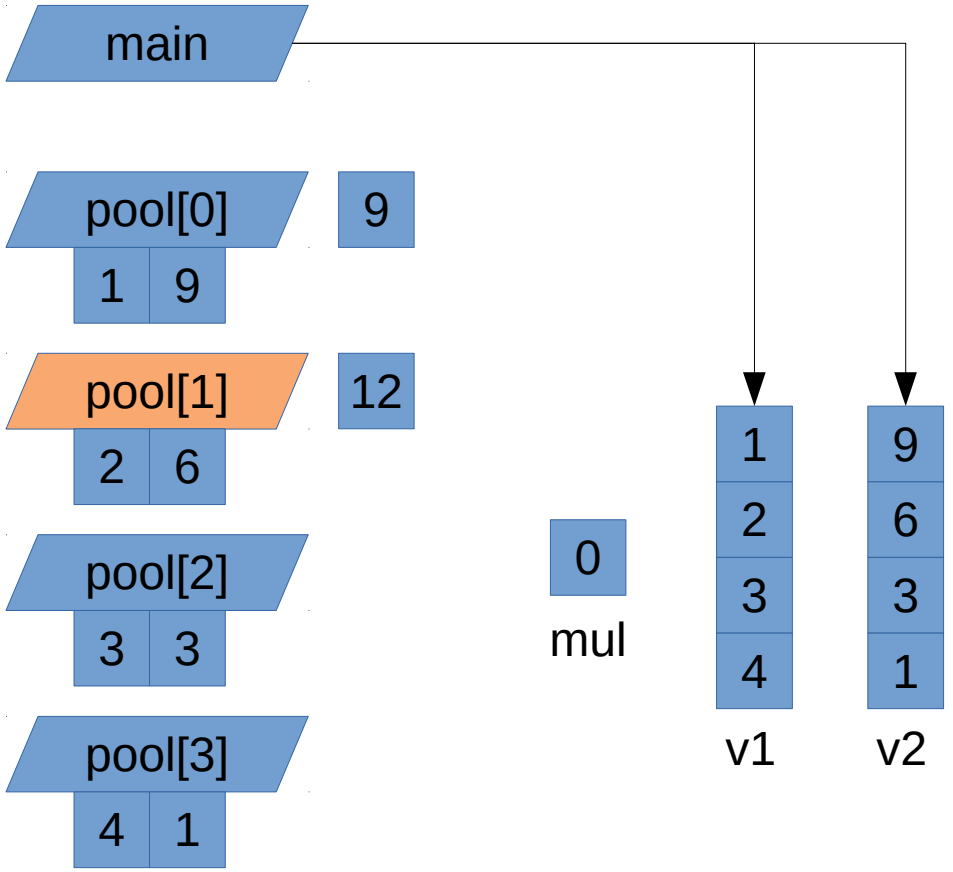


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

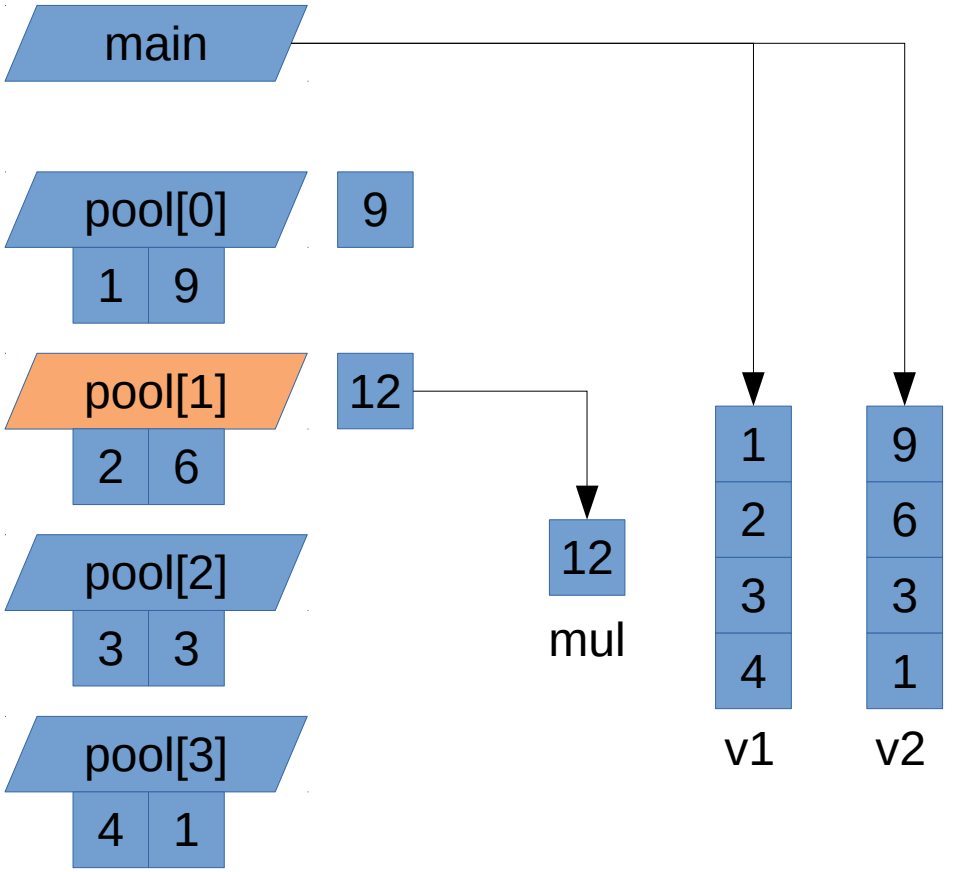


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

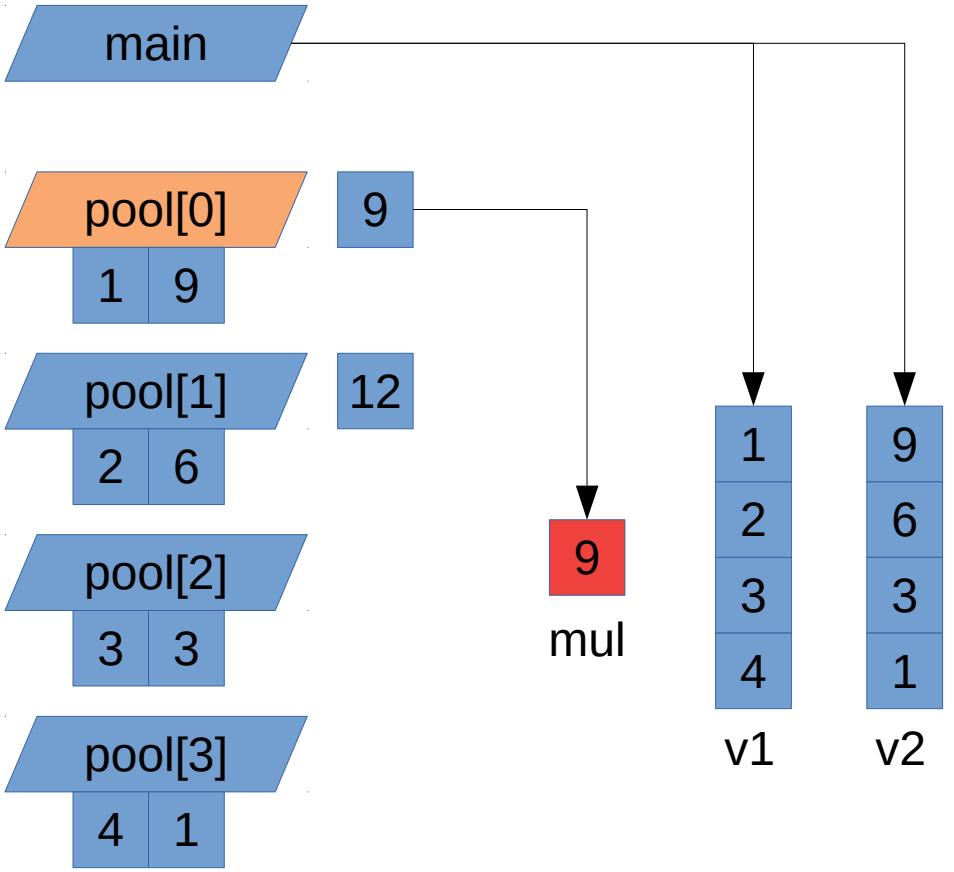


Мьютекс

```
#include <thread>
using namespace std;
int mul = 0;

void smul(int a, int b)
{
  mul = mul + a*b;
}

int main() {
  int v1[4] = {1, 2, 3, 4};
  int v2[4] = {9, 6, 3, 1};
  vector<thread> pool;
  for (int a=0; a<4; a++)
    pool.push_back(thread(
      smul, v1[a], v2[a]));
  for (thread t& : pool)
    t.join();
  return 0;
}
```

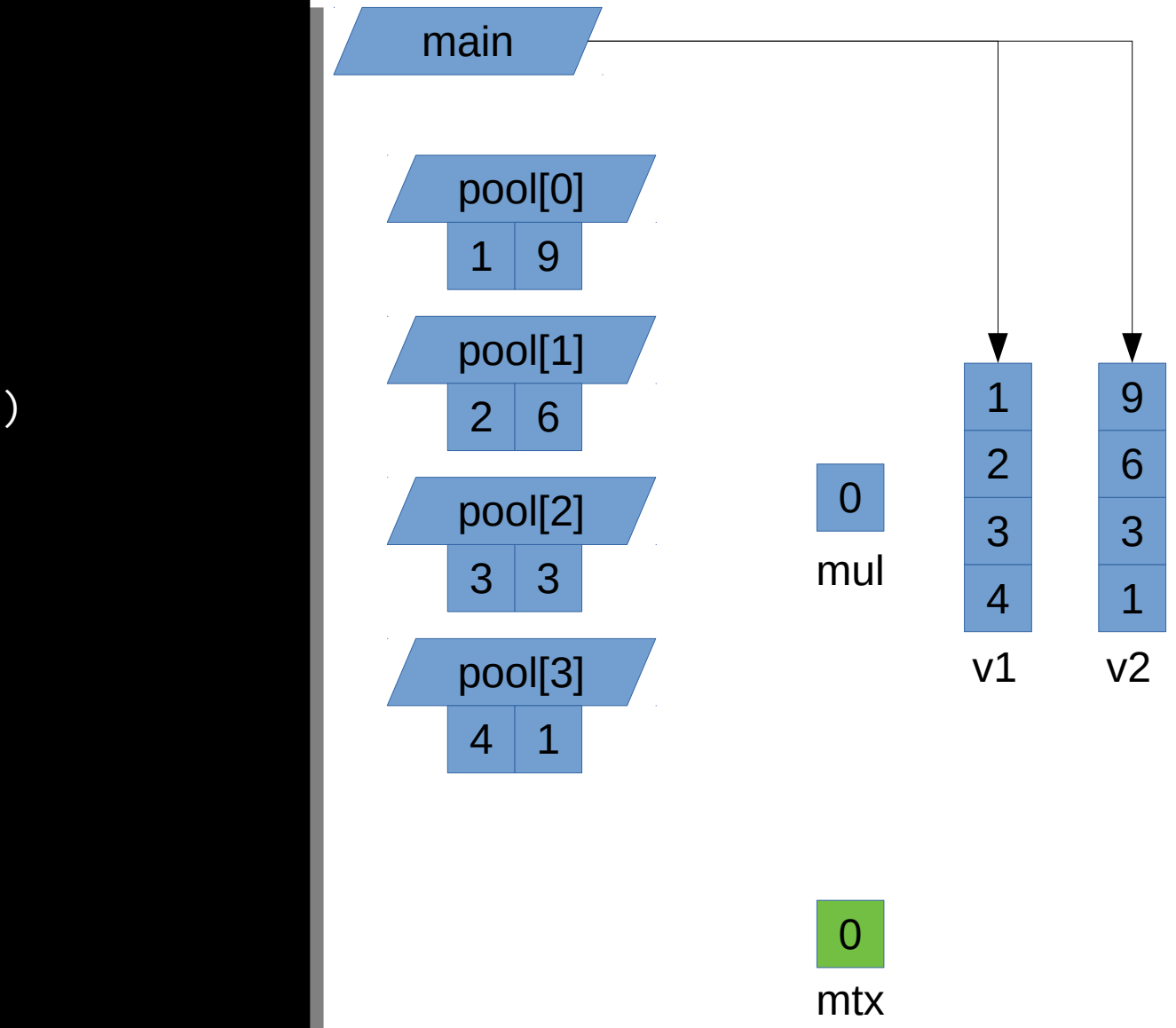


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

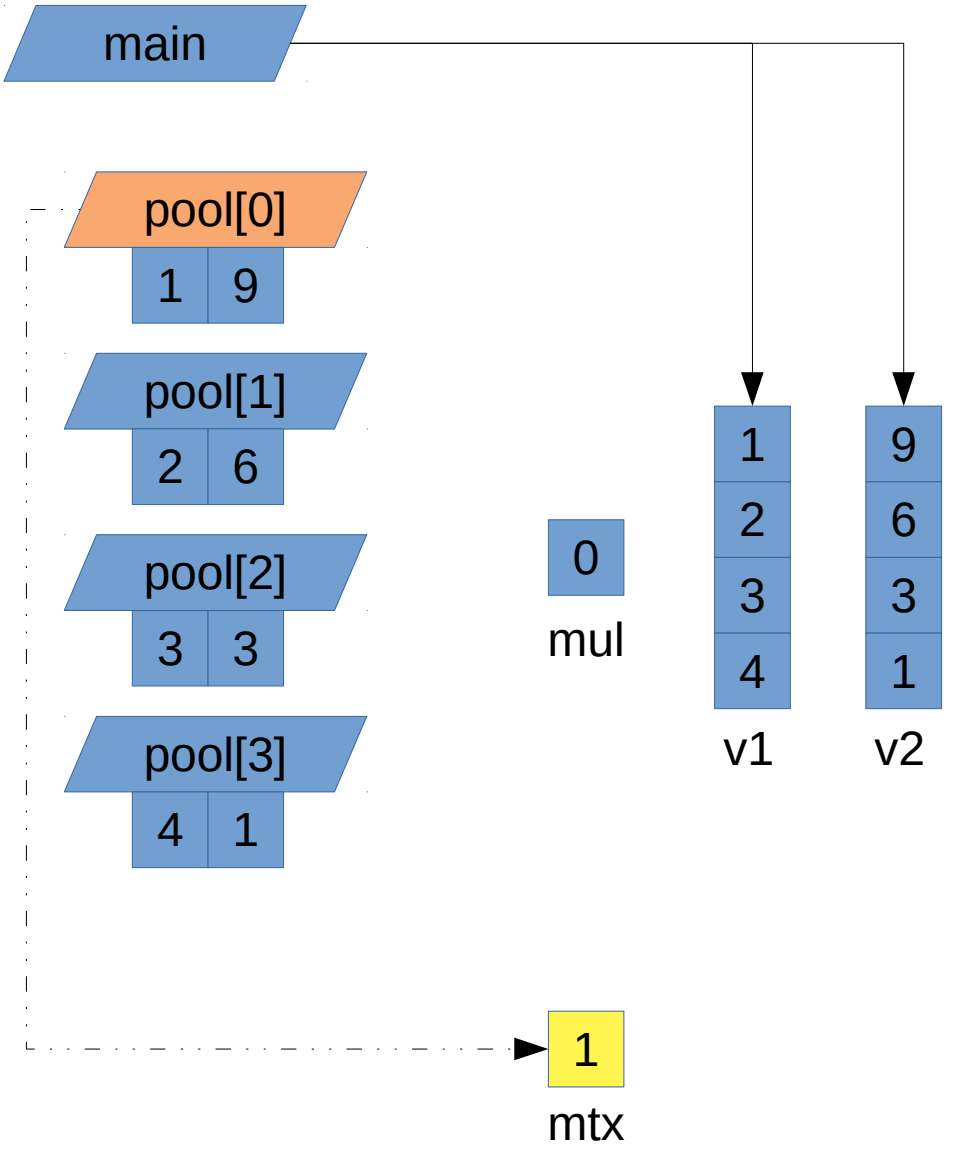


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

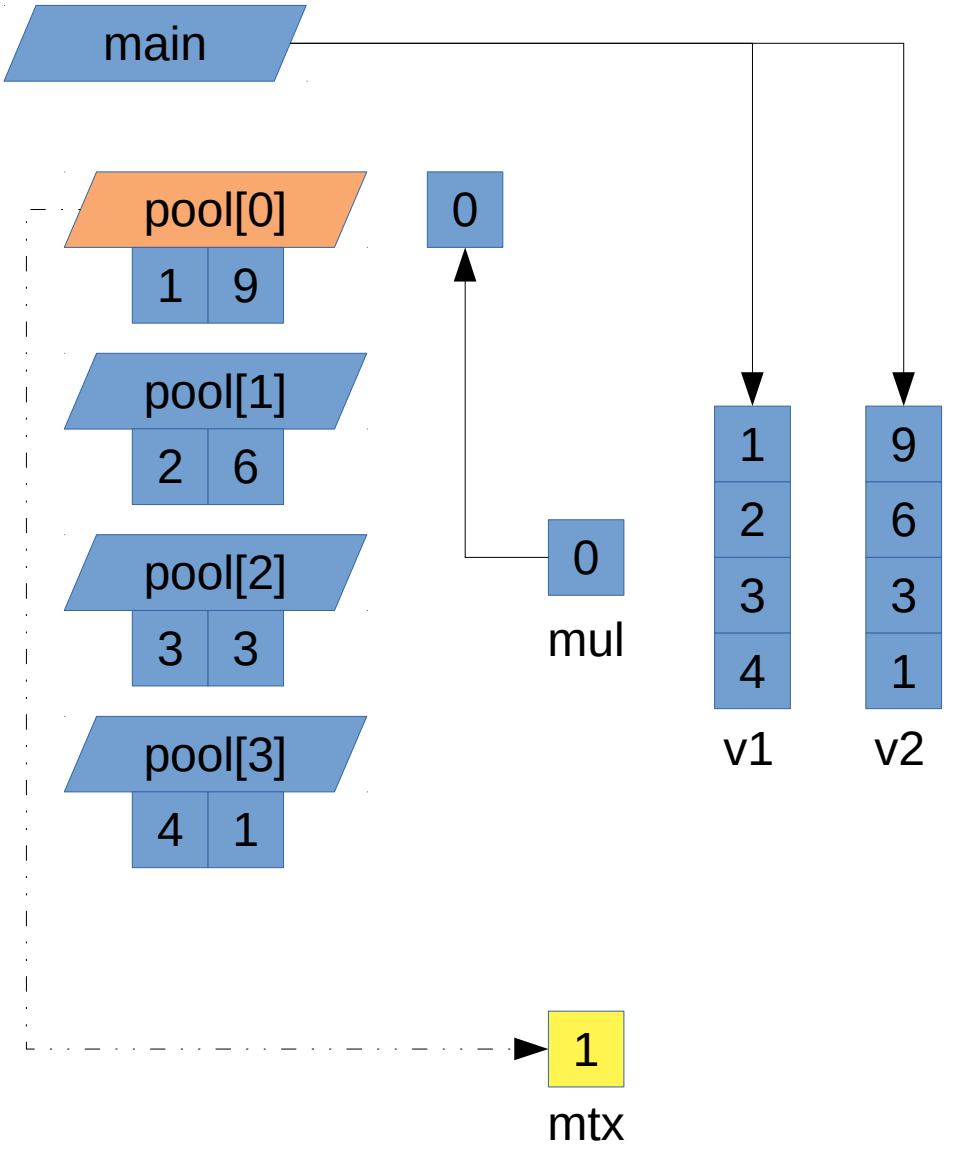


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

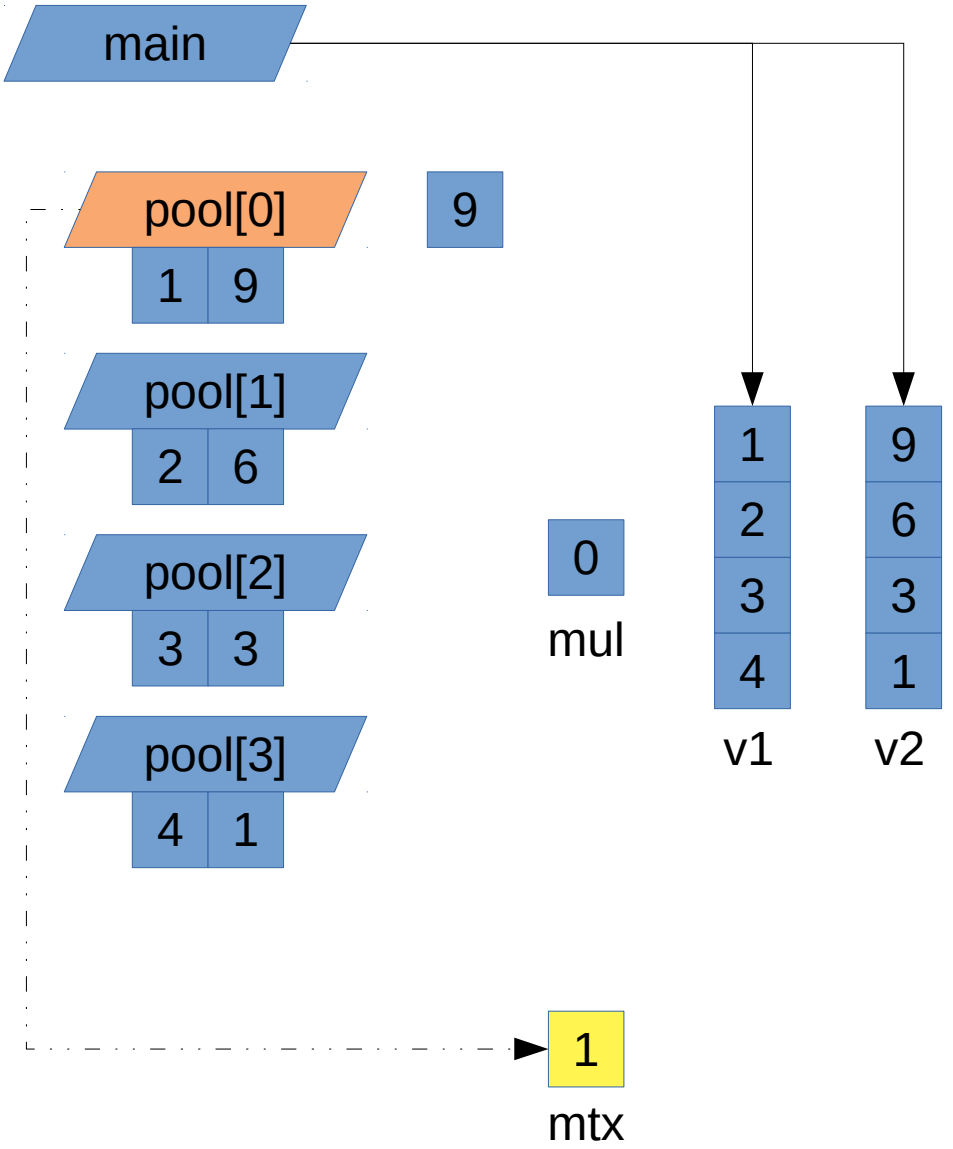


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

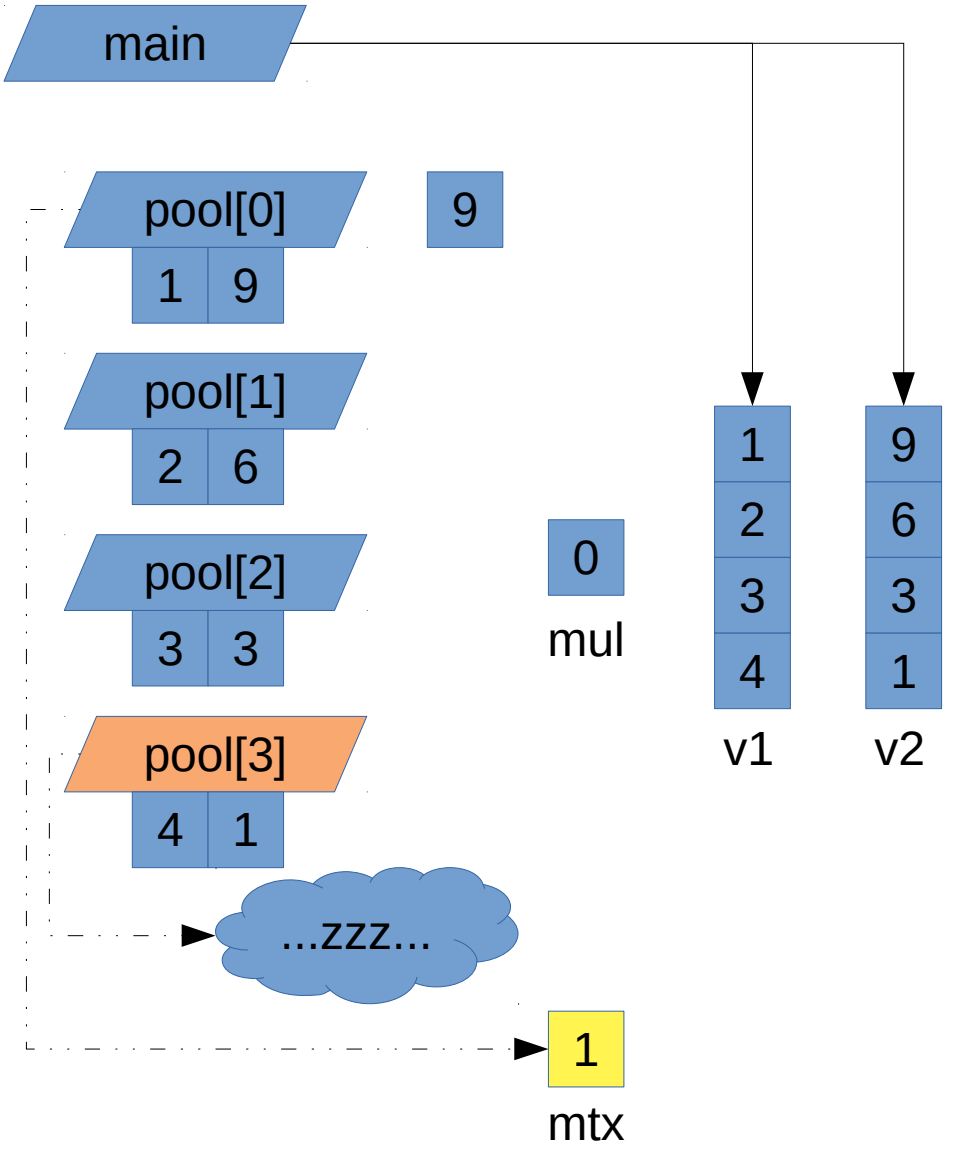


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

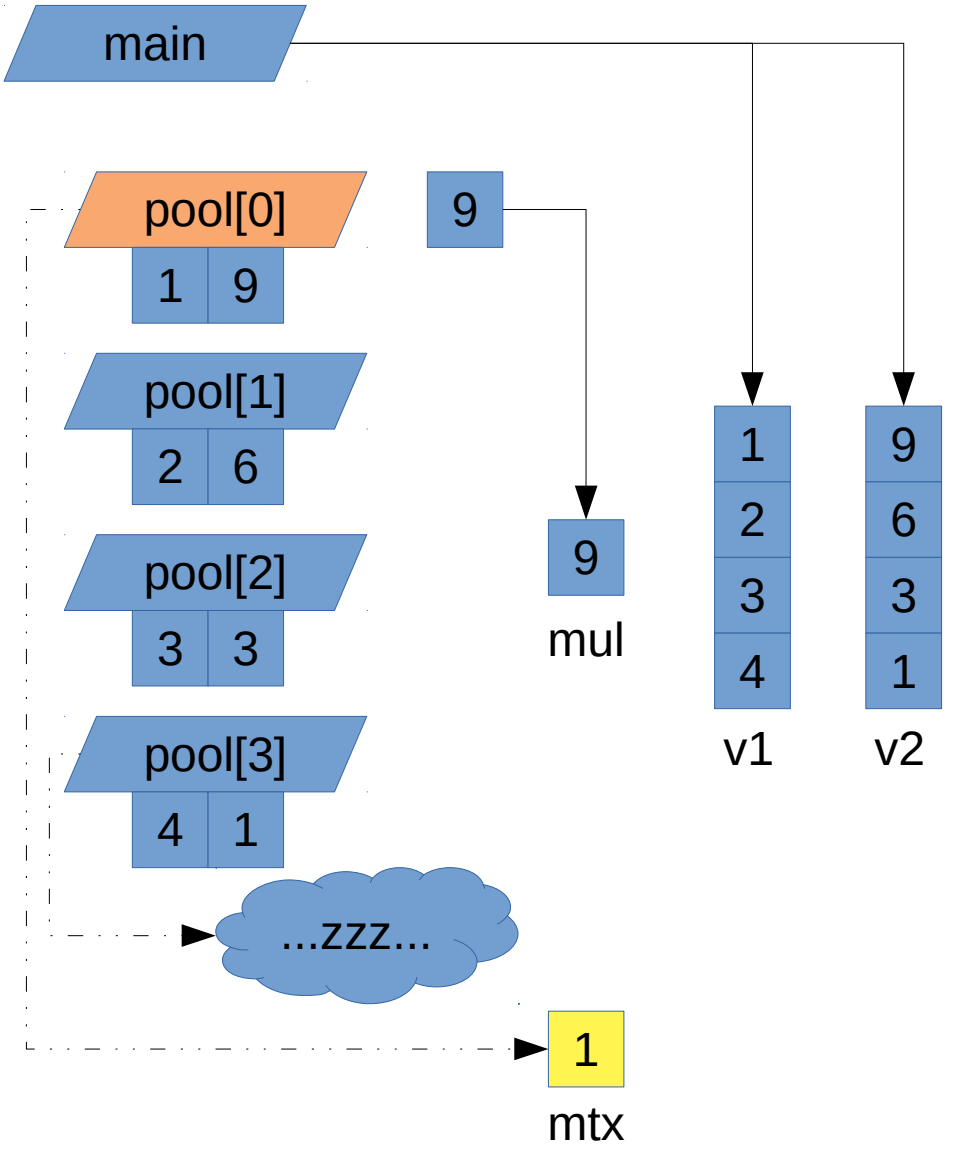


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

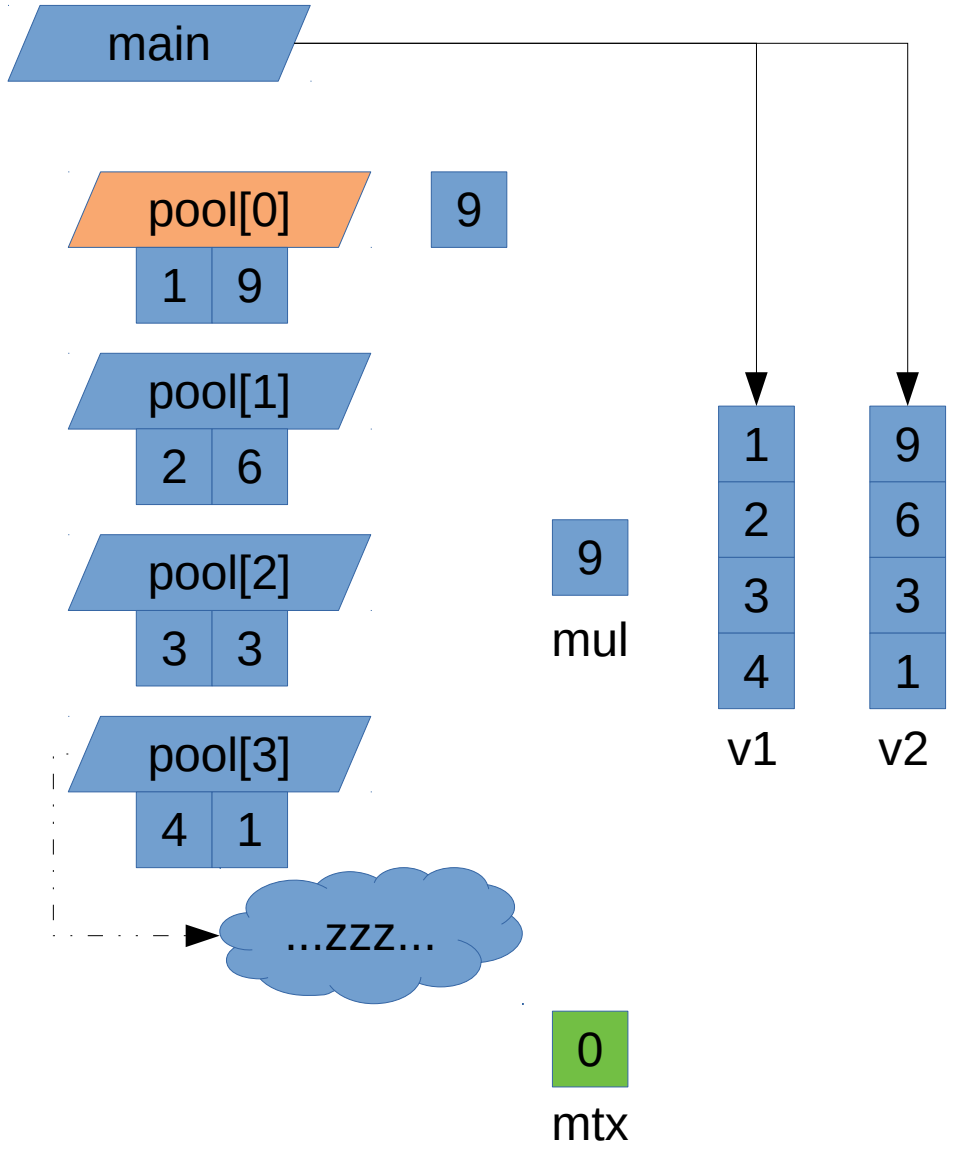


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```

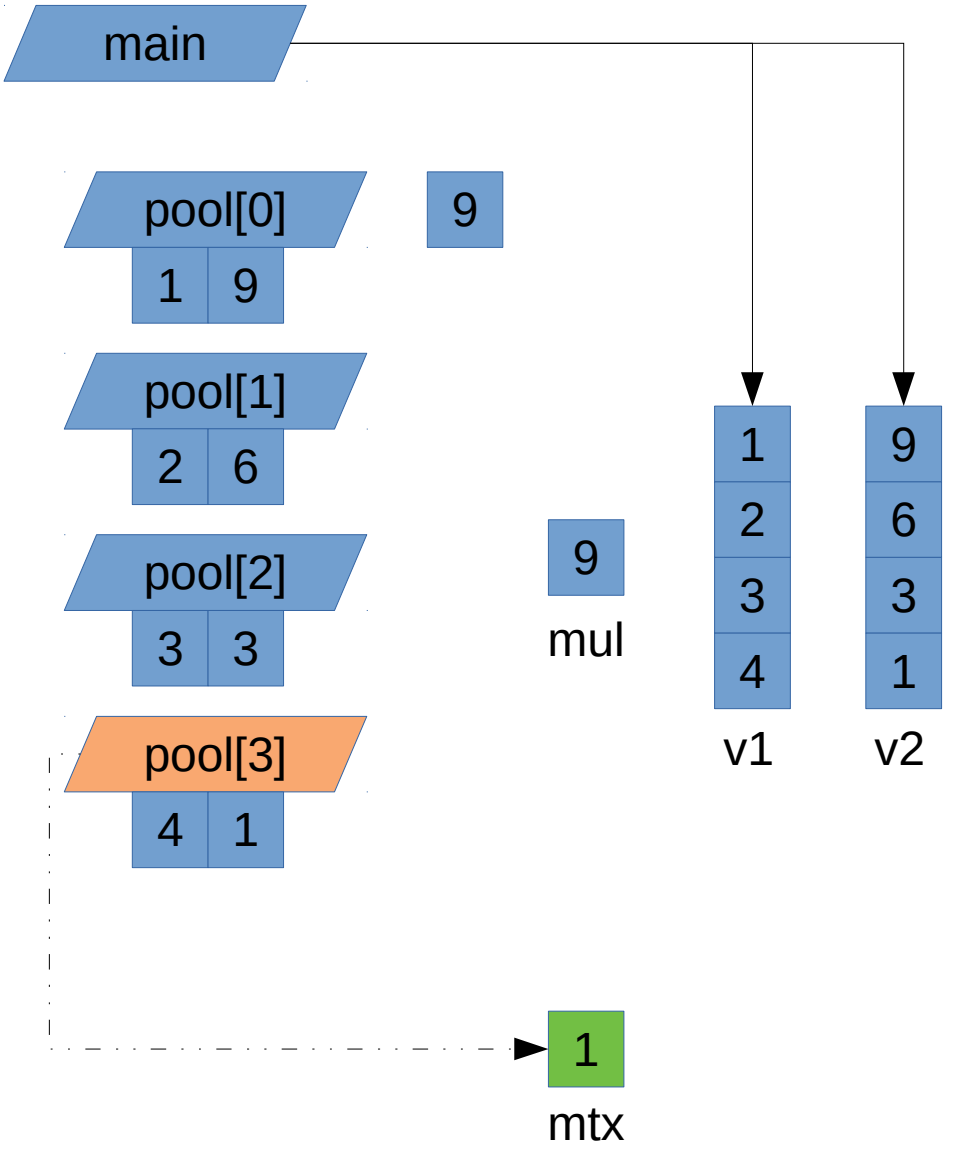


Мьютекс

```
#include <thread>
#include <mutex>
using namespace std;

int mul = 0;
mutex mtx;

void smul(int a, int b)
{
  mtx.lock();
  mul = mul + a*b;
  mtx.unlock();
}
```



Примитивы синхронизации

Семафор — атомарный счетчик, увеличивающий или уменьшающий свое значение на 1. Уменьшение значения семафора ниже 0 невозможно: поток, пытающийся произвести такую операцию, устанавливается в ожидание до тех пор, пока эта операция не станет возможной (другой поток увеличил семафор).

Условная переменная позволяет усыпить поток в ожидании оповещения. Такой поток не потребляет ресурсы до момента оповещения из другого потока.

Критическая секция — более быстрый аналог мьютекса, не использующий ресурсы ядра ОС. В некоторых ОС называется Фьютекс (быстрый мьютекс).

Потоки в Python

Python поддерживает механизм потоков при помощи модуля threading.

```
import threading
```

Для создания потока выполнения необходимо определить функцию и передать ее в качестве аргумента в конструктор класса **threading.Thread**. Также в конструктор можно передать аргументы запускаемой функции в виде кортежа.

```
def thr(sign):
    print(sign)

t = threading.Thread(target=thr, args=("Hello",))
```

Поток не запускается до вызова метода **start**. Метод **join** ждет завершения работы потока.

```
t.start()
t.join()
```

Потоки в Python

Синхронизация потоков выполнения производится при помощи класса threading.Lock. Его работа аналогична мьютексу.

```
lk = threading.Lock()
```

Перед доступом к общим данным необходимо вызвать метод **acquire()** у полученного замка (Lock), чтобы захватить его. По окончании работы с общими данными необходимо освободить замок — вызвать **release**. Другой поток, пытающийся захватить тот же замок, будет поставлен в ожидание до освобождения замка.

```
lk.acquire()
# <...>
lk.release()
```

Потоки в Python

Потоки работают в одном адресном пространстве, и им доступны все глобальные переменные процесса. Программно потоки Python аналогичны потокам C++.

Все потоки в пределах одного сценария Python выполняются одним и тем же интерпретатором в однопоточном режиме, поэтому не могут быть автоматически распределены на разные ядра.

Для использования всех доступных вычислительных ресурсов следует создавать процессы. Например, при помощи модуля **multiprocessing**. Однако, необходима организация межпроцессного взаимодействия.

Ошибки синхронизации

Состояние гонки (англ. Race Condition). Нарушение синхронизации, при котором результат работы программы непредсказуем и зависит от того, какой поток первым приступил к выполнению участка кода.

Ошибки работы с мьютексом. Попытка освободить свободный мьютекс приводит к аварийному завершению работы. Не освобожденный мьютекс может привести к блокировке приложения.

Мертвая блокировка (англ. Deadlock). Возникает, когда поток 1 захватил мьютекс А и ждет мьютекс Б, и в то же время поток 2 захватил мьютекс Б и ждет мьютекс А.

Материалы для проработки

- **MPI** — средство организации распределенных вычислений.
- **OpenCL** — библиотека распределенных вычислений, в том числе с использованием графического процессора.
- **Deadlock Empire** (https://deadlockempire.github.io/) — игра, наглядно демонстрирующая ошибки проектирования многопоточного ПО.