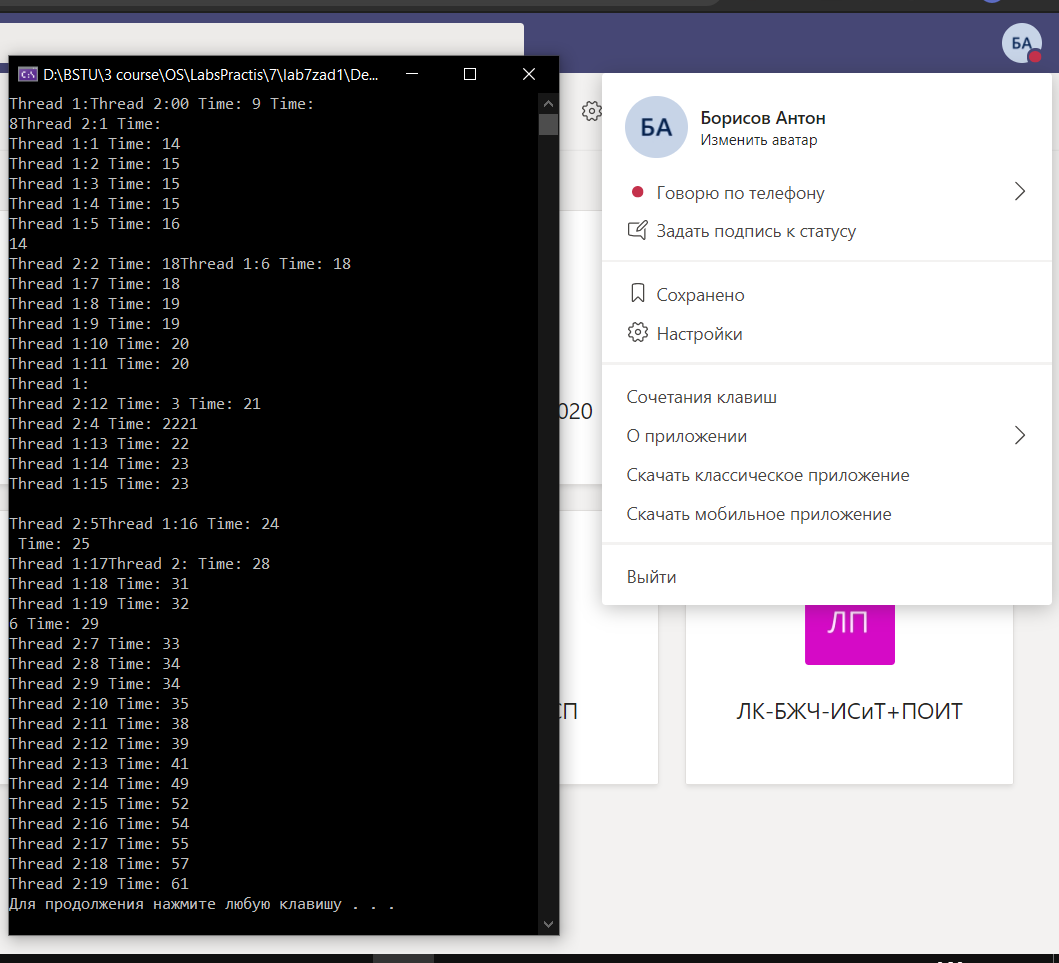
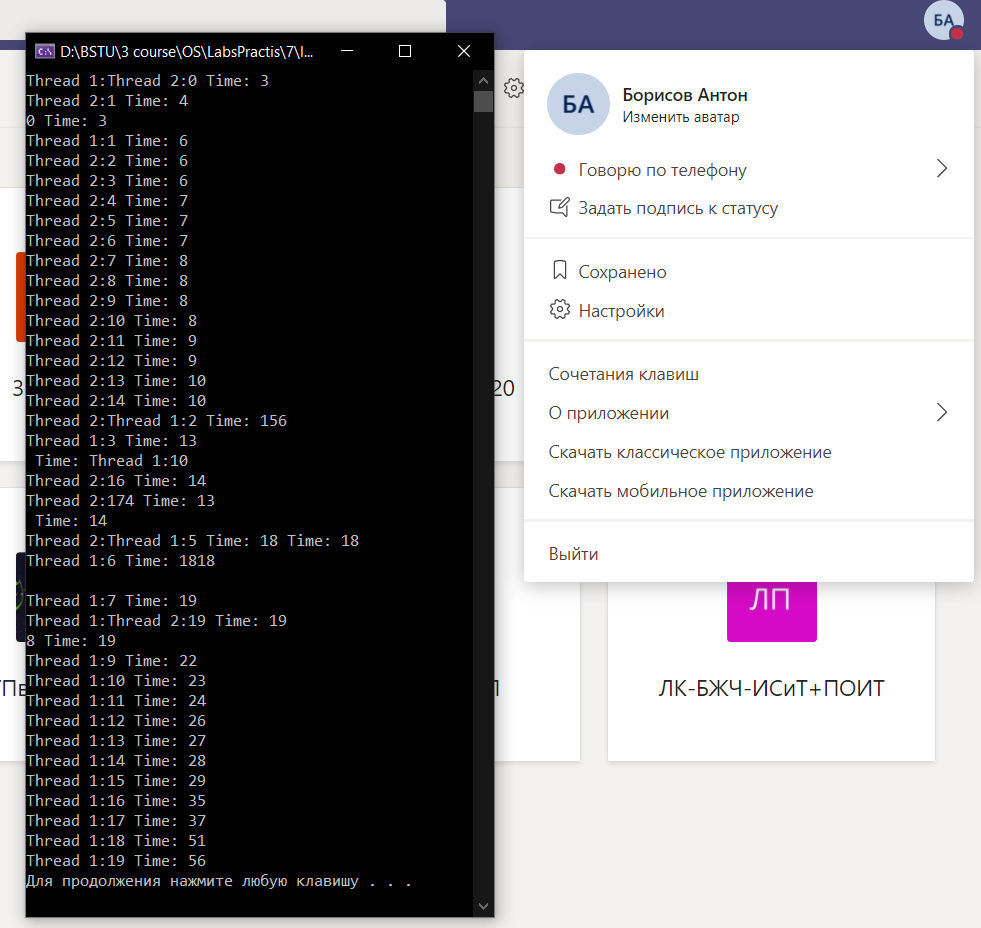
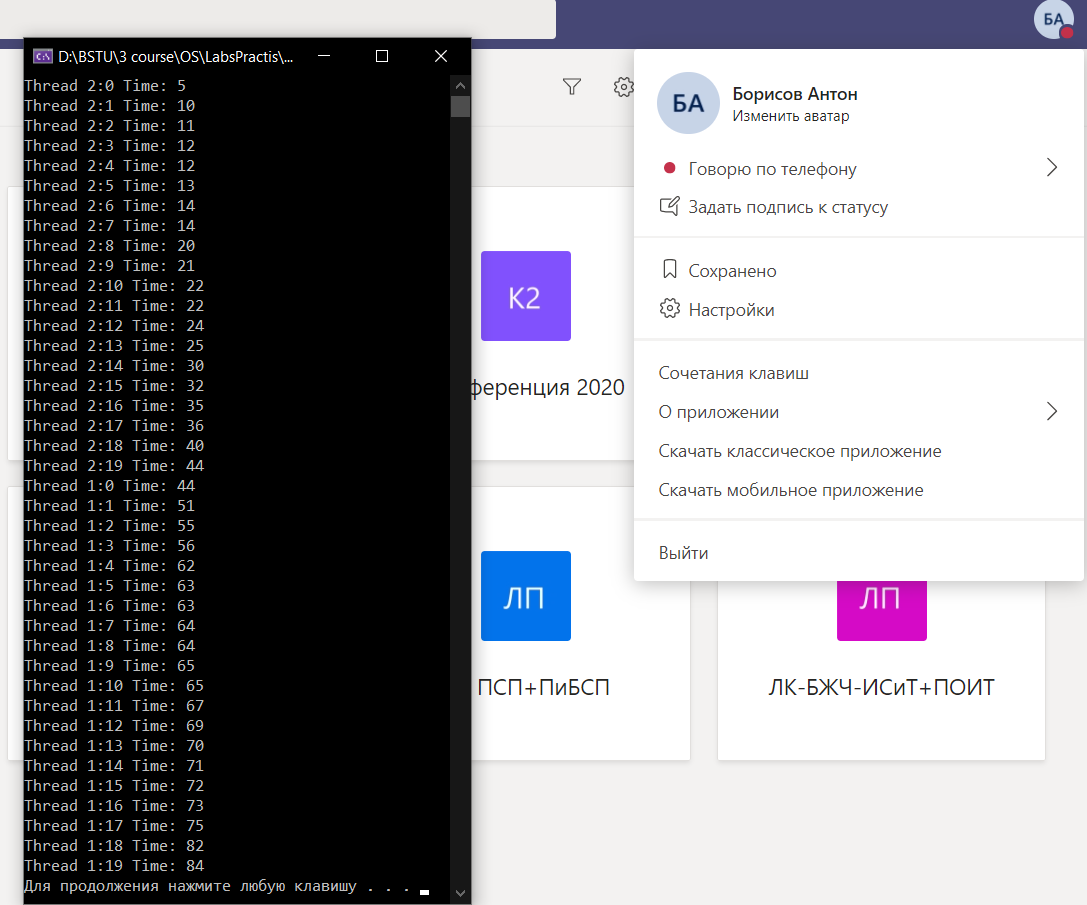
**Задание-1**

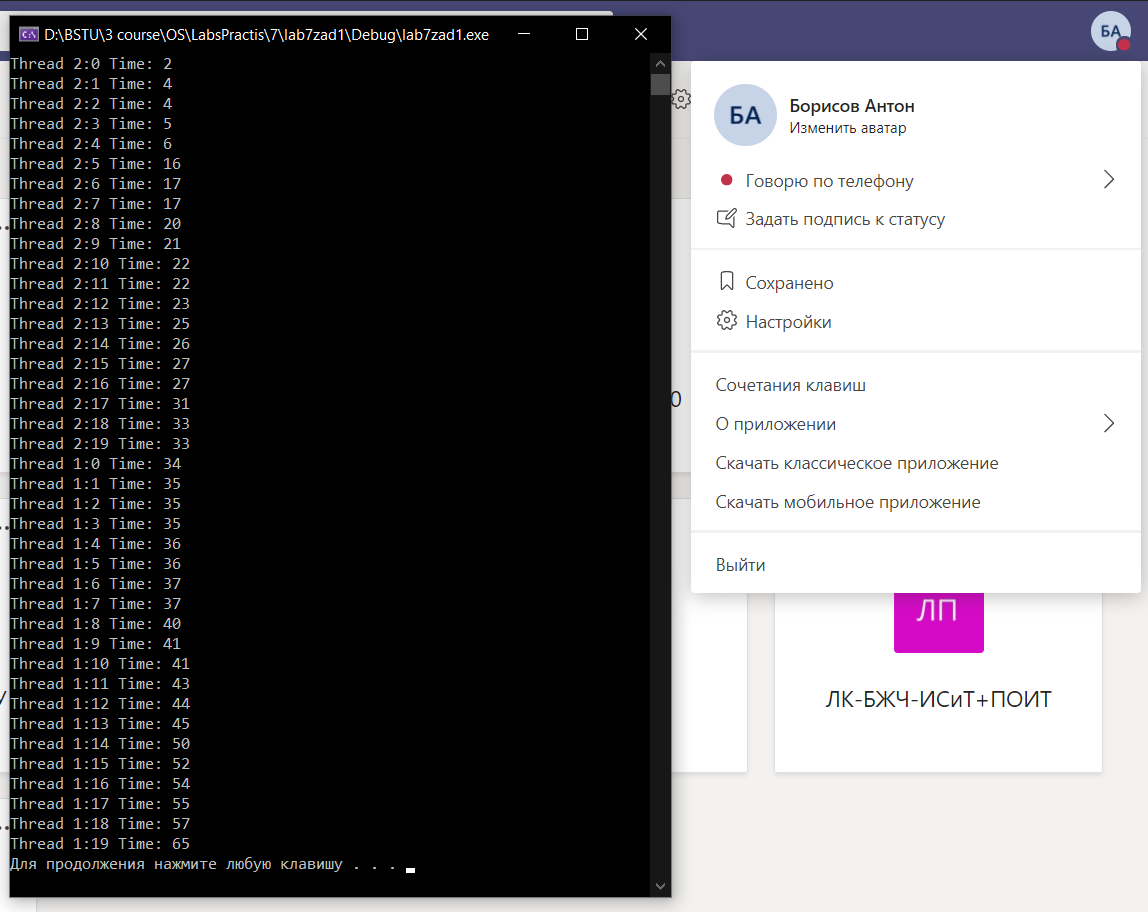
- Скриншот(ы) нескольких запусков приложения и результатов без синхронизации.





- Скриншот(ы) нескольких запусков приложения и результатов с синхронизацией.



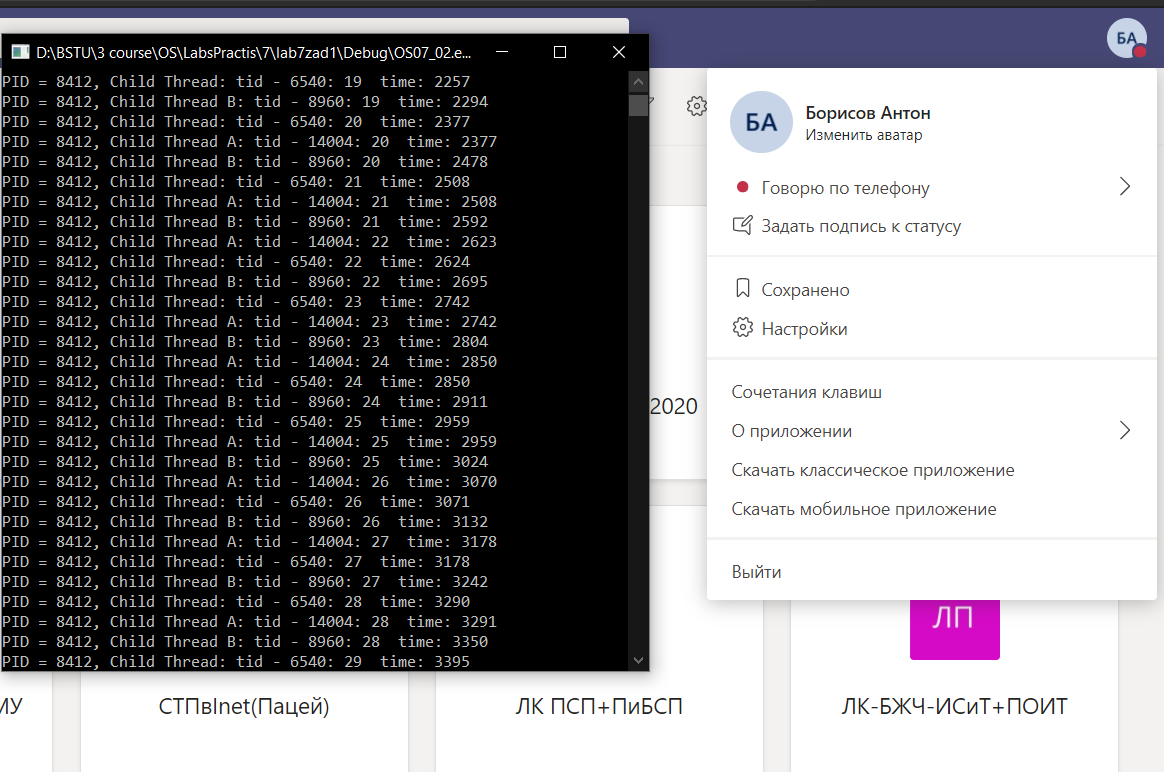


Описать алгоритм работы вашего механизма синхронизации:

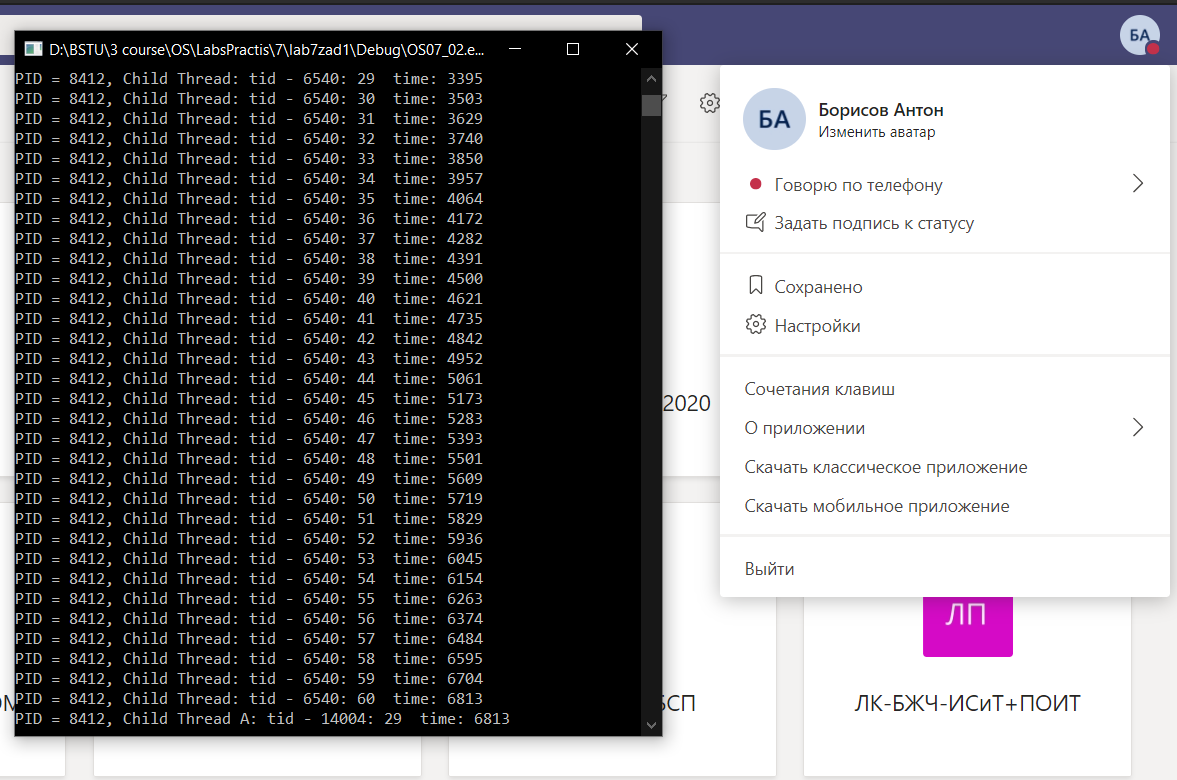
При старте работы отдельного потока с помощью команды ассемблера BTS сохраняем значение из первого операнда(в нашем случаи из переменной mutex) в флаг CF и устанавливаем значение первого операнда в 1. Дальше мы используем команду jc, которая выполняет переход по метке если в флаге CF содержится единица(то есть, если до этого какой то поток изменил значение переменной mutex, то этот поток будет находиться в цикле пока потом который установил 1 в этой переменной не установит ее в 0). По окончанию выполнения потока выполяется команда BTR которая обнуляет первый операнд(переменную mutex).

**Задание-2**

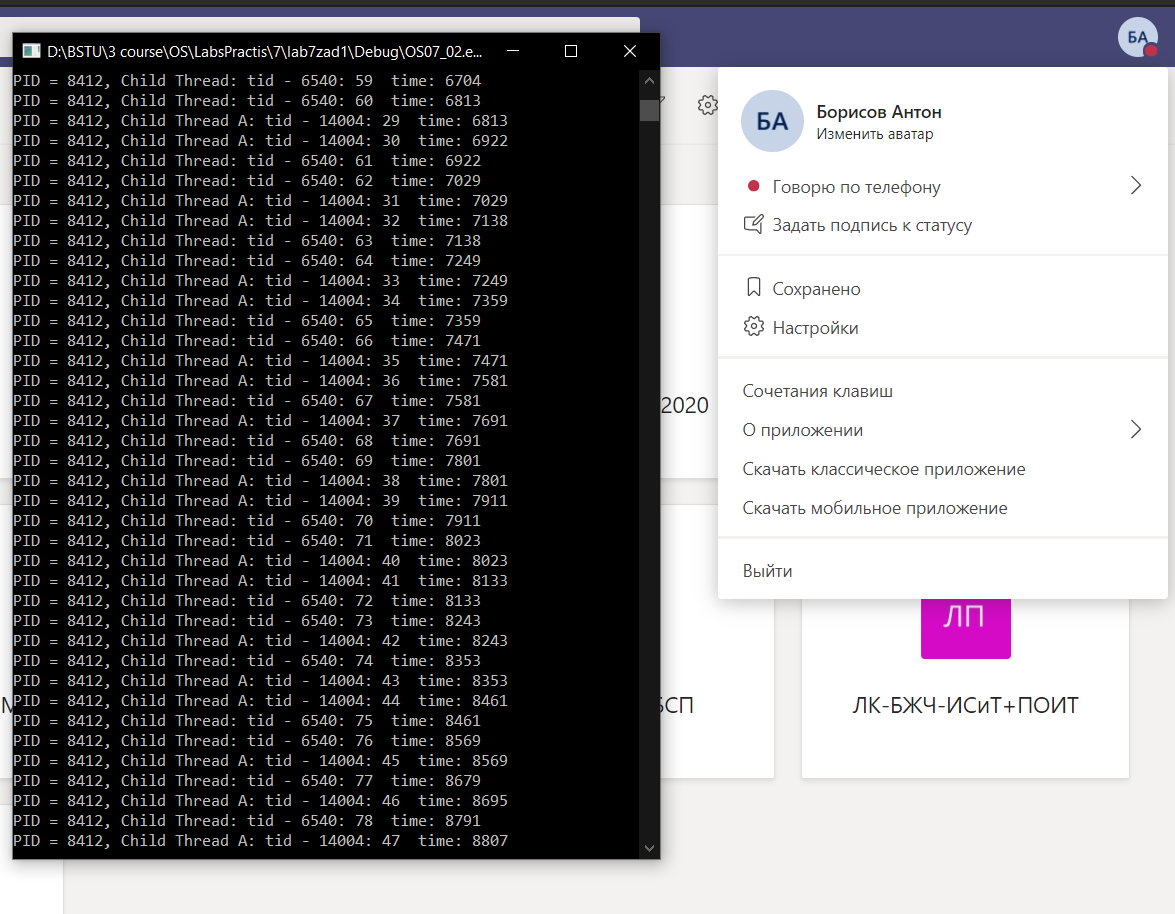
**Скриншот(ы) запуска приложения и результатов до 30 итерации.**



Скриншот(ы) работы приложения между 30 и 60 итерацией.



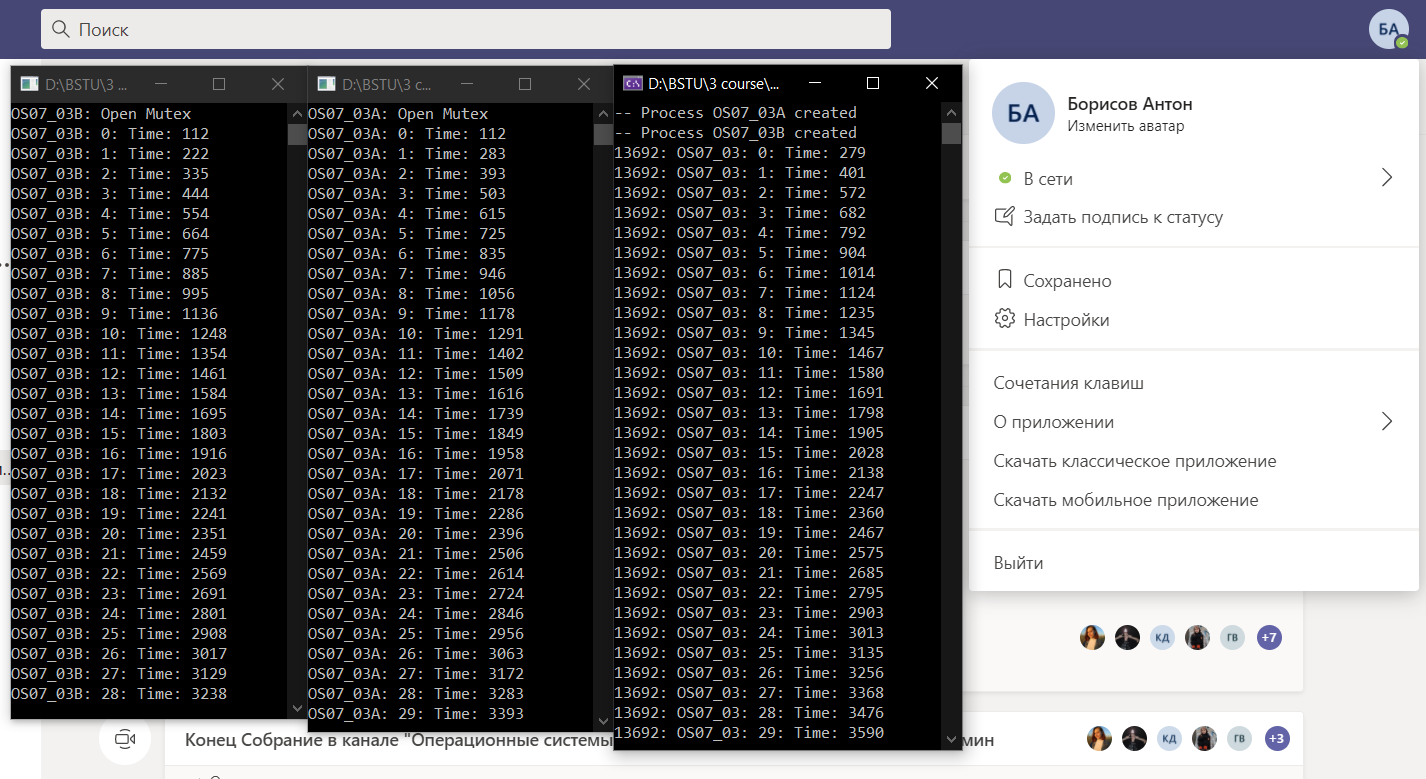
- Скриншот(ы) работы приложения между после 60 итерации.



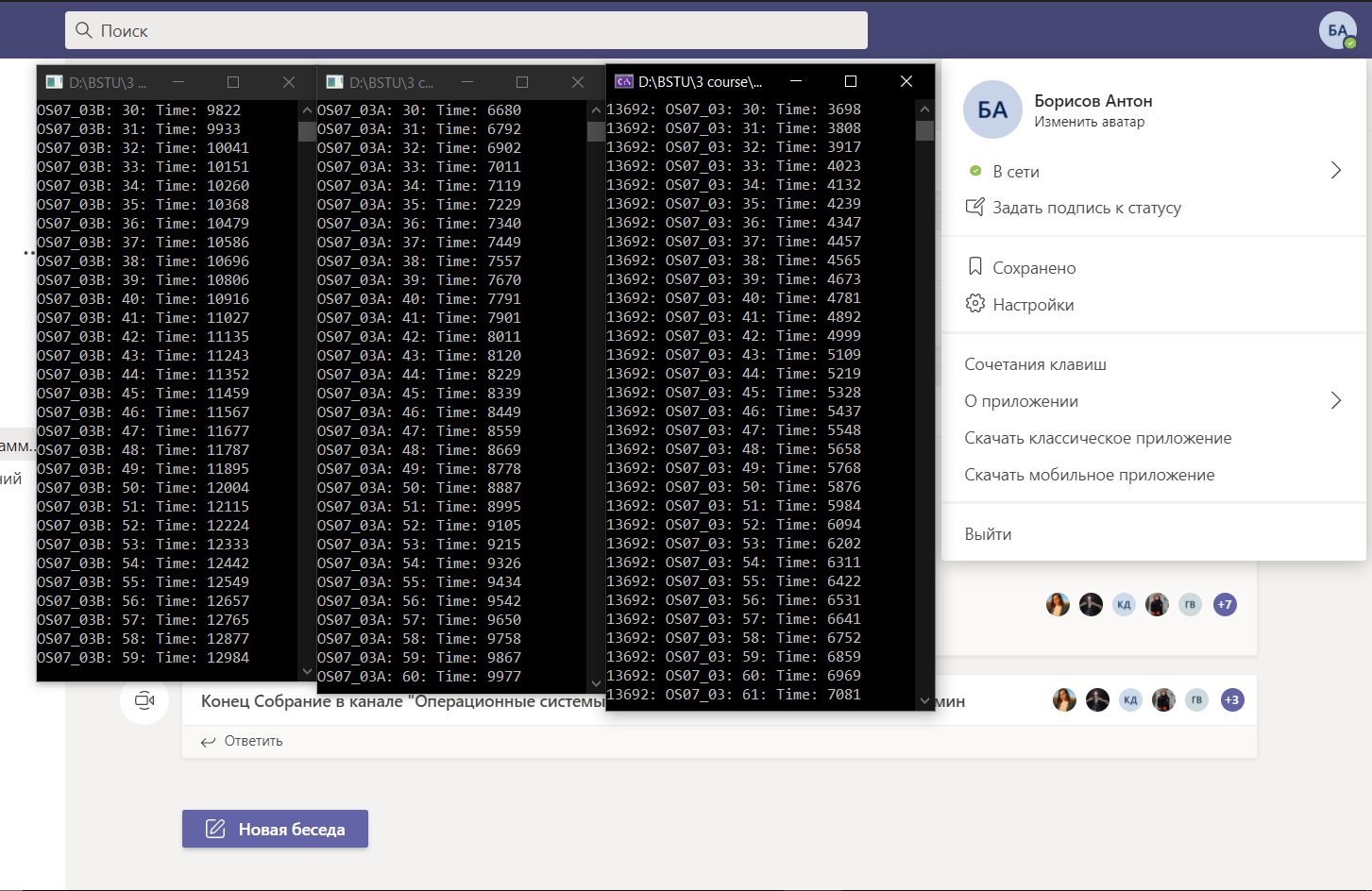
Объяснить принцип работы секций: Захват критической секции означает атомарное увеличение переменной на 1. Переход к ожиданию на событии ядра осуществляется только в случае, если значение переменной до захвата было уже больше 0, то есть происходит реальное «соревнование» двух или более потоков за ресурс. расписать функции для работы с ними в С++: CRTITICAL\_SECTION csFlag; //объявление критической секции InitializeCriticalSection(LPCRITICAL\_SECTION lpCriticalSection); //инициализациия критической секции EnterCriticalSection(LPCRITICAL\_SECTION lpCriticalSection); //функция входа в критическую секция LeaveCriticalSection(LPCRITICAL\_SECTION lpCriticalSection); //функция выхода из критической секции DeleteCriticalSection(LPCRITICAL\_SECTION lpCriticalSection); //Удаление(деинициализация) критической секции

**Задание-3**

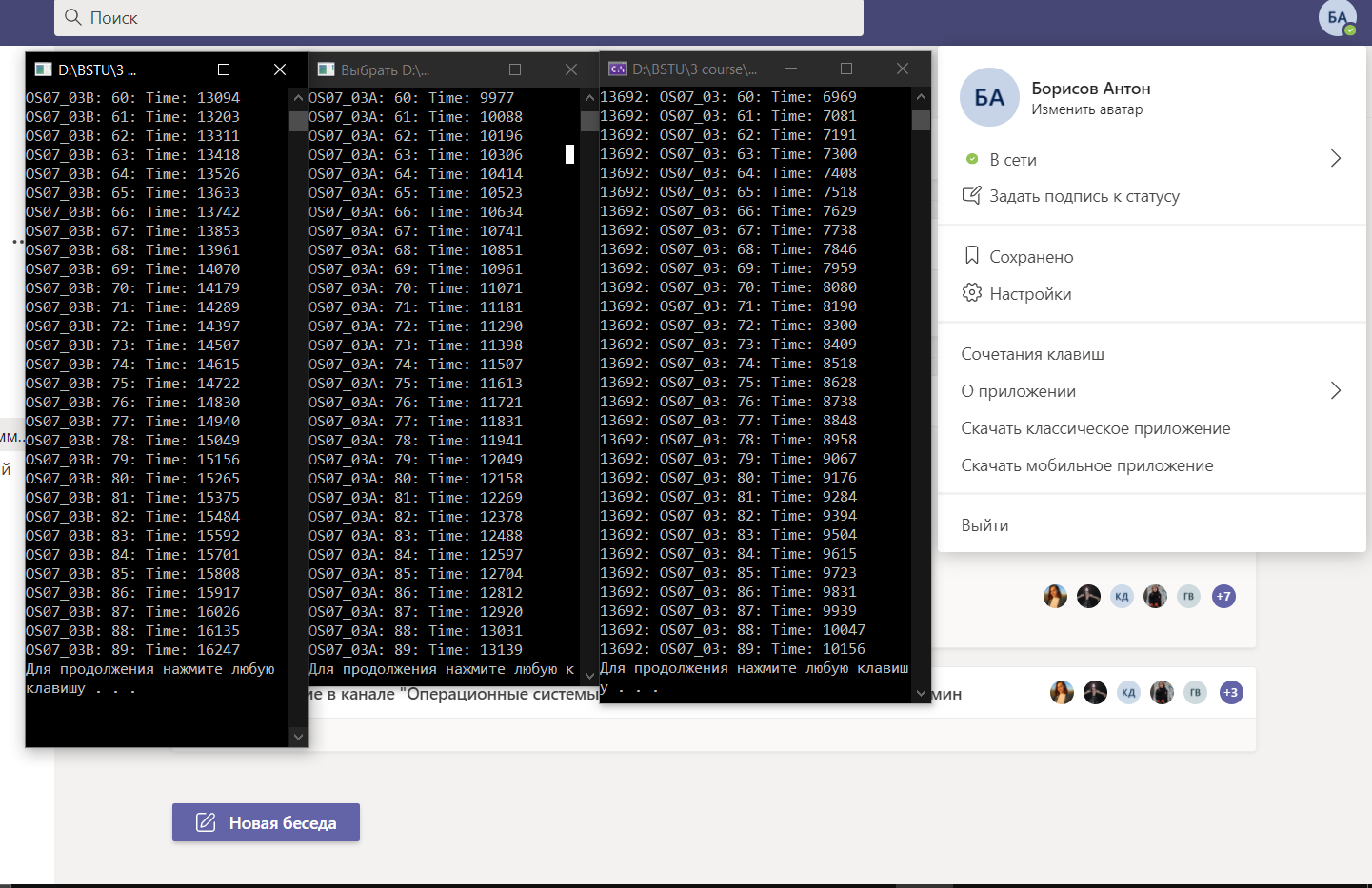
- Скриншот(ы) запуска приложения и результатов до 30 итерации.



- Скриншот(ы) работы приложения между 30 и 60 итерацией.



- Скриншот(ы) работы приложения между после 60 итерации.



Объяснить принцип работы мьютекса:

Условно классический мьютекс можно представить в виде переменной, которая может находиться в двух состояниях: в заблокированном и в незаблокированном. При входе в свою критическую секцию поток вызывает функцию перевода мьютекса в заблокированное состояние, при этом поток блокируется до освобождения мьютекса, если другой поток уже владеет им. При выходе из критической секции поток вызывает функцию перевода мьютекса в незаблокированное состояние. В случае наличия нескольких заблокированных по мьютексу потоков во время разблокировки мьютекса выбирается произвольный из них. расписать функции для работы с ним в С++:

Создать мьютекс: function CreateMutex(

PSecurityAttributes lpMutexAttributes; // Адрес структуры

TSecurityAttributes BOOL bInitialOwner; // Указывает, будет ли процесс владеть мьютексом сразу после создания

PChar lpName // Имя мьютекса );

Получить мьютекс:

function OpenMutex( DWORD dwDesiredAccess; // Задает права доступа к объекту

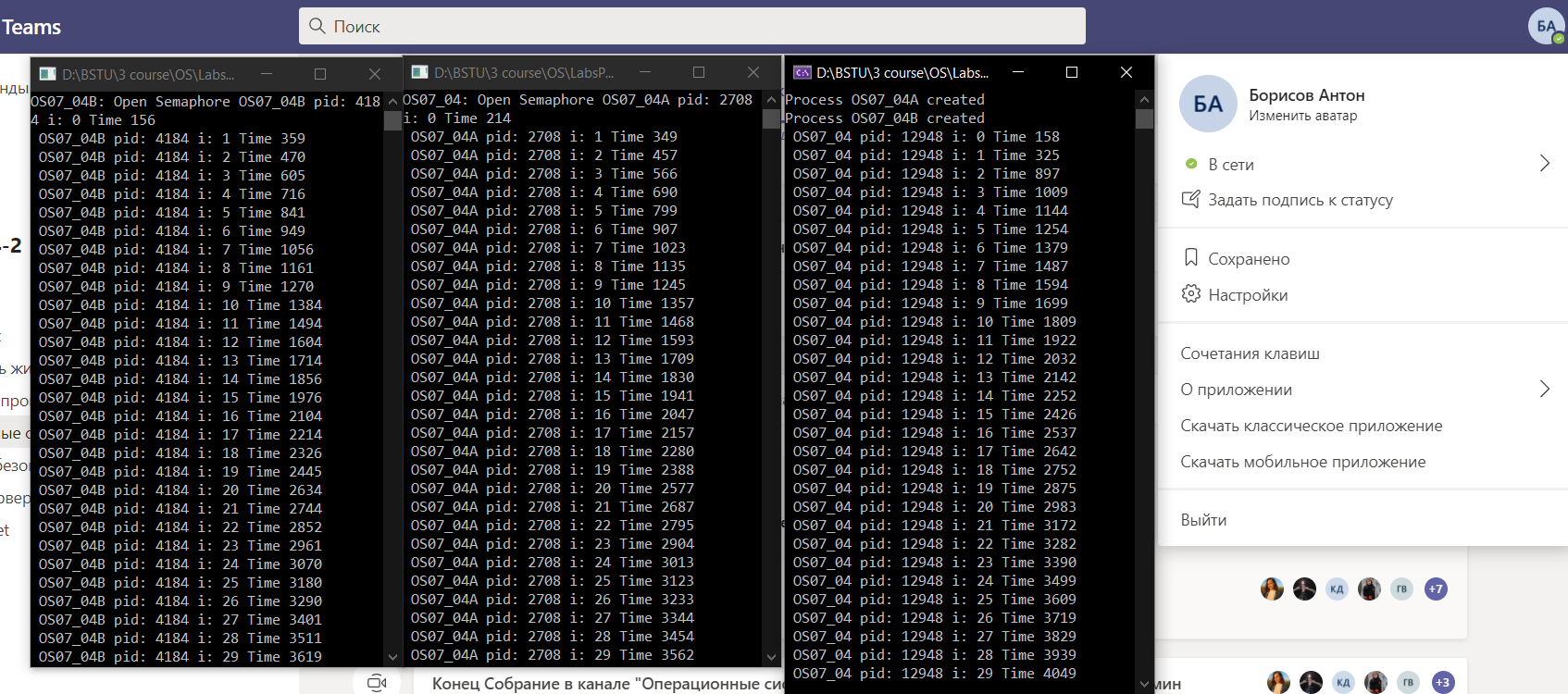
BOOL bInheritHandle; // Задает, может ли объект наследоваться дочерними процессами

PChar lpName; // Имя объекта )

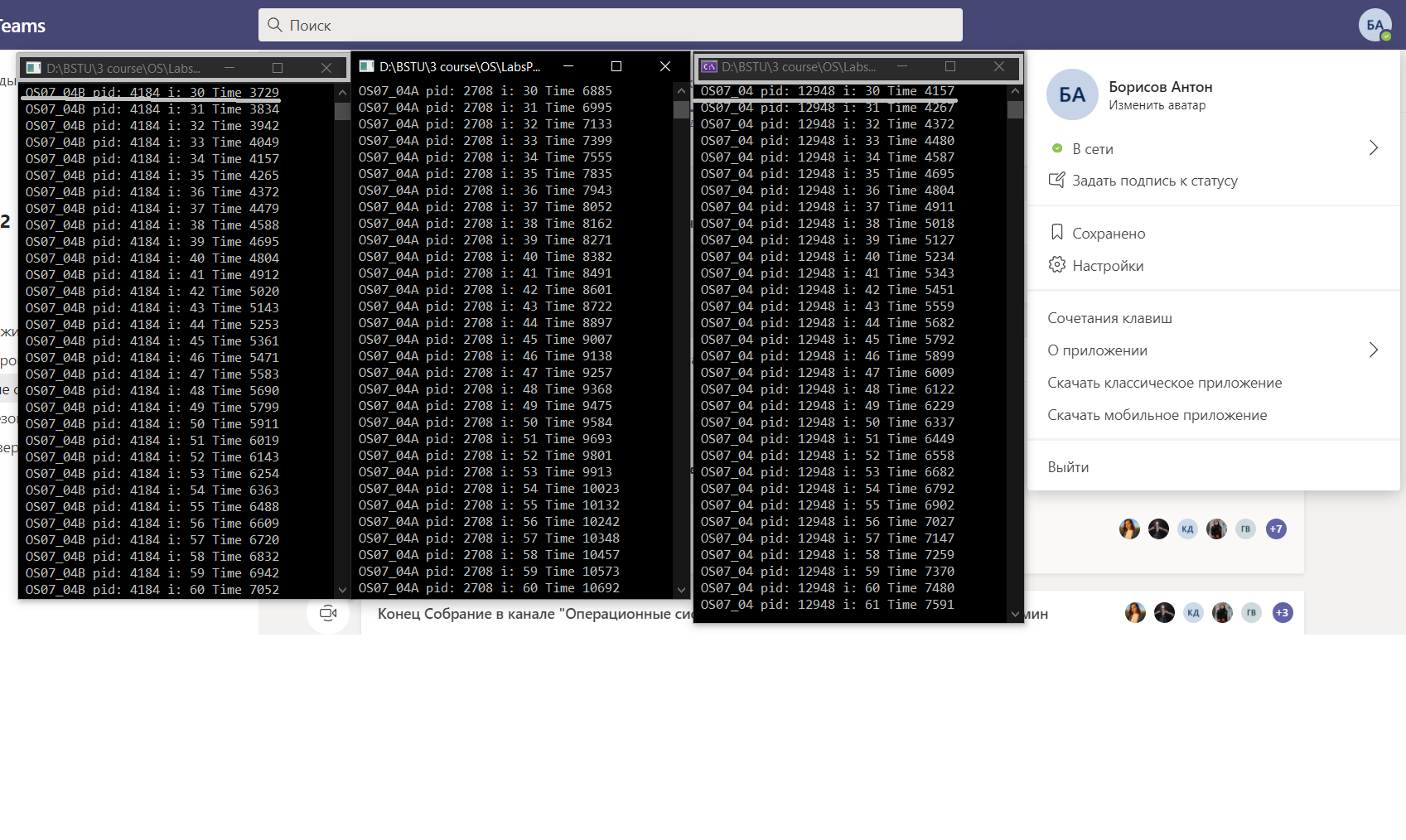
Возвращение мьютекса в несигнальное состояние: function ReleaseMutex(hMutex: THandle)

**Задание-4**

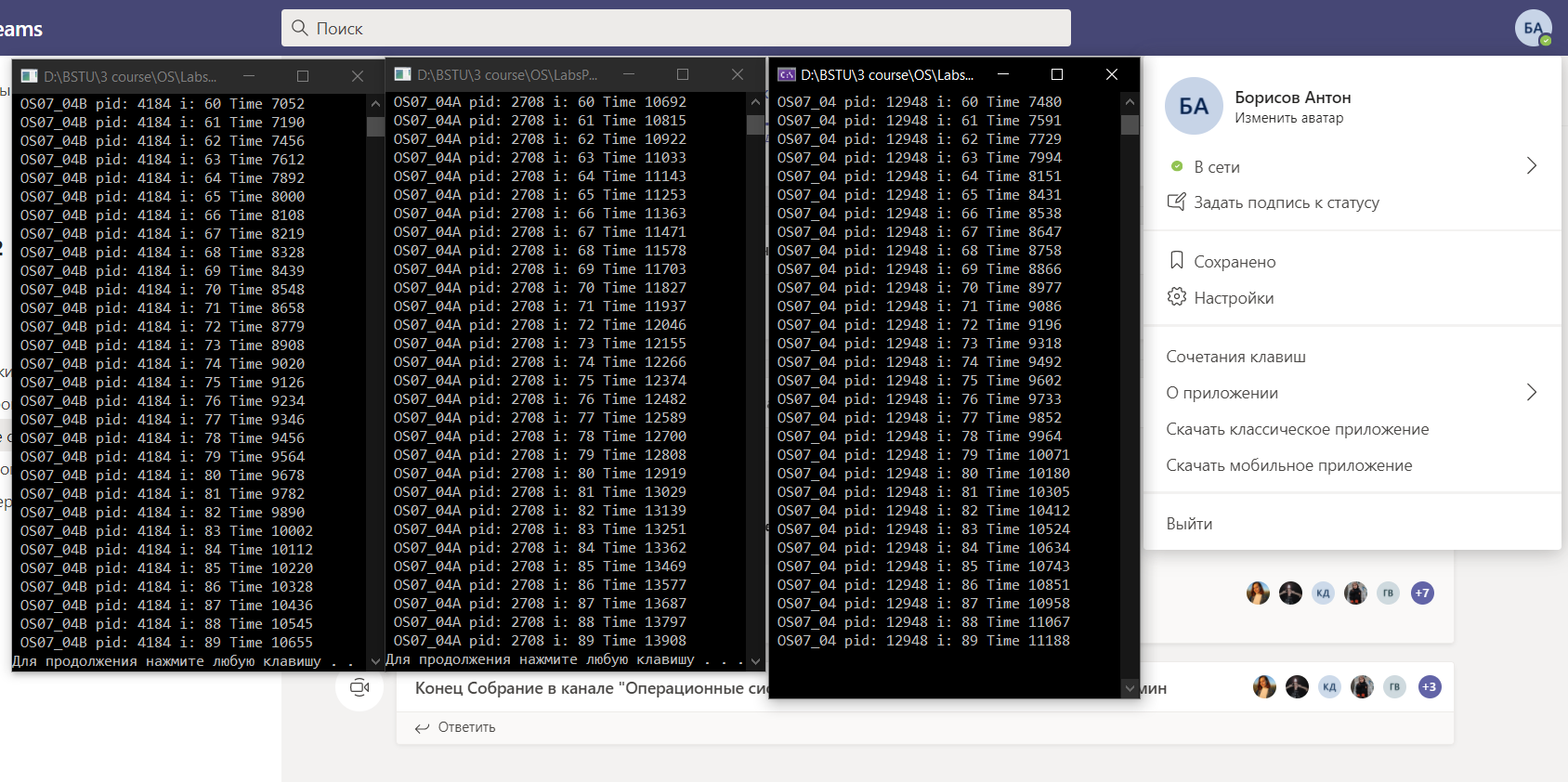
- Скриншот(ы) запуска приложения и результатов до 30 итерации.



- Скриншот(ы) работы приложения между 30 и 60 итерацией.



- Скриншот(ы) работы приложения между после 60 итерации.



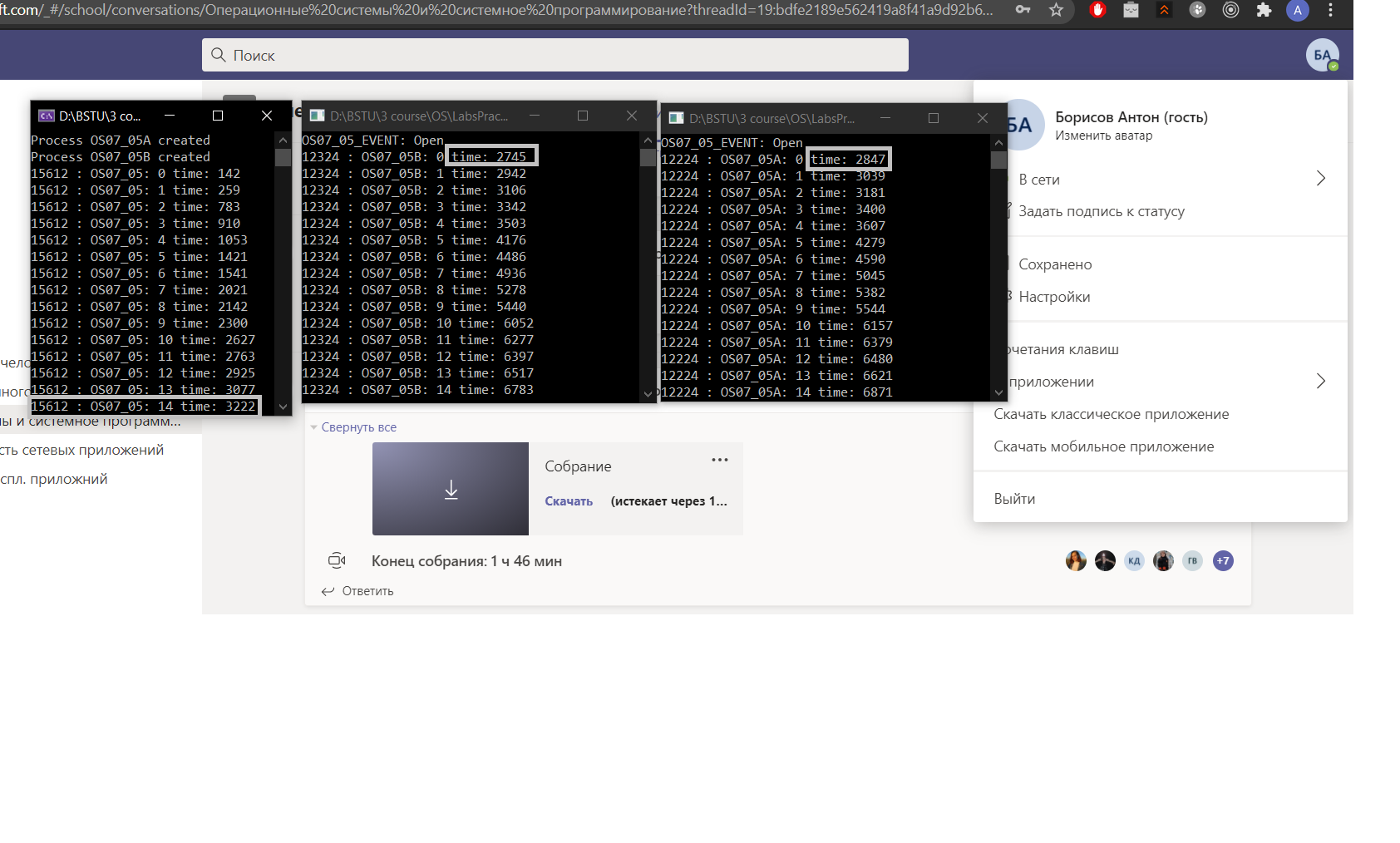
Объяснить принцип работы семафора и расписать функции для работы с ним в С++:

В основе семафора лежит счётчик, над которым можно производить две [атомарные операции](https://ru.wikipedia.org/wiki/%D0%90%D1%82%D0%BE%D0%BC%D0%B0%D1%80%D0%BD%D0%B0%D1%8F_%D0%BE%D0%BF%D0%B5%D1%80%D0%B0%D1%86%D0%B8%D1%8F): увеличение и уменьшение значения на единицу, при этом операция уменьшения для нулевого значения счётчика является блокирующейся.(т.е при счетчике = 0, поток будет ожидать пока счетчик станет > 0)

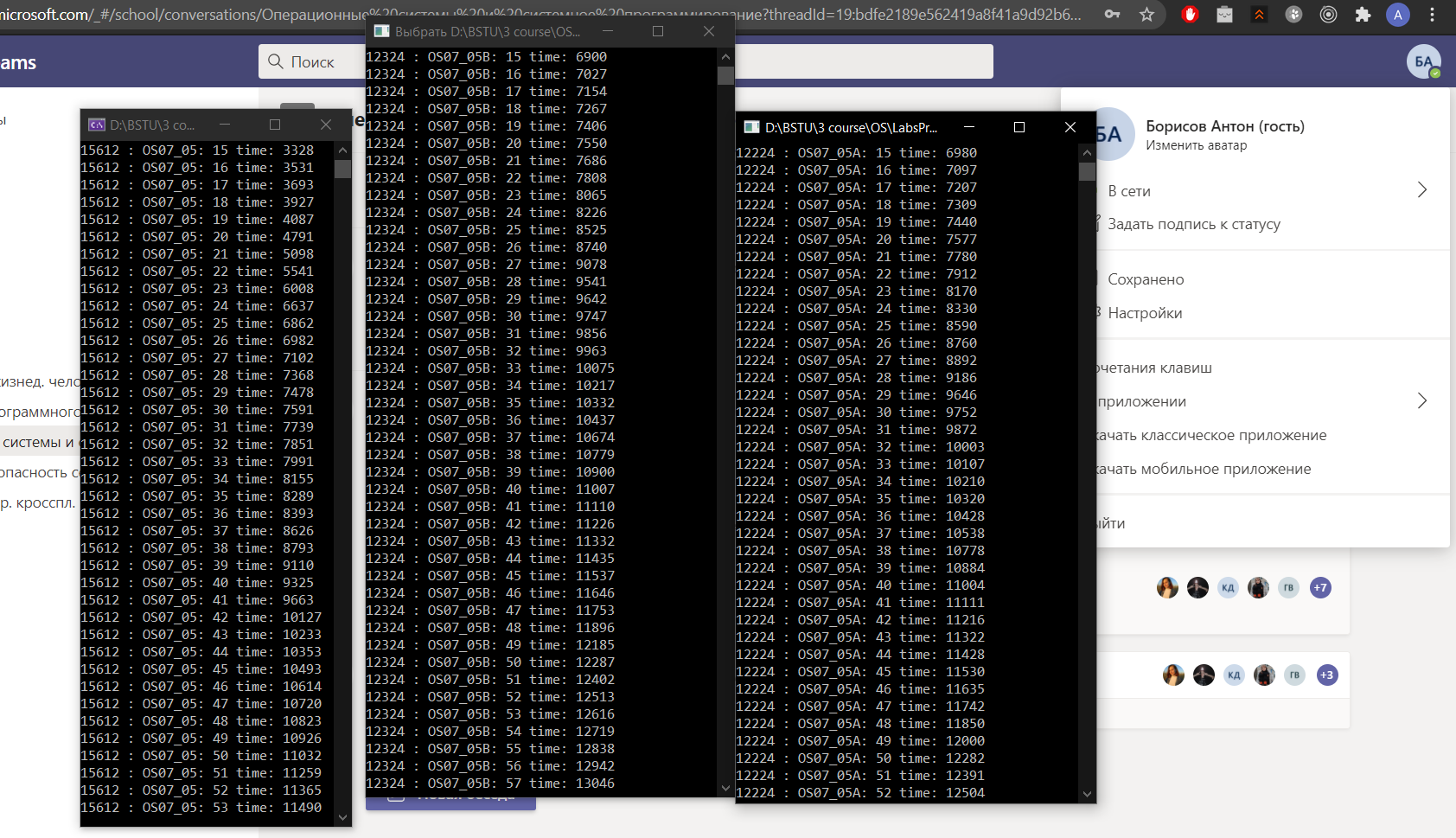
**CreateSemaphore()** – функция создания семафора. Эта функция создает семафор с заданным начальным значением счетчика и максимальным значением. Это значение ограничивает доступ. Функция **OpenSemaphore()** осуществляет доступ к семафору. Функция **ReleaseSemaphore()** увеличивает значение счетчика. Счетчик может меняться от 0 до максимального значения. После завершения работы достаточно вызвать **CloseHandle().** Обращение к семафору происходит на основе функции **WaitForSingleObject**

**Задание 5.**

- Скриншот(ы) запуска приложения и результатов до 15 итерации.



- Скриншот(ы) работы приложения между после 15 итерации.



Объяснить принцип работы событий:

С помощью команды CreateEvent создаётся событие, с помощью команды OpenEvent процессы могут “подписаться на событие" и с помощью функции WaitForSingleObject ожидать его возникновения, команда PulseEvent инициирует событие.

расписать функции для работы с ним в С++:

HANDLE CreateEvent Создает именованный или неназванный объект события.

(

LPSECURITY\_ATTRIBUTES lpEventAttributes, // атрибут защиты

BOOL bManualReset, // тип сброса TRUE - ручной

BOOL bInitialState, // начальное состояние TRUE - сигнальное

LPCTSTR lpName // имя обьекта

);

OpenEvent - Открывает существующий именованный объект события.

HANDLE OpenEventA(

DWORD dwDesiredAccess, // Доступ к объекту события

BOOL bInheritHandle, // Если это значение равно TRUE, то дескриптор унаследуют процессы, созданные этим процессом. В противном случае процессы не наследуют этот дескриптор.

LPCSTR lpName // Название события, которое будет открыто.

);

PulseEvent - Устанавливает указанный объект события в сигнальное состояние, а затем сбрасывает его в несигнальное состояние после освобождения соответствующего количества ожидающих потоков.

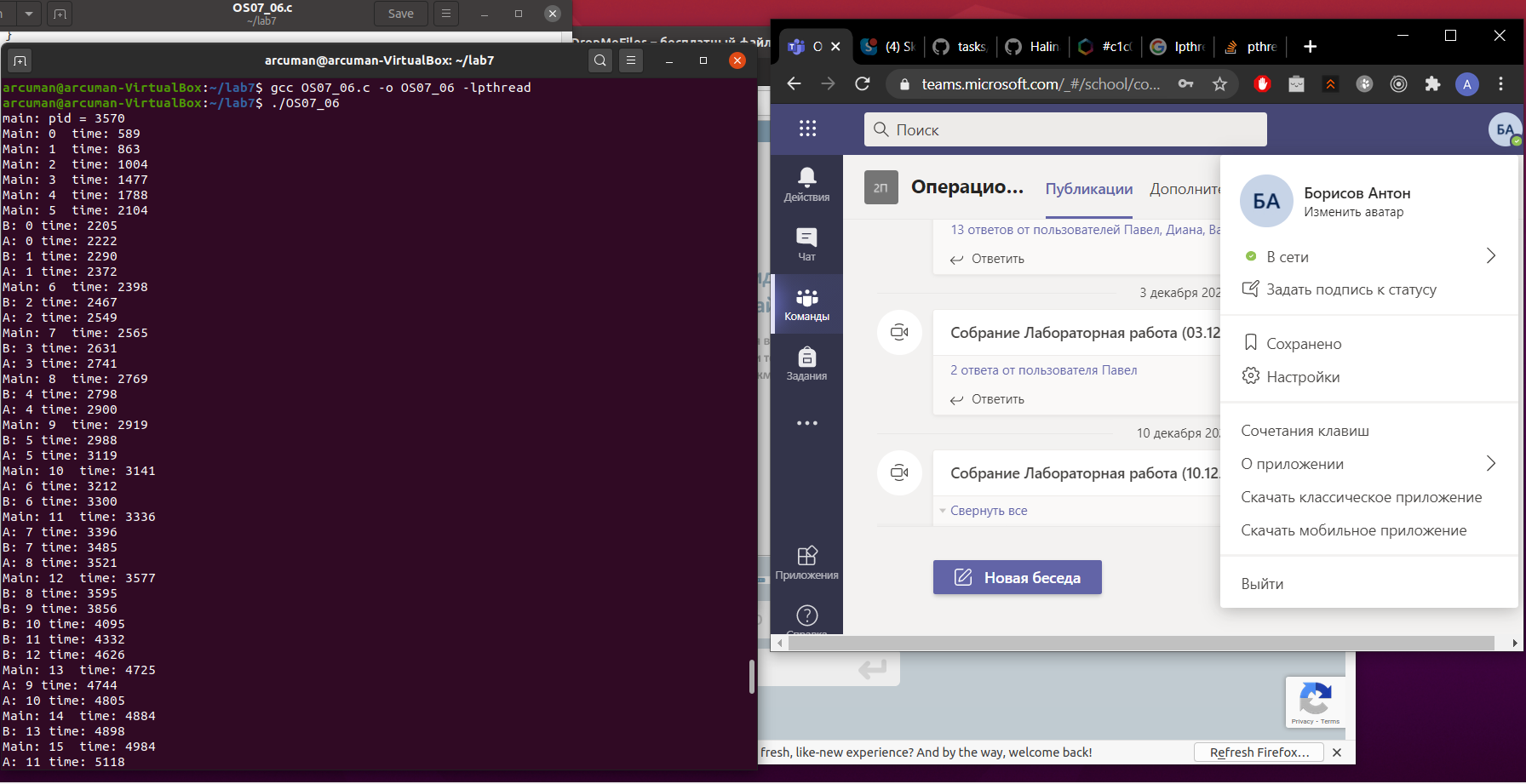
BOOL PulseEvent(

HANDLE hEvent

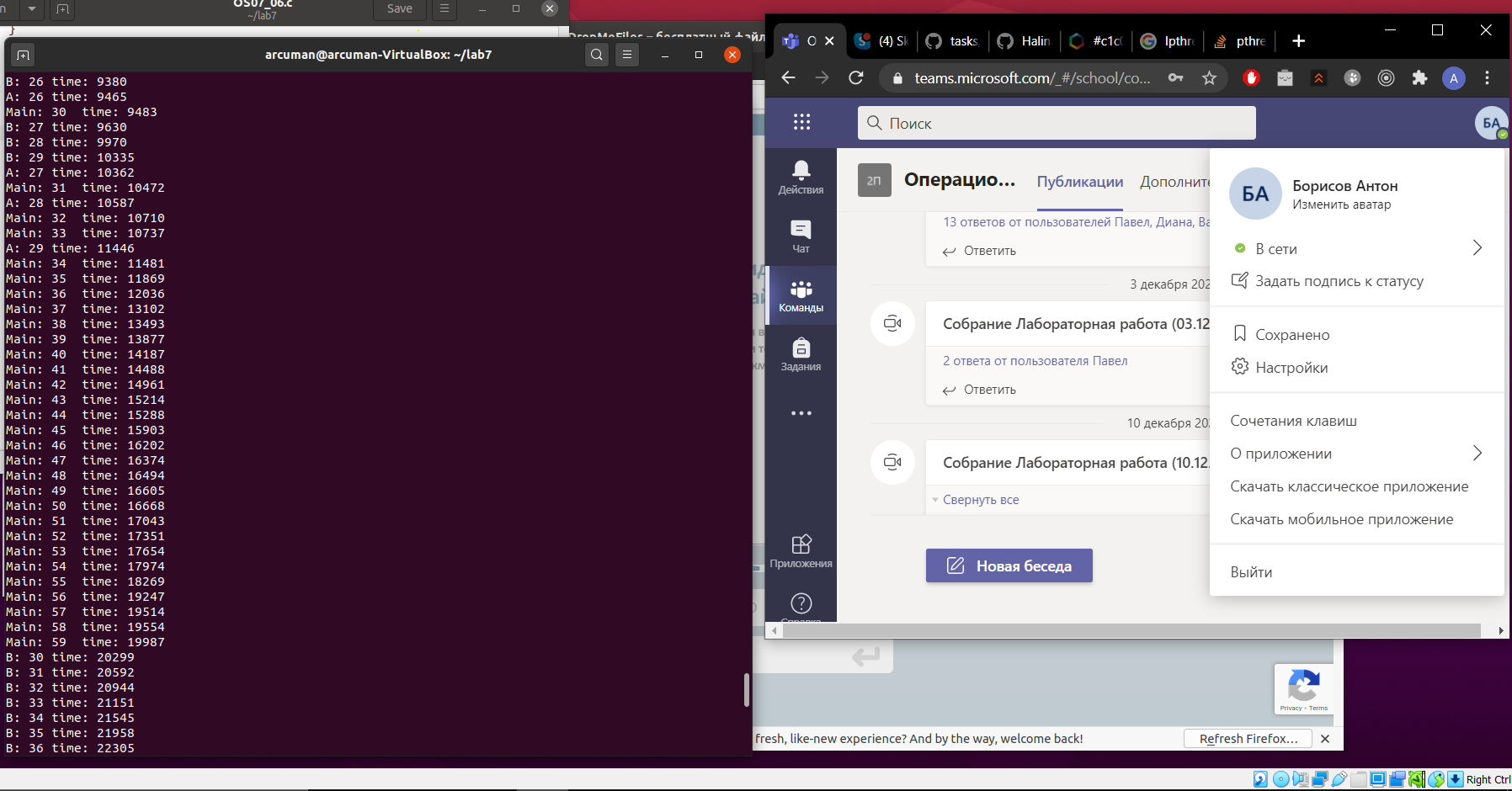
);

Задание 6.

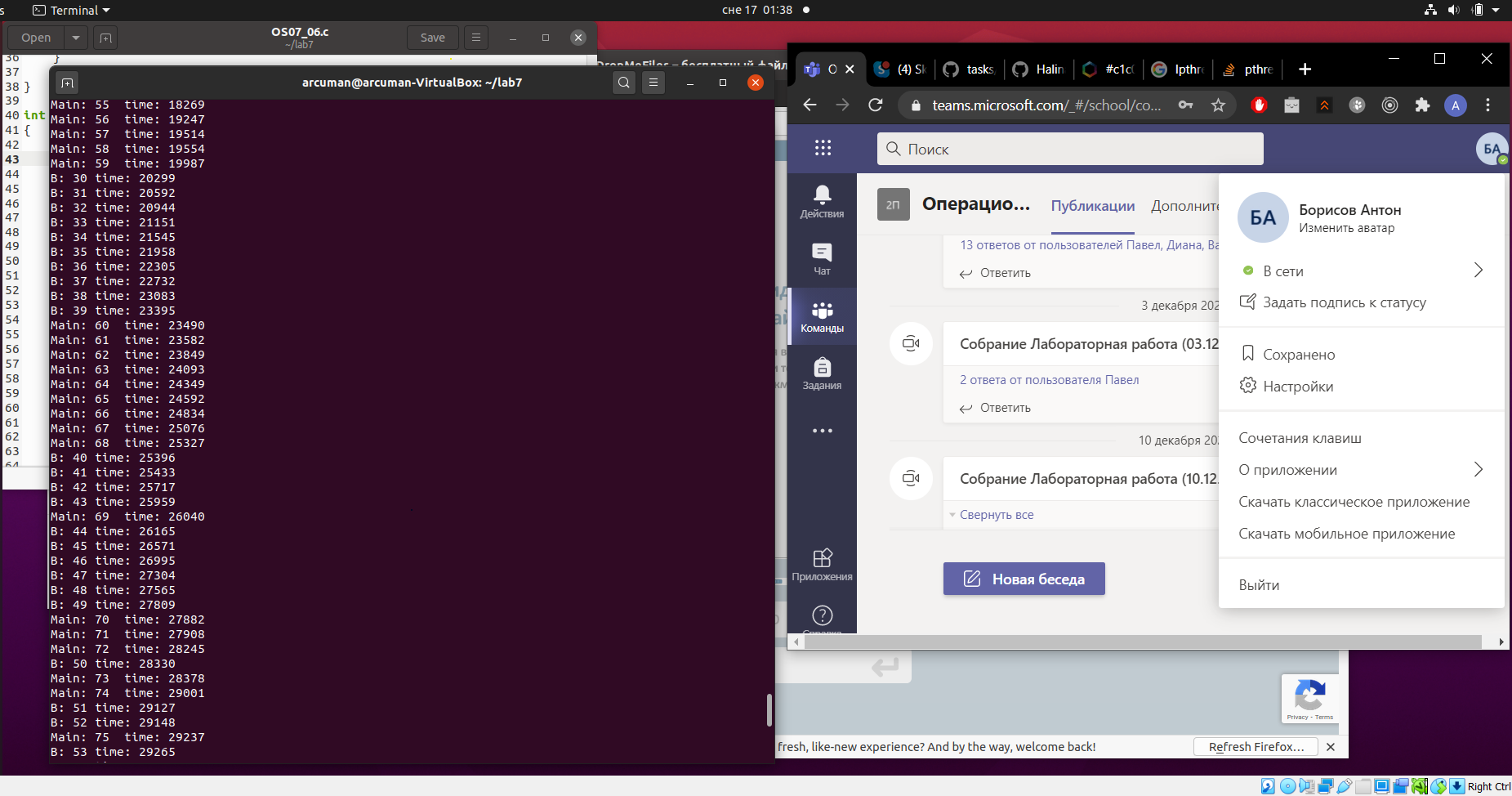
- Скриншот(ы) запуска приложения и результатов до 30 итерации.



- Скриншот(ы) работы приложения между 30 и 60 итерацией.



- Скриншот(ы) работы приложения между после 60 итерации.



Расписать функции для работы с ним в С.

Функция pthread\_mutex\_init(pthread\_mutex\_t \*mutex, const pthread\_mutexattr\_t \*attr) предназначена для инициализации мьютекса. где первый аргумент – указатель на мьютекс, а второй – аттрибуты мьютекса.

После создания мьютекса он может быть захвачен с помощью функции

|  |  |  |
| --- | --- | --- |
| int pthread\_mutex\_lock(pthread\_mutex\_t \*mutex);  После этого участок кода становится недоступным остальным потокам – их выполнение блокируется до тех пор, пока мьютекс не будет освобождён. Освобождение должен провести поток, заблокировавший мьютекс, вызовом   |  |  | | --- | --- | | int pthread\_mutex\_unlock(pthread\_mutex\_t \*mutex);  После использования мьютекса его необходимо уничтожить с помощью функции   |  | | --- | | int pthread\_mutex\_destroy(pthread\_mutex\_t \*mutex); | | |