Практическая работа №4

Задание

Разработайте программное обеспечение проверки криптостойкости паролей с использованием атаки методом грубой силы (брутфорс).

Найдите с помощью алгоритма полного перебора пятибуквенные пароли, соответствующие следующим хэш-значениям SHA-256 и выведите их на экран:

Первый хэш:

```
1115dd800feaacefdf481f1f9070374a2a81e27880f187396db67958b207cbad
```

Второй хэш:

```
3a7bd3e2360a3d29eea436fcfb7e44c735d117c42d1c1835420b6b9942dd4f1b
```

Третий хэш:

```
74e1bb62f8dabb8125a58852b63bdf6eaef667cb56ac7f7cdba6d7305c50a22f
```

Хэш значения могут считываться из файла или непосредственно с консоли (формы для ввода хэш-значения).

Ваша программа должна перебрать все возможные пароли, состоящие только из пяти строчных букв английского алфавита ASCII.

Программа должна иметь возможность запуска перебора в однопоточном режиме или в многопоточном режиме (количество потоков может задаваться пользователем). Для каждого режима необходимо выводить затраченное время на подбор.

Код должен быть стойким к ошибкам ввода (предполагается использование конструкций обработки ошибок `try:... except:...` и дополнительных проверок входных данных). Программа должна содержать интерфейс для ввода данных и выбора действий, тип интерфейса: графический или консольный.

Рекомендуемый язык программирования: Python

Форма отчёта: файл с титульной страницей в формате PDF или DOCX (с ФИО, группой), копируемый исходный код. Прикреплённый отчёт в СДО.

Формат защиты: после прикрепления отчёта в СДО, подходить к преподавателю лично, для демонстрации полученных результатов. Для наглядности можно показать работоспособность кода в среде разработки (с компьютера, ноутбука, телефона). Суметь ответить на поясняющие вопросы.

Срок сдачи работы: до 6-го практического занятия

Теоретическая часть

Среды разработки Python:

* PyCharm Community
* VScode
* Онлайн-компилятор CodeChef
* Компилятор Online Python
* PyDroid (на Android, доступен в Play Market)

Установка дополнительных пакетов: `pip install название_пакета`

Вы можете вычислять хэши SHA-256, инициализировав функцию `sha256` из модуля `hashlib`. Пример вычисления из строки:

```
# Импорт библиотеки для вычисления хэша
from hashlib import sha256

data = "Произвольная строка"
hash_sha256 = sha256(data.encode('utf-8')) # Кодирование и вычисление хэша
hash_sha256=hash_sha256.hexdigest()         # Преобразование хэша в строку
print(hash_sha256) #
4a02e7ba229337852aa663ecd8f95a5ac26d2d8e0c3475c4c97c5fd3b003d817
```

Пример вычисления sha256 файла:

```
# Импорт библиотеки для вычисления хэша
from hashlib import sha256
```

```
# Открытие файла на чтение по байтам
file=open('Файл с произвольной строкой.txt', 'rb')
hash_sha256 = sha256(file.read())     # Вычисление хэша байт-
содержимого файла
hash_sha256 = hash_sha256.hexdigest() # Преобразование хэша в строку
print(hash_sha256) #
4a02e7ba229337852aa663ecd8f95a5ac26d2d8e0c3475c4c97c5fd3b003d817
```

Пример реализации кода, функция `my_function` которого будет выполняться многопоточно, с объединением результатов:

```
# Импорт инструмента для создания пулов потоков
from concurrent.futures import ThreadPoolExecutor, as_completed

# Функция, принимающая на вход исходные данныа, производящая
вычисление и возвращающая результат
def my_function(x):
    return x * x

# Список входных данных
inputs = [1, 2, 3, 4, 5]

# Количество одновременно запущенных поотоков (рекомендуется указывать
число ядер процессора)
max_workers=8

# Создание пула потоков
with ThreadPoolExecutor(max_workers=max_workers) as executor:

    # Запуск функции в нескольких потоках через executor.submit
    futures = []
    for i in inputs:
        future=executor.submit(my_function, i)
        futures.append(future)

    # Объединение результатов по готовности
    results = []
    for future in as_completed(futures):
        rezult=future.result()
        results.append(rezult)

# Вывод результата
print(results)
```

Наиболее очевидная область применения многопоточности – это программирование интерфейсов. Многопоточность незаменима тогда, когда

необходимо, чтобы графический интерфейс продолжал отзываться на действия пользователя во время выполнения некоторой обработки информации. Например, поток, отвечающий за интерфейс, может ждать завершения другого потока, загружающего файл из интернета, и в это время выводить некоторую анимацию или обновлять прогресс-бар. Кроме того, он может остановить поток загружающий файл, если была нажата кнопка «отмена».

Еще одна популярная и, пожалуй, одна из самых хардкорных областей применения многопоточности – игры. В играх различные потоки могут отвечать за работу с сетью, анимацию, расчет физики и т.п.

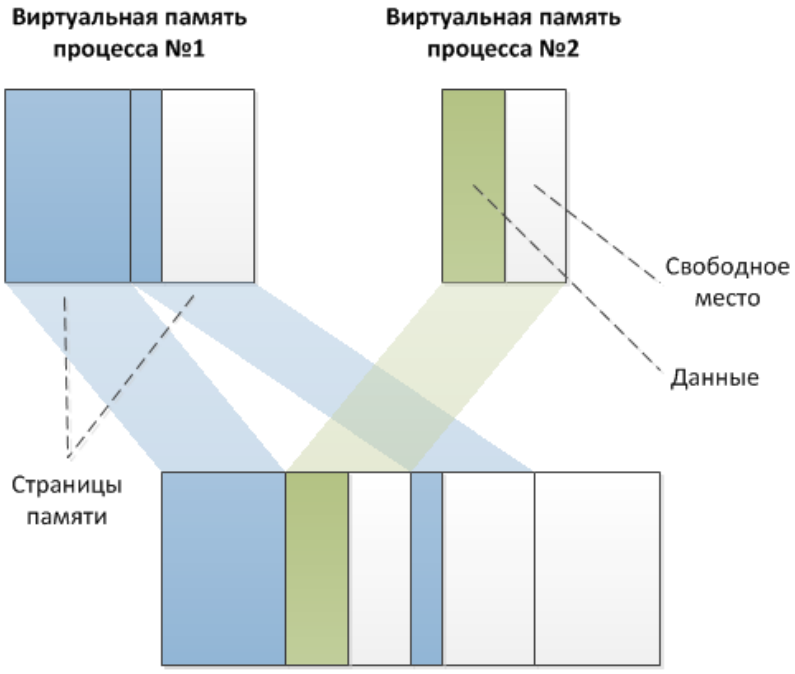
Процессы

Процесс — это совокупность кода и данных, разделяющих общее виртуальное адресное пространство. Чаще всего одна программа состоит из одного процесса, но бывают и исключения (например, браузер Chrome создает отдельный процесс для каждой вкладки, что дает ему некоторые преимущества, вроде независимости вкладок друг от друга). Процессы изолированы друг от друга, поэтому прямой доступ к памяти чужого процесса невозможен (взаимодействие между процессами осуществляется с помощью специальных средств).

Для каждого процесса ОС создает так называемое «виртуальное адресное пространство», к которому процесс имеет прямой доступ. Это пространство принадлежит процессу, содержит только его данные и находится в полном его распоряжении. Операционная система же отвечает за то, как виртуальное пространство процесса проецируется на физическую память.

Схема этого взаимодействия представлена на картинке. Операционная система оперирует так называемыми страницами памяти, которые представляют собой просто область определенного фиксированного размера. Если процессу становится недостаточно памяти, система выделяет ему

дополнительные страницы из физической памяти. Страницы виртуальной памяти могут проецироваться на физическую память в произвольном порядке.



При запуске программы операционная система создает процесс, загружая в его адресное пространство код и данные программы, а затем запускает главный поток созданного процесса.

Потоки

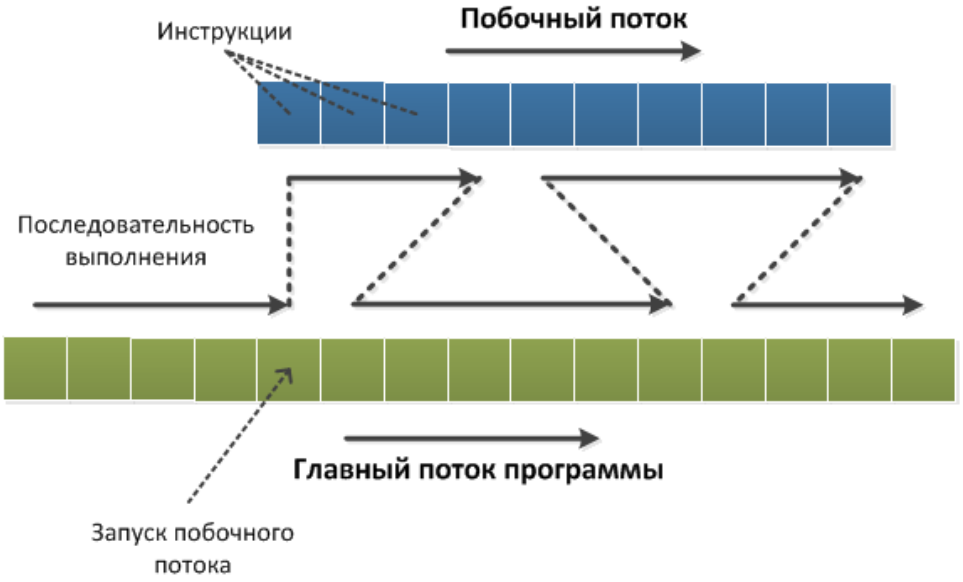
Один поток – это одна единица исполнения кода. Каждый поток последовательно выполняет инструкции процесса, которому он принадлежит, параллельно с другими потоками этого процесса.

Следует отдельно обговорить фразу «параллельно с другими потоками». Известно, что на одно ядро процессора, в каждый момент времени, приходится одна единица исполнения. То есть одноядерный процессор может обрабатывать команды только последовательно, по одной за раз (в упрощенном случае). Однако запуск нескольких параллельных потоков возможен и в системах с одноядерными процессорами. В этом случае система будет периодически переключаться между потоками, поочередно давая

выполняться то одному, то другому потоку. Такая схема называется псевдопараллелизмом. Система запоминает состояние (контекст) каждого потока, перед тем как переключиться на другой поток, и восстанавливает его по возвращению к выполнению потока. В контекст потока входят такие параметры, как стек, набор значений регистров процессора, адрес исполняемой команды и прочее…

Проще говоря, при псевдопараллельном выполнении потоков процессор мечется между выполнением нескольких потоков, выполняя по очереди часть каждого из них.

Вот как это выглядит:



Цветные квадраты на рисунке – это инструкции процессора (зеленые – инструкции главного потока, синие – побочного). Выполнение идет слева направо. После запуска побочного потока его инструкции начинают выполняться вперемешку с инструкциями главного потока. Кол-во выполняемых инструкций за каждый подход не определено.

Каждый процесс имеет хотя бы один выполняющийся поток. Тот поток, с которого начинается выполнение программы, называется главным.

Потоки имею свойство запускаться и работать асинхронно. Асинхронность означает то, что нельзя утверждать, что какая-либо

инструкция одного потока, выполнится раньше или позже инструкции другого. Или, другими словами, параллельные потоки независимы друг от друга, за исключением тех случаев, когда программист сам описывает зависимости между потоками с помощью предусмотренных для этого средств языка.

Завершение процесса и демоны

В Python процесс завершается тогда, когда завершается последний его поток. Даже если главный поток уже завершился, но еще выполняются порожденные им потоки, система будет ждать их завершения.

Однако это правило не относится к особому виду потоков – демонам. Если завершился последний обычный поток процесса, и остались только потоки-демоны, то они будут принудительно завершены и выполнение процесса закончится. Чаще всего потоки-демоны используются для выполнения фоновых задач, обслуживающих процесс в течение его жизни.

Объявить поток демоном достаточно просто — нужно во время инициализации объекта потока явно указать, что он демон: `Thread(target=function, daemon=True)`;

Проверить, является ли поток демоном, можно вызвав его метод `isDaemon()`.