Моделирование и анализ данных
2024. Том 14. № 3. С. 135–148
DOI: https://doi.org/10.17759/mda.2024140308
ISSN: 2219-3758 (печатный)
ISSN: 2311-9454 (online)
© 2024 ФГБОУ ВО МГППУ

Modelling and Data Analysis
2024. Vol. 14, no. 3, pp. 135–148
DOI: https://doi.org/10.17759/mda.2024140308
ISSN: 2219-3758 (print)
ISSN: 2311-9454 (online)
© 2024 Moscow State University of Psychology & Education

УДК 004.416.6

Разработка и внедрение многопоточного сервиса распознавания речи с помощью связки C#, C++ и Python

Левонович Н.И.*
Московский государственный психолого-педагогический университет (ФГБОУ ВО МГППУ), г. Москва, Российская Федерация
ORCID: https://orcid.org/0000-0002-8580-0490
e-mail: levonikitatech@yandex.ru

В статье рассматривается особенности реализации многопоточных промышленных систем, реализующих научные вычисления с помощью средств, доступных в языке программирования Python. Статья содержит описание теоретических аспектов, таких как работа механизма глобальной блокировки интерпретатора (GIL), архитектура управления зависимостями, библиотека параллелизма, основанного на процессах. В практическая часть статьи посвящена реализации многопоточного сервиса распознавания речи, который использует взаимодействие процессов через разделяемую память, на базе библиотеки «boost.interprocess». В результате внедрения описанной в статье архитектуры в конкретном случае удалось существенно снизить нагрузку на процессор.

Ключевые слова: программная инженерия, Python, C++, C#, boost, многопоточные приложения, распознавание речи.

Благодарности. Автор благодарит за помощь в сборе данных о внутреннем устройстве интерпретатора Python специалиста по отношениям с разработчиками Evrone Григория Петрова.

Для цитаты:
Левонович Н.И. Разработка и внедрение многопоточного сервиса распознавания речи с помощью связки C#, C++ и Python // Моделирование и анализ данных. 2024. Том 14. № 3. С. 135–148. DOI: https://doi.org/10.17759/mda.2024140308

Левонович Никита Ильич, студент магистратуры, младший научный сотрудник, Московский государственный психолого-педагогический университет (ФГБОУ ВО МГППУ), г. Москва, Российская Федерация, ORCID: https://orcid.org/0000-0002-8580-0490, e-mail: levonikitatech@yandex.ru

1. ВВЕДЕНИЕ

В современном мире очень востребовано программное обеспечение, способное проводить научные расчеты. Для реализации подобного программного обеспечения часто используется язык программирования Python, так как он содержит множество библиотек позволяющих реализовать научные вычисления. Также Python позволяет быстро и интерактивно пробовать различные методы решения одной и той же задачи. Однако после выбора окончательного решения исследователь или сотрудничающие с ним инженеры неизбежно сталкиваются с необходимостью разработки промышленного решения. При разработке решений промышленного уровня необходимо иметь в виду некоторые особенности языка Python.

2. ОСОБЕННОСТИ ЯЗЫКА PYTHON

Язык Python имеет особенности, которые могут осложнить внедрение разработанного программного кода в промышленные системы:

- интерпретируемость;
- недостатки архитектуры работы с зависимостями;
- механизм глобальной блокировки интерпретатора (далее GIL).

Рассмотрим подробнее данные особенности. Для того чтобы запустить приложение, реализованное на языке Python, на целевой машине необходимо иметь интерпретатор и пакеты, от которых зависит приложение[1]. Специально для управления пакетами вместе с интерпретатором Python устанавливается система управления программными пакетами, написанными на Python – pip, и средство управления виртуальным окружением Python – virtualenv[1].

Virtualenv [2] – средство для создание изолированного окружения Python. Основная проблема, которую решает данное средство, связана с зависимостями и версиями, а также, косвенно, с разрешениями. Предположим, что нужно установить приложение, которому требуется версия LibFoo 1, и приложение, которому требуется версия 2. Если устанавливать все на системный python (например, python3.12), то можно столкнутся со следующими проблемами: отсутствует возможность при импорте указать версию библиотеки, отсутствует возможность установить несколько версий одной библиотеки. Вторая проблема, которую решает данное средство – зависимость приложения от изменения системного Python. Например, у распространяемого приложения может не быть разрешений на установку зависимостей системного Python или их установка может привести к нарушению функционирования иных приложений. Для решения данных проблем используются изолированные окружения. Изолированное окружение содержит интерпретатор и все необходимые зависимости для запуска Python приложения на конкретном типе систем (Windows/macOs/Linux) и их разрядности.

[1] Такой подход является «классическим», но не единственным существуют средства для упаковки интерпретатора вместе с приложением, например PyInstaller [3].

GIL [4] – механизм, присутствующий в эталонной реализации Python – CPython. Он обеспечивает безопасную работу с потоками, путем установления ограничения – в конкретный момент времени выполнять байт-код Python может лишь один поток операционной системы (в рамках одного процесса).

При запуске приложения Python первым делом стартует главный поток (поток ОС), который инициализирует интерпретатор, затем компилирует Python-код в байт-код и входит в цикл выполнения байт-кода. Для того, чтобы появился Python-поток, с потоком операционной системы связывается структура, которая содержит состояние Python-потока.

Цикл выполнения байт-кода – бесконечный цикл, содержащий больших размеров оператор switch, который обрабатывает всевозможные инструкции байт-кода, для входа в этот цикл поток должен удерживать GIL (что делает главный поток с момента своей инициализации). В начале каждой итерации цикла выполнения байт-кода поток проверяет, есть ли причины освободить GIL. Если в коде Python создан новый Python-поток, он попытается захватить GIL. Если это невозможно (GIL занят), то поток будет ожидать в течение фиксированного временного интервала, называемого интервалом переключения. Если GIL по-прежнему занят, то он пошлет запрос на принудительное освобождение GIL. Если в начале очередной итерации цикла поток, владеющий GIL, увидит запрос на принудительное освобождение GIL, то он освободит GIL, и другой поток захватит GIL. В связи с этим распределение активности потоков при выполнении многопоточного приложения на Python, является таковым, что в один момент времени активен только 1 поток (рисунок 1).

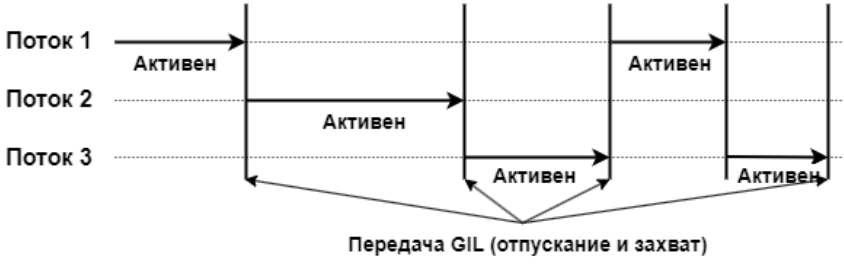

Рис. 1. Распределение активности потоков при выполнении многопоточного приложения на Python

Данный подход эффективно решает проблему одновременного доступа к памяти, среди потоков Python процесса, жертвуя возможностью действительно параллельного выполнения кода. Необходимость GIL обусловлена тем, что потоки операционной системы могут засыпать и просыпаться в самый неожиданный момент, реализовать простой и быстрый сборщик мусора и простой и быстрый доступ разных потоков к общим изменяемым данным (контейнерам, объектам и т.п). Несмотря на это

некоторые библиотеки для научных расчетов умеют отдельно работать с GIL отпуская его при выполнении расчетов, однако установление этого факта требует детального анализа отдельных операций.

3. МНОГОПОТОЧНОСТЬ И ПАРАЛЛЕЛИЗМ, ОСНОВАННЫЙ НА ПРОЦЕССАХ

Одним из решений проблемы параллельного программирования на Python является модуль multiprocessing [5]. Данный модуль позволяет запускать несколько процессов интерпретатора Python (каждый из который имеет свой GIL), в удобном для программиста режиме, предоставляя примитивы синхронизации, каналы коммуникации и межпроцессные коллекции. Примитивы синхронизации: Lock, Recursive Lock, Condition, Semaphore, Event, Timer, Barier – являются стандартным набором примитивов синхронизации потоков. Для коммуникации между процессами используются каналы (Pipe) и очереди (Queue). Канал представляет область памяти доступную связанным процессам, родительскому и дочернему. Данные в канале организованы по принципу FIFO, как только данные прочитаны из канала, они удаляются из него. Межпроцессные очереди, основаны на каналах, они предоставляют интерфейс очереди, для передачи объектов через каналы (добавляя примитивы синхронизации). При передаче объектов в межпроцессной очереди, используется pickle [6] – бинарный формат упаковки Python объектов.

Таким образом в рамках Python приложения можно успешно управлять потоками и процессами. В документации есть примеры использования данного функционала, в качестве работы более подробно рассматривающий данный модуль можно привести книгу «High Performance Python: Practical Performant Programming for Humans» [7].

4. КОММУНИКАЦИЯ PYTHON С ДРУГИМИ ПРИЛОЖЕНИЯМИ

После разработки вычислительных приложений на языке Python, может возникнуть необходимость их интеграции в промышленные системы, реализованные на других языках. Здесь есть два принципиальных подхода: Python приложение, как отдельный сервис с сетевым доступом; встраивание Python через CPython.

Первый способ применим в ситуациях, когда взаимодействие между приложениями можно свести к парадигме запрос-ответ, причем запросы не являются регулярными или имеют слабую плотность.

Примером такого сервиса может служить удаленный вызов функции решения некоторого уравнения. Программа высылает запрос и ждет ответа причем события не сильно зависят от времени.

В противоположность первой архитектуре, встраивание Python позволяет использовать функционал Python более гибко, можно вызывать функции Python прямо

из кода C/C++, который может быть использован другими языками в качестве библиотеки. Однако при использовании данного подхода, следует помнить про наличие GIL. В связи с наличием GIL в рамках одного процесса может быть запущен только один интерпретатор, даже если он встроенный.

5. ОСОБЕННОСТИ СЕРВИСА РАСПОЗНАВАНИЯ РЕЧИ

Сервис распознавания речи должен получать данные с микрофона, нарезать их на порции, делать над ними вычисления (подавление шума, вычисление спектрограммы, сравнение спектрограммы с образцами с помощью вероятностной сети) и отправлять далее в приложение на C#. Управление (запуск, остановка) должны реализовываться средствами C#.

Реализовать данный процесс с помощью сервиса с сетевым доступом, затруднительно, так как данные, результат обработки которых должно получить C# приложение, изначально порождаются в Python приложении. Так инициализировать взаимодействие, могут обе стороны C# (при отправке управляющих команд) и Python (при отправке распознанных данных с микрофона) обе стороны должны постоянно опрашивать друг друга, что является ресурсозатратным процессом. Дополнительным ограничением является частота обновления данных, и соответственно обращений к сервису, в рабочей системе обновление результатов достигает 50мс.

В начале процесса внедрения, была предпринята попытка реализовать коммуникацию C# и Python приложения через архитектуру сервиса с сетевым доступом. В процессе опытной эксплуатации данной архитектуры, поступили жалобы на излишнее потребление ресурсов процессора приложениями. В процессе профилирования было установлено, что 85 % процентов процессорного времени, используемого C# приложением, уходит на сетевое взаимодействие, в связи с чем был совершен переход на использование встраивания Python.

В связи с необходимостью одновременной работы нескольких потоков распознавания речи, встраивание должно производится в отдельные приложения, которые могут быть запущены C# приложением и обмениваться с ним данными.

Для взаимодействия.NET приложений и C++ приложений, через разделяемую память корпорацией Microsoft разработана библиотека с открытым исходным кодом IPC [8], которая внутри себя использует boost interprocess [9].

6. РАЗРАБОТКА C++ ПРИЛОЖЕНИЙ

Для разработки C++ приложений, которые выполняют Python код, необходимо максимально инкапсулировать функциональность приложения в классы, примером такой использования инкапсуляции может служить листинг 1.

Листинг 1

Процесс записи с микрофона

```
def sender(wav_queue, settings, lock, stop_event):
microphone_controller = MicrophoneController(settings)
while not stop_event.is_set():
if lock.acquire(False):
microphone_controller.init_stream()
lock.release()
while lock.acquire(False) and not stop_event.is_set():
start_time = time.time()
microphone_controller.process_stream(start_time)
chunks = microphone_controller.get_chunks()
for chunk in chunks:
wav_queue.put(chunk, block=False)
lock.release()
microphone_controller.close_stream()
```

После создания обертки можно использовать класс в приложении C++, в той области видимости, в которой создан интерпретатор. В описываемых приложениях для работы с интерпретатором создается отдельный поток. Функция которую выполняет данный поток (листинг 3), запускает интерпретатор, загружает модули python, и реализует логику, аналогичную логике работы кода представленного в листинге 1.

Листинг 2

Обертка класса MicrophoneController

```
#include “MicrophoneController.h”

namespace PythonAudioController {
MicrophoneController:: MicrophoneController(
py:: module_ mainModule,
std:: shared_ptr<Config> config,
) {
_module = mainModule;
_class = mainModule.attr(“MicrophoneController”);
_config = config;
_object = _class(“settings”_a = _config->getRawObject());
}
py:: object MicrophoneController:: getRawObject() {
return _object;
}

void MicrophoneController:: initStream() {
_object.attr(“init_stream”)();
}

void MicrophoneController:: processStream(time_t startTime) {
_object.attr(“process_stream”)(startTime);
}
```

```
24
25  void MicrophoneController:: getChunks(
26  std:: vector<PythonCPPCommon:: WaveChunkRaw>& waveChunks
27  ) {
28  py:: object chunksObject = _object.attr("get_chunks")();
29  py:: list chunksList = chunksObject.cast<py:: list>();
30  for (auto chunk: chunksList) {
31  py:: tuple chunkTuple = chunk.cast<py:: tuple>();
32  PythonCPPCommon:: WaveChunkRaw waveChunk;
33  waveChunk.number = chunkTuple[0].cast<long>();
34  waveChunk.buffer = chunkTuple[1].cast<std:: string>();
35  waveChunk.time = chunkTuple[2].cast<float>();
36  waveChunks.push_back(waveChunk);
37  }
38  }
39  void MicrophoneController:: getMicrophones(std:: string& microphones){
40  py:: object microphonePy = _object.attr("get_microphones")();
41  microphones = microphonePy.cast<std:: string>();
42  }
43  }
```

В случае, если бы в программе было бы необходимо запустить один поток, взаимодействующий с Python, он мог бы быть упакован в динамически загружаемую библиотеку, однако в случае стоящей задачи, необходимо запустить несколько потоков, следовательно каждый поток должен быть упакован в свой процесс.

Листинг 3

Поток для работы с Python

```
1   void pythonThread(
2   std:: condition_variable& pythonCond,
3   std:: mutex& recordingMutex,
4   std:: reference_wrapper<volatile bool> recording,
5   ThreadsafeQueue<PythonCPPCommon:: WaveChunkRaw>& tsWavechunkQueue
6   ) {
7   py:: scoped_interpreter guard{}; // start the interpreter
8   py:: object python_audio_controller = py:: module_:: import(
9   "python_audio_controller"
10  );
11  py:: object python_config = py:: module_:: import("python_config");
12  std:: vector<PythonCPPCommon:: WaveChunkRaw> waveChunks;
13  std:: shared_ptr<Config> config(new Config(python_config, "setting.cfg"));
14  std:: shared_ptr<MicrophoneController> microphoneController(
15  new MicrophoneController(python_audio_controller, config)
16  );
17  bool isStreamInit = false;
18  config->readConfig();
19  while (true)
20  {
21  if (recording) {
```

```
22 if (isStreamInit == false) {
23 try {
24 microphoneController->initStream();
25 isStreamInit = true;
26 } catch (py:: error_already_set& e) {
27 recording.get() = false;
28 continue;
29 }
30 }
31 std:: time_t time = std:: time(nullptr);
32 microphoneController->processStream(time);
33 microphoneController->getChunks(waveChunks);
34 for (PythonCPPCommon:: WaveChunkRaw waveChunk: waveChunks) {
35 tsWavechunkQueue.push(waveChunk);
36 }
37 waveChunks.clear();
38 } else {
39 std:: unique_lock<std:: mutex> lck{recordingMutex};
40 pythonCond.wait(lck, [recording] {return recording;});
41 }
42 }
43 }
```

7. КОММУНИКАЦИЯ МЕЖДУ ПРОЦЕСАМИ

Для коммуникации между C# и C++ процессами один из процессов (C++) создает сервер удаленных вызовов, который реализует сервис, к которому C# имеет асинхронный доступ.

Функция обработки асинхронного запроса приведена в листинге 4. Пример выполнения асинхронного запроса приведен в листинге 5.

Листинг 4

Сервис сервера удаленных вызовов

```
1  template <typename Callback>
2  void Service:: operator()(
3  const RequestAudio& request,
4  Callback&& callback
5  ) {
6  ResponseAudio response;
7  std:: ostringstream text;
8  if (request.Op == Operation:: Start) {
9  recording.get() = true;
10 pythonCond.notify_one();
11 text << “OK”;
12 } else if (request.Op == Operation:: Get) {
13 response.WaveChunks = WaveChunkRawVector(
14 memory->GetAllocator<WaveChunkRawShared>()
15 );
16 if (tsWavechunkQueue.empty()) {
```

```
17  WaveChunkRaw wavechunk;
18  WaveChunkRawShared wavechunkShared;
19  tsWavechunkQueue.wait_and_pop(wavechunk);
20  wavechunkShared.number = wavechunk.number;
21  wavechunkShared.time = wavechunk.time;
22  wavechunkShared.buffer = SharedUInt8Vector(
23  wavechunk.buffer.begin(),
24  wavechunk.buffer.end(),
25  memory->GetAllocator<uint8_t>()
26  );
27  response.WaveChunks->push_back(wavechunkShared);
28  }
29  while (!tsWavechunkQueue.empty()) {
30  WaveChunkRaw wavechunk;
31  WaveChunkRawShared wavechunkShared;
32  tsWavechunkQueue.try_pop(wavechunk);
33  wavechunkShared.number = wavechunk.number;
34  wavechunkShared.time = wavechunk.time;
35  wavechunkShared.buffer = SharedUInt8Vector(
36  wavechunk.buffer.begin(),
37  wavechunk.buffer.end(),
38  memory->GetAllocator<uint8_t>()
39  );
40  response.WaveChunks->push_back(wavechunkShared);
41  }
42  text << “OK”;
43  } else if (request.Op == Operation:: Pause) {
44  std:: lock_guard<std:: mutex> lock(recordingMutex);
45  recording.get() = false;
46  pythonCond.notify_one();
47  text << “OK”;
48  }
49  response.Text.emplace(
50  text.str().c_str(),
51  memory->GetAllocator<char>()
52  );
53  try {
54  callback(std:: move(response));
55  } catch (const std:: exception& e) {
56  std:: cout << “Failed to send response:” << e.what() << std:: endl;
57  }
58  }
```

Листинг 5

Обращение к сервису

```
1  if (command == IPCMicrophoneClientCommand.StartRecord) {
2  var request = new RequestAudio {Op = Operation.Start};
3  ResponseAudio response;
4  try {
5  response = client.InvokeAsync(request).Result;
6  } catch (System.Exception e) {
```

```
7   Console.WriteLine($”Failed to send request: {e.Message}”);
8   client = null;
9   continue;
10  }
11  }
```

Сервис удаленных вызовов позволяет управлять процессом записи сигнала с микрофона с помощью логической переменной и двух примитивов синхронизации, условной переменной и связанных с ней мьютексов.

При запросе запуска или остановки записи сервис изменяет логическую переменную и уведомляет поток Python с помощью условной переменной. При запросе на получение данных, сервис возвращает данные из очереди и опустошает ее. Аналогичный сервис используется для распознавания отрезков. C# реализует связь с данным сервисом через асинхронные вызовы.

Всего в приложении участвуют 10 типов потоков и 3 типа процесса, описание которых дано в таблице 1. Данное распределение деятельности позволяет эффективно по времени и потребляемым ресурсам осуществлять распознавание речи.

Таблица 1

Описание потоков

Процесс	Поток	Назначение и особенности работы
C# приложение	Основой	Управление настройками, управление процессом распознавания (запуск, остановка), отображение результатов распознавания.
C# приложение	Контроллер распознавания речи	Координирование работы, запрашивание аудио данных от контроллера процесса записи, передача записанных фрагментов для распознавания контроллеру процесса распознавания, передача распознанных данных основному потоку. Для синхронизации с контроллерами процесса записи, контроллером процесса распознавания используются события. Для приема данных от контроллера процесса записи используется неблокирующее чтение и событие. Для отправки данных контроллеру процесса распознавания используется неблокирующая запись. Для приема данных от контроллера процесса распознавания используется неблокирующее чтение и событие. Блокировка потока происходит в момент ожидания событий.
	Контроллер процесса записи	Управление коммуникацией с процессом записи. Прием команд о начале записи, паузе в записи, получении записанного и их выполнение. После выполнения команды отправляет событие и блокируется до следующей команды. В процессе выполнения команд поток блокируется в ожидании результатов.
	Контроллер процесса распознавания	Управление коммуникацией с процессом записи. Выполняет блокирующее чтение очереди фрагментов, направленных на распознавание. Отправка прочитанных фрагментов на распознавание Python приложению. При ожидании результатов распознавания поток блокируется.

Процесс	Поток	Назначение и особенности работы
	Основной	Запуск потока Python интерпретатора, создание сервера, который при подключении к нему создает потоки сервиса, которые обрабатывают запросы от C# приложения.
Python приложение записи	Поток Python интерпретатора	При начале своей работы инициализирует микрофон для записи. В случае активной записи читает новые фрагменты с микрофона и с помощью неблокирующей записи помещает в очередь. В случае неактивной записи блокируется по условной переменной, которая отслеживает условие изменения активности записи.
Python приложение записи	Поток сервиса	Запускается при приеме нового запроса от C# приложения. Осуществляет активацию и деактивацию записи, путем изменения значения логической переменной и уведомления условной переменной. Так же позволяет считать записанные фрагменты из очереди.
	Основной	Запуск потока Python интерпретатора, создание сервера, который при подключении к нему создает потоки сервиса, которые обрабатывают запросы от C# приложения.
Python приложение распознавания	Поток Python интерпретатора	Инициализация распознавателя речи. Блокирующее чтение отрезков аудио из очереди входящих данных, запись распознанных отрезков в очередь исходящих данных.
	Поток сервиса	Прием запроса и данных для распознавания, размещение данных в очереди входящих данных, блокирующее чтение из очереди исходящих данных, отправка распознанных данных в ответ.
	Основной	Запуск потока Python интерпретатора, создание сервера, который при подключении к нему создает потоки сервиса, которые обрабатывают запросы от C# приложения.

8. ЗАКЛЮЧЕНИЕ

В данной работе изложен один из способов организации сервиса, использующего возможности языка Python для осуществления научных вычислений, позволяющий обойти особенности многопоточной работы интерпретатора Python. Данный способ позволяет связать с Python любой язык, который поддерживает C++ библиотеки. В рамках статьи опущены темы упаковки приложений и тонкости транслирования некоторых типов, однако приведенная в списке литературы документация покрывает данные, в большей мере технические подробности.

В результате внедрения описанной в статье архитектуры в конкретном случае удалось существенно снизить нагрузку на процессор (на 75 % по сравнению с использованием очереди сообщений ZeroMQ и multiprocessing).

В начале 2023 сообществом разработчиков Python был предложен PEP (Python Enhancement Proposal) 703, который дает возможность отключить GIL. В Python 3.13, выпуск которого назначен на 1 октября 2024 года, будет такая возможность. Поэтому в течении двух-трех лет сообщество может увидеть новые подходы в реализации многопоточных систем на Python.

Литература

1. Tool recommendations [Электронный ресурс] // Python Packaging User Guide URL: https://packaging.python.org/en/latest/guides/tool-recommendations/ (дата обращения: 05.06.2024)
2. virtualenv [Электронный ресурс] // virtualenv URL: https://virtualenv.pypa.io/en/latest/index.html (дата обращения: 05.06.2024)
3. PyInstaller Manual [Электронный ресурс] // PyInstaller 6.8.0 documentation URL: https://pyinstaller.org/en/stable/ (дата обращения: 25.06.2024)
4. Глобальная блокировка интерпретатора (GIL) и её воздействие на многопоточность в Python [Электронный ресурс] // Хабр URL: https://habr.com/ru/companies/wunderfund/articles/586360/ (дата обращения: 07.06.2024)
5. multiprocessing – Process-based parallelism [Электронный ресурс] // Python 3.12.4 documentation URL: https://docs.python.org/3/library/multiprocessing.html (дата обращения: 07.06.2024)
6. pickle – Python object serialization [Электронный ресурс] // Python 3.12.4 documentation URL: https://docs.python.org/3/library/pickle.html (дата обращения: 08.06.2024)
7. *Gorelick, Micha, and Ian Ozsvald.* High Performance Python: Practical Performant Programming for Humans. O'Reilly Media, 2020.
8. microsoft/IPC: IPC is a C++ library that provides inter-process communication using shared memory on Windows. A.NET wrapper is available which allows interaction with C++ as well. [Электронный ресурс] // GitHub URL: https://github.com/microsoft/IPC (дата обращения: 09.06.2024)
9. Chapter 16. Boost.Interprocess – 1.85.0 [Электронный ресурс] // The Boost C++ Libraries URL: https://www.boost.org/doc/libs/1_85_0/doc/html/interprocess.html (дата обращения: 09.06.2024)
10. pybind11 – Seamless operability between C++11 and Python [Электронный ресурс] // pybind11 documentation URL: https://pybind11.readthedocs.io/en/stable/ (дата обращения: 09.06.2024)

Multithread Speech Recognition Service Development and Integration Using C#, C++ and Python

Nikita I. Levonovich*
Moscow State University of Psychology and Education (MSUPE), Moscow, Russia
ORCID: https://orcid.org/0000-0002-8580-0490
e-mail: levonikitatech@yandex.ru

This article discusses specificity of multithread industrial systems implementation using Python and its computation libraries. Article contains description of theory, including working principle of the Python's Global Interpreter Locker (GIL), dependency management, multiprocessing library. The practical part of the article devoted to speech recognition service implementation. The implementation uses interprocess communication through shared memory based on "boost.interprocess" library. The result of integration architecture described by the article is reducing of CPU loading, in that case.

Keywords: software engineering, Python, C++, C#, boost, multithread applications, speech recognition.

Acknowledgements. The author is grateful for assistance in data about Python interpreter internals collection DevRel Evrone Grigory Petrov.

For citation:
Levonovich N.I. Multithread Speech Recognition Service Development and Integration Using C#, C++ and Python. *Modelirovanie i analiz dannykh = Modelling and Data Analysis,* 2024. Vol. 14, no. 3, pp. 135–148. DOI: https://doi.org/10.17759/mda.2024140308 (In Russ., abstr. in Engl.).

References

1. Tool recommendations [Elektronnyi resurs] // Python Packaging User Guide URL: https://packaging.python.org/en/latest/guides/tool-recommendations/ (Accessed 05.06.2024)
2. virtualenv [Elektronnyi resurs] // virtualenv URL: https://virtualenv.pypa.io/en/latest/index.html (Accessed 05.06.2024)
3. PyInstaller Manual [Elektronnyi resurs] // PyInstaller 6.8.0 documentation URL: https://pyinstaller.org/en/stable/ (Accessed 25.06.2024)
4. Global'naja blokirovka interpretatora (GIL) i ejo vozdejstvie na mnogopotochnost' v Python [Elektronnyi resurs] // Хабр URL: https://habr.com/ru/companies/wunderfund/articles/586360/ (Accessed 07.06.2024) (In Russ.).

***Nikita I. Levonovich,** Master's Degree Student, Junior Research Associate, Youth Laboratory Information Technologies for Psychological Diagnostics, Moscow State University of Psychology & Education, Moscow, Russia, ORCID: https://orcid.org/0000-0002-8580-0490, e-mail: levonikitatech@yandex.ru

5. multiprocessing – Process-based parallelism [Elektronnyi resurs] // Python 3.12.4 documentation URL: https://docs.python.org/3/library/multiprocessing.html (Accessed 07.06.2024)
6. pickle – Python object serialization [Elektronnyi resurs] // Python 3.12.4 documentation URL: https://docs.python.org/3/library/pickle.html (Accessed 08.06.2024)
7. *Gorelick, Micha, and Ian Ozsvald.* High Performance Python: Practical Performant Programming for Humans. O'Reilly Media, 2020.
8. microsoft/IPC: IPC is a C++ library that provides inter-process communication using shared memory on Windows. A.NET wrapper is available which allows interaction with C++ as well. [Elektronnyi resurs] // GitHub URL: https://github.com/microsoft/IPC (Acessed 09.06.2024)
9. Chapter 16. Boost.Interprocess – 1.85.0 [Elektronnyi resurs] // The Boost C++ Libraries URL: https://www.boost.org/doc/libs/1_85_0/doc/html/interprocess.html (Acessed 09.06.2024)
10. pybind11 – Seamless operability between C++11 and Python [Elektronnyi resurs] // pybind11 documentation URL: https://pybind11.readthedocs.io/en/stable/ (Acessed 09.06.2024)

Получена 05.08.2024 Received 05.08.2024
Принята в печать 29.08.2024 Accepted 29.08.2024