Параллелизм

МГТУ им. Н.Э. Баумана

May 16, 2016

Параллелизм

Одновременное выполнение нескольких операций.

Варианты параллелизма

- Эмуляция
  Переключение задач
  <картинка>
- Аппаратная поддержка
  Несколько процессоров, несколько ядер, возможность на одном ядре параллельно выполнять несколько задач
  - Одно ядро – одна задача
    <картинка>
  - Квантование
    <картинка>

Организация параллелизма

- Многопроцессность
  - + Собственный констекст выполнения
  - + Безопасность адресного пространства
  - - Большие накладные расходы при запуске/переключении задач
  - - Взаимодействие между задачами только через средства межпроцессного взаимодействия (файлы, сокеты, сигналы, ...)
- Многопоточность
  - + Разделяемая память: легкость взаимодействия
  - - Разделяемая память: проблемы синхронизации доступа

Зачем нужен параллелизм

- Разделение обязанностей между независимо выполняющимися последовательностями инструкций
  <картинка>
- Повышение производительности
  Растет не скорость, а количество ядер
  <картинка>

Параллелизм может быть вреден!

- Потоки – ограниченный и сравнительно дорогой ресурс (стандартный стек 1 МБ)
- Затраты на переключение между потоками могут превосходить пользу от распараллеливания задачи
- Затраты на создание потока могут быть выше, чем затраты на решение задачи

Поддержка многопоточности

C++11

- Потоки исполнения (std::thread)
- Механизмы синхронизации (std::mutex, std::lock_guard<>, std::conditional_variable)
- Механизмы доступа к разделяемым данным (std::atomic<>)
- Механизмы поддержки асинхронных операций (std::async<>, std::promise<>, std::future<>)

Hello, world

```cpp
#include <iostream>
#include <thread>

void helloWorld() {
  std::cout << "Hello, world!" << std::endl;
}

int main() {
  std::thread t(helloWorld);

  t.join();

  return 0;
}
```

Запуск потока – RAII

```
1  void hello(const std::string& who) {
2    std::cout << "Hello, " << who <<std::endl;
3  }
4
5  void helloWorld() {
6    std::cout << "Hello, world!" << std::endl;
7  }
8
9  void f() {
10   char iu[] = {'I', 'U', '\0'};
11   std::thread t(hello, iu); // ERROR: UNDEFINED BEHAVIOR!
12   // ...
13 }
14
15 int main() {
16   std::thread t0(helloWorld); // no argument
17   std::thread t1(hello, std::string("IU8")); // arguments transition
18   std::thread t2(std::move(t1)); // move into t2
19
20   std::thread t3; // thread object without thread
21   t3 = std::move(t2); // move-operator=
22   // ...
23 }
```

Завершение потока

```
1  void helloWorld() {
2    std::cout << "Hello, world!" << std::endl;
3  }
4
5  void bad() {
6    std::thread t(helloWorld);
7    // std::terminate() when t is destructed!
8  }
9
10 void good() {
11   std::thread t(helloWorld);
12   if(t.joinable()) // true
13     t.join(); // wait till thread stops
14   if(t.joinable()) // false
15     t.join();
16 }
17
18 int main() {
19   good();
20   std::thread t(helloWorld);
21   t.detach(); // do not wait till thread stops
22 }
```

Еще методы

```
int main() {
  std::map<std::thread::id, std::thread> threadMap; // !
  uint8_t buf[100000];
  size_t range = sizeof(buf);
  size_t threadCount = std::thread::hardware_concurency(); // !
  if(threadCount == 0)
    threadCount = 2;
  size_t threadPart = range / threadCount;
  for(size_t i = 0; i < threadCount; ++i) {
    std::thread t(doThis, buf, i*threadPart, (i+1)*threadPart);
    threadMap.emplace(t.get_id(), std::move(t)); // !
  }

  for(auto& t: threadMap)
    t.second.join();
}
```

Добавление элемента в список: состояние гонок

```
1  void list_add_element(el* previous, el* newEl) {
2    // list is consistent here
3    el* next = previous->next;
4    previous->next = newEl; // unconsistent
5    newEl->prev = previous; // unconsistent
6    newEl->next = next; // unconsistent
7    next->prev = newEl; // consistent
8
9    // list is consistent here
10 }
11
12 void add2elements(el* previous, el* el1, el* el2) {
13   std::thread t1(list_add_element, previous, el1);
14   std::thread t2(list_add_element, previous, el2);
15   // DATA RACE!
16   t2.join();
17   t1.join();
18 }
```

Mutex: защита от одновременного выполнения

Mutex

mutual exclusion = взаимное исключение
Механизм синхронизации, предполагающий операции:

- lock – захват мьютекса потоком выполнения
  Если мьютекс захвачен, никто другой его захватить не может.
  Захват мьютекса задействует барьер по памяти (memory barrier)
- unlock – освобожение мьютекса потоком выполнения

std::mutex

```
1  void synchronized_list_add_el(std::mutex m, el* previous, el* newEl) {
2    m.lock();
3    list_add_element(previous, newEl);
4    m.unlock();
5  }
6
7  void add2elements(el* previous, el* el1, el* el2) {
8    std::mutex m;
9    std::thread t1(synchronized_list_add_el, m, previous, el1);
10   std::thread t2(synchronized_list_add_el, m, previous, el2);
11   t2.join();
12   t1.join();
13 }
```

std::mutex: практика использования

std::lock_guard<>

```
1  void synchronized_list_add_el(std::mutex& m, el* previous, el* newEl) {
2    std::lock_guard<std::mutex> lock(m); // m is locked here!
3    list_add_element(previous, newEl);
4    // lock is destructed -> m is unlocked
5  }
6
7  void add2elements(el* previous, el* el1, el* el2) {
8    std::mutex m;
9    std::thread t1(synchronized_list_add_el, m, previous, el1);
10   std::thread t2(synchronized_list_add_el, m, previous, el2);
11   t2.join();
12   t1.join();
13 }
```

Взаимоблокировка

```
1  void synchronized_f(mutex& m1, mutex& m2) {
2    lock_guard<mutex> l1(m1);
3    lock_guard<mutex> l2(m2);
4    doSmth();
5  }
6
7  void synchronized_g(mutex& m1, mutex& m2) {
8    lock_guard<mutex> l2(m2);
9    lock_guard<mutex> l1(m1);
10   doSmthElse();
11 }
12
13 void f() {
14   std::mutex m1, m2;
15   std::thread t1(synchronized_f, m1, m2);
16   std::thread t2(synchronized_g, m1, m2);
17   // DEADLOCK
18   t2.join();
19   t1.join();
20 }
```

Как избежать?

1. Захватывать мьютексы в одинаковом порядке 2. std::lock() – захват всех переданых объектов

```
1  void synchronized_f(mutex& m1, mutex& m2) {
2    std::lock(m1, m2); // locks both mutexes
3    std::lock_guard<mutex> l1(m1, std::adopt_lock); // not locking
4    std::lock_guard<mutex> l2(m2, std::adopt_lock);
5    doSmth();
6  }
7
8  void synchronized_g(mutex& m1, mutex& m2) {
9    std::lock(m1, m2);
10   std::lock_guard<mutex> l1(m2, std::adopt_lock); // not locking
11   std::lock_guard<mutex> l2(m1, std::adopt_lock);
12   doSmth();
13 }
```

3. Избегать взаимного ожидания потоков

std::atomic

std::atomic<bool>

Конструктор преобразования и оператор присваивания.

```
std::atomic<bool> b(true);
b = false; // returns value, not reference
```

Методы std::atomic:

```
void f(std::atomic<bool>& b) {
  bool value = b.load(); // read value
  b.store(false); // set value to false
  bool prevState = b.exchange(true); // returns false and sets true
}
```

compare_exhange

```
void f(std::atomic<int>& ai) {
  ai = 2;
  int test_val = 5;
  // if ai!=test_val ==> test_val = ai
  bool exchanged = ai.compare_exchange_strong(test_val, 10); // false
  // if ai==test_val ==> ai = new_val
  exchanged = ai.compare_exchange_strong(test_val, 10); // true

  // compare_exchange_weak -- better performance
  // May yield *obj != *expected, though they are equal
  // must be done in loop
}
```

Ожидание события или условия

```cpp
void longTask(std::atomic<bool> ready) {
  // ...
  ready.store(true);
}
```

Ждем готовности:

```cpp
int main() {
  std::atomic<bool> ready = false;
  std::thread(longTask, ready);

  while(!ready.load());
  // ...
}
```

Даем потоку поработать:

```cpp
int main() {
  std::atomic<bool> ready = false;
  std::thread t(longTask, ready);

  // busy waiting
  while(!ready.load()) {
    std::this_thread::sleep_for(std::chrono::milliseconds(100));
  }
  // ...
}
```

Ожидание события или условия

std::conditional_variable: notify_one/notify_all

```
void f(std::mutex& m, std::conditional_variable& cv, bool& ready) {
  // ...
  ready = true;

  std::lock_quard<std::mutex> lck(m);
  cv.notify_one();
}
```

Ждем готовности (wait/wait_for/wait_until):

```
int main() {
  std::mutex m;
  std::conditional_variable cv;
  bool ready = false;

  std::unique_lock<std::mutex> lck(m); // m is locked
  std::thread t(f, std::ref(m), std::ref(cv), std::ref(ready));
  while (!ready) {
    cv.wait(lck); // mutex is unlocked
    // woken on notify
  }

  t.join();
}
```

Конец