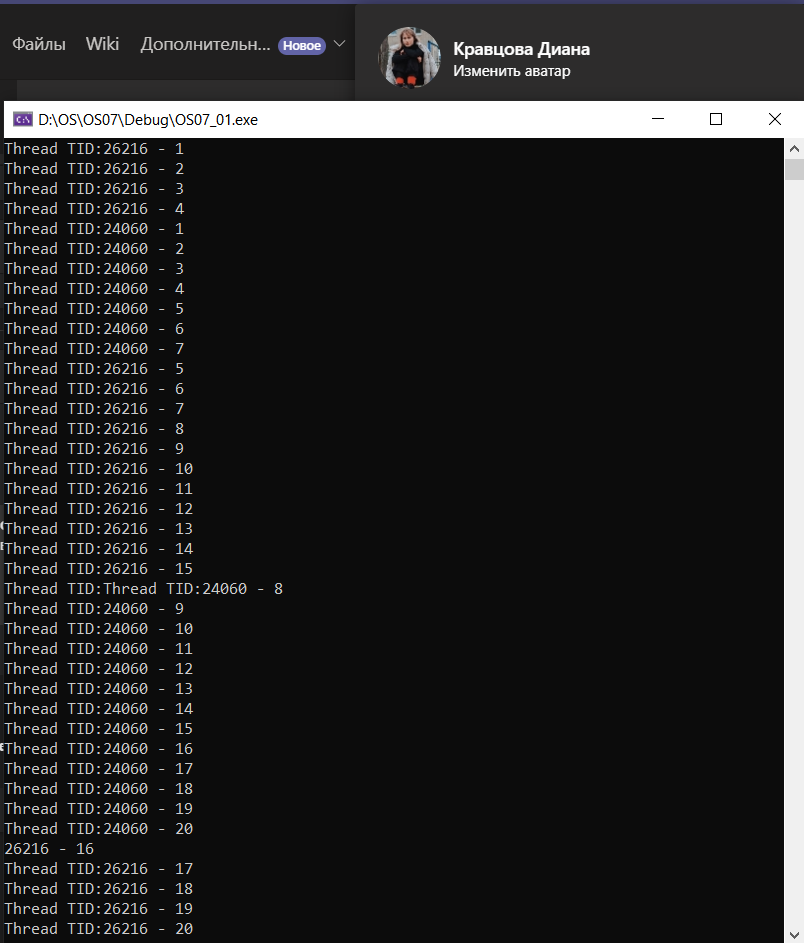
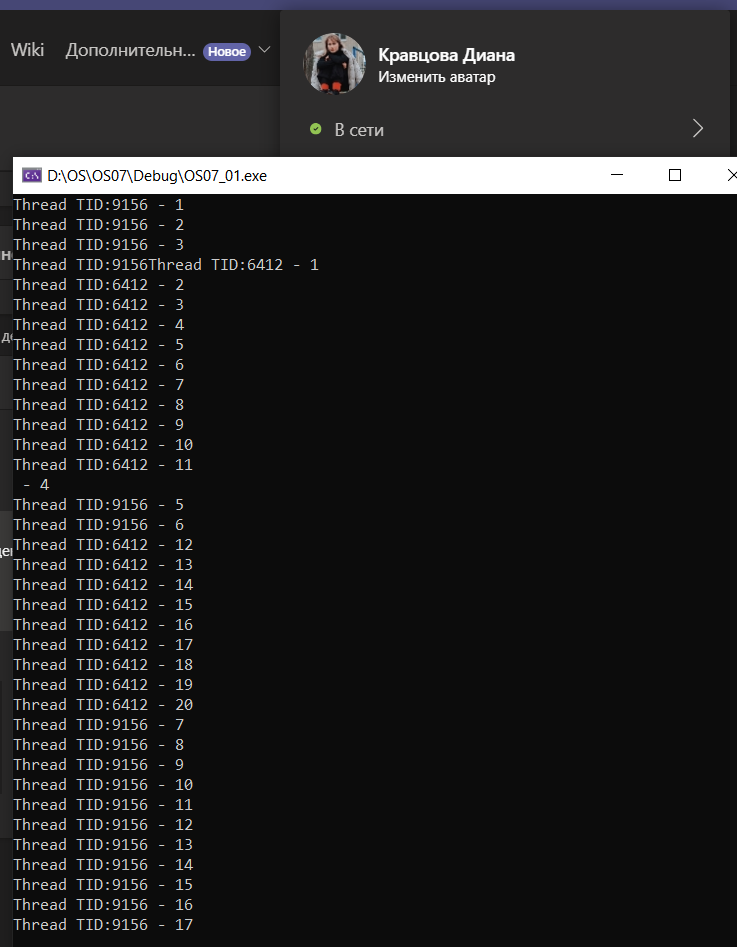
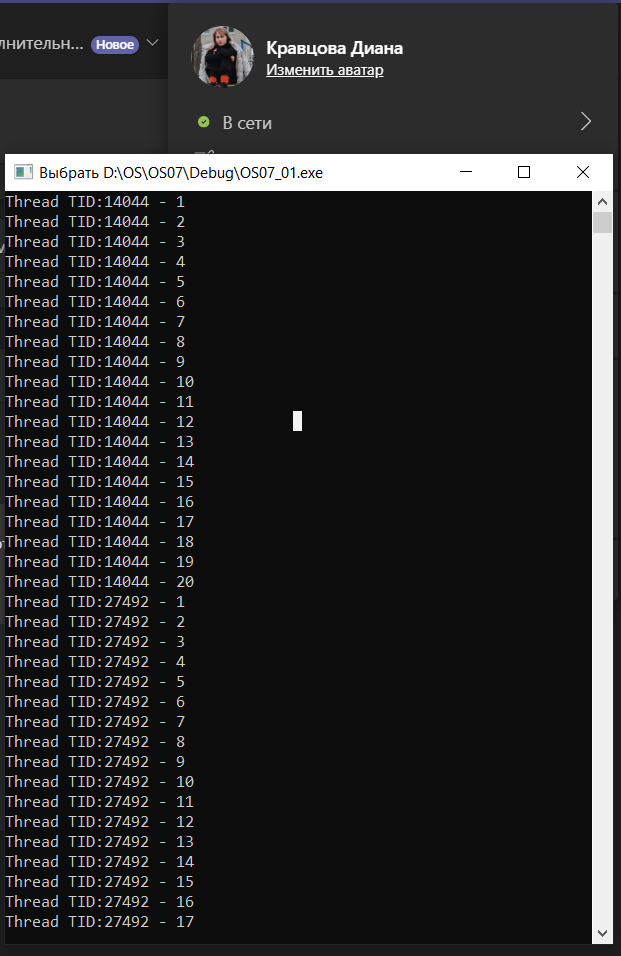
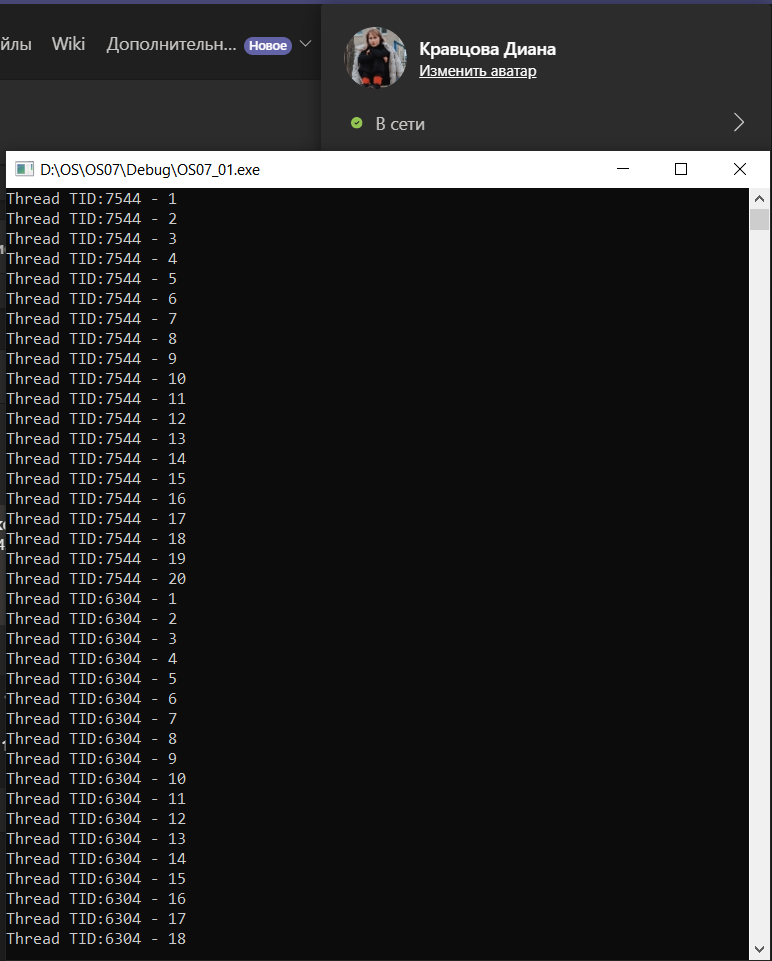
**Задание 1.**

Скриншоты нескольких запусков приложения и результатов без синхронизации.





Скриншоты нескольких запусков приложения и результатов с синхронизацией. 

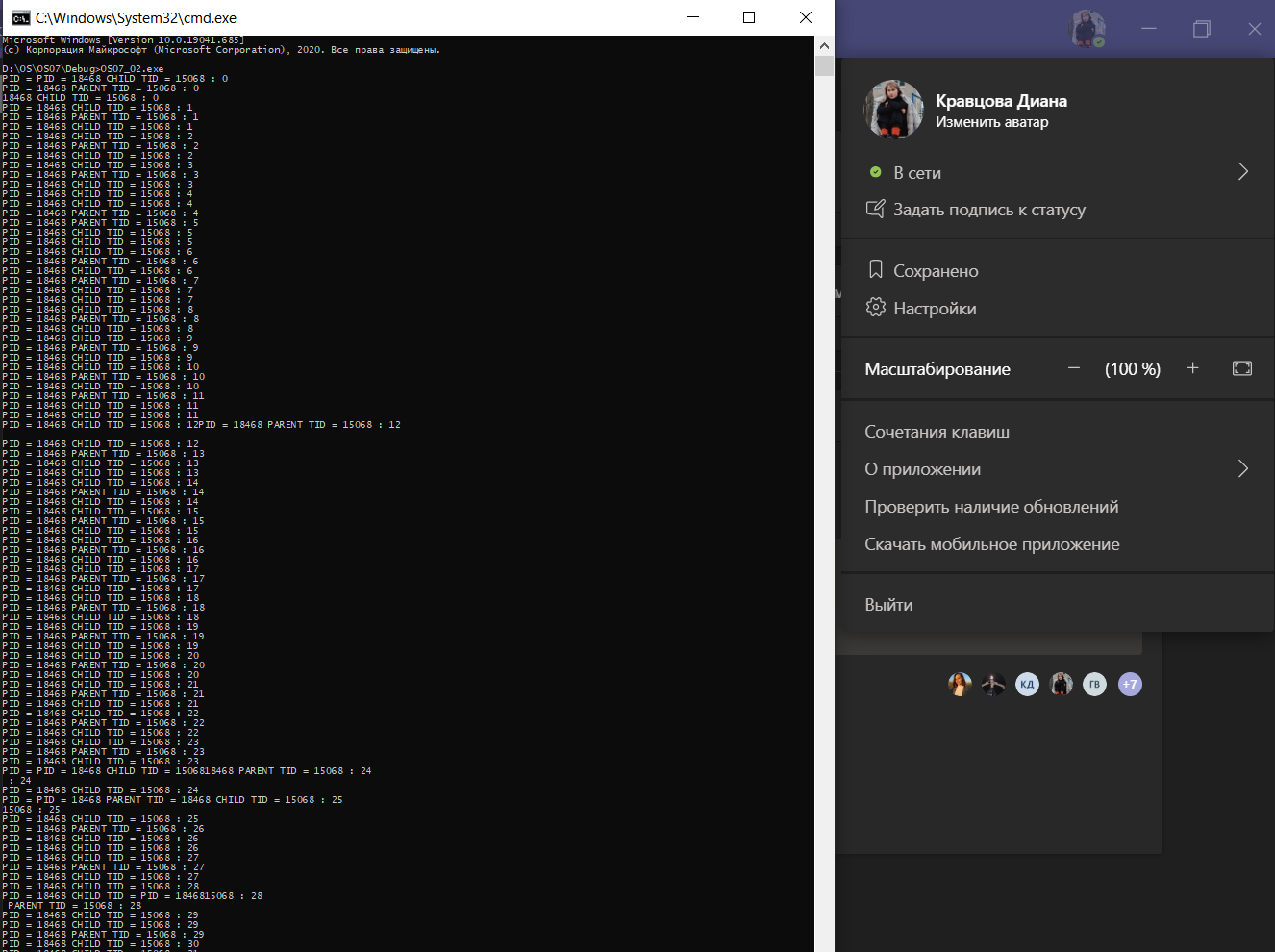


Описать алгоритм работы вашего механизма синхронизации.

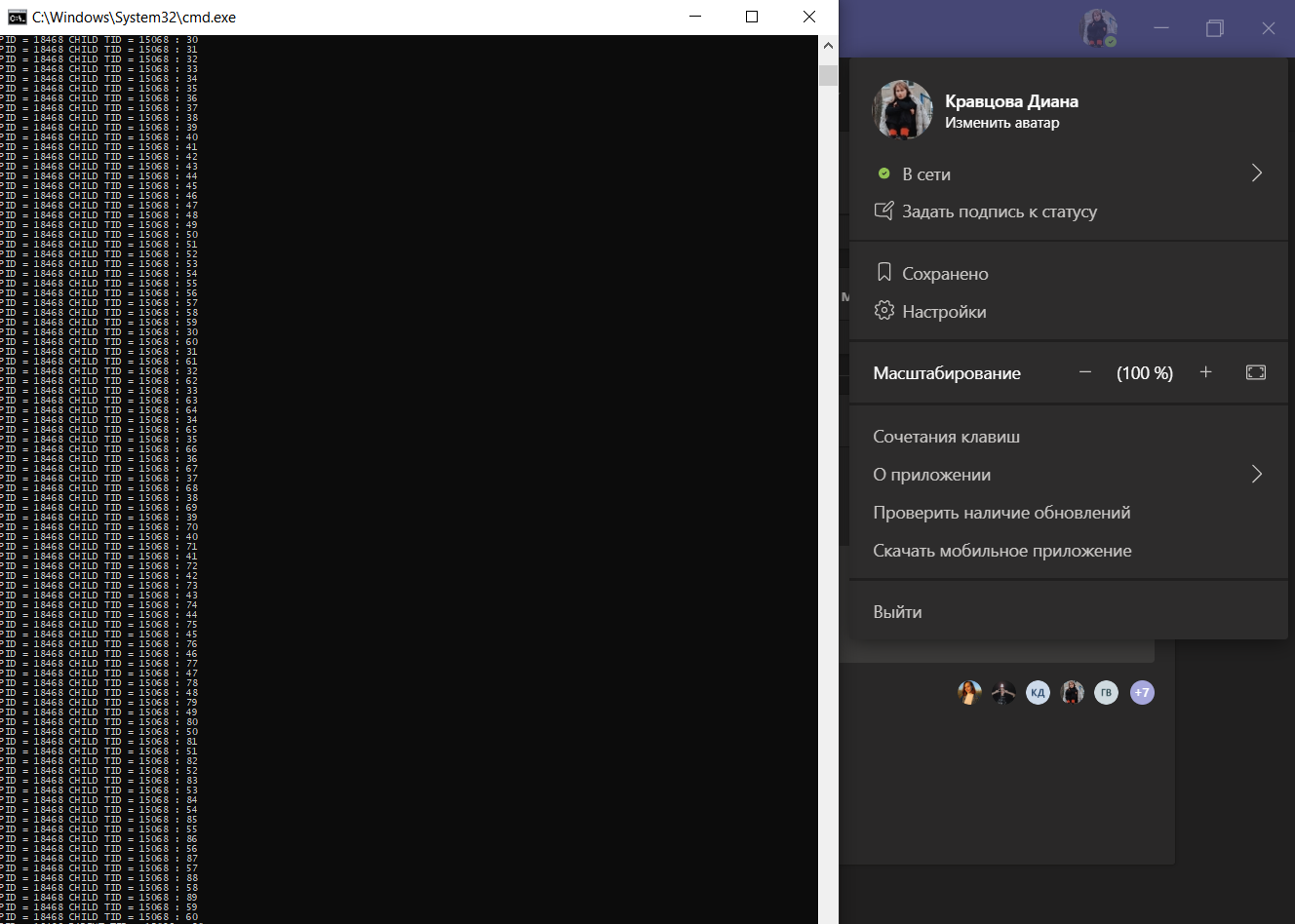
При старте работы отдельного потока с помощью команды ассемблера BTS сохраняем значение из первого операнда(переменная mutex) в флаг CF и устанавливаем значение первого операнда в 1. Дальше мы используем команду jc, которая выполняет переход по метке если в флаге CF содержится единица(то есть, если до 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.**

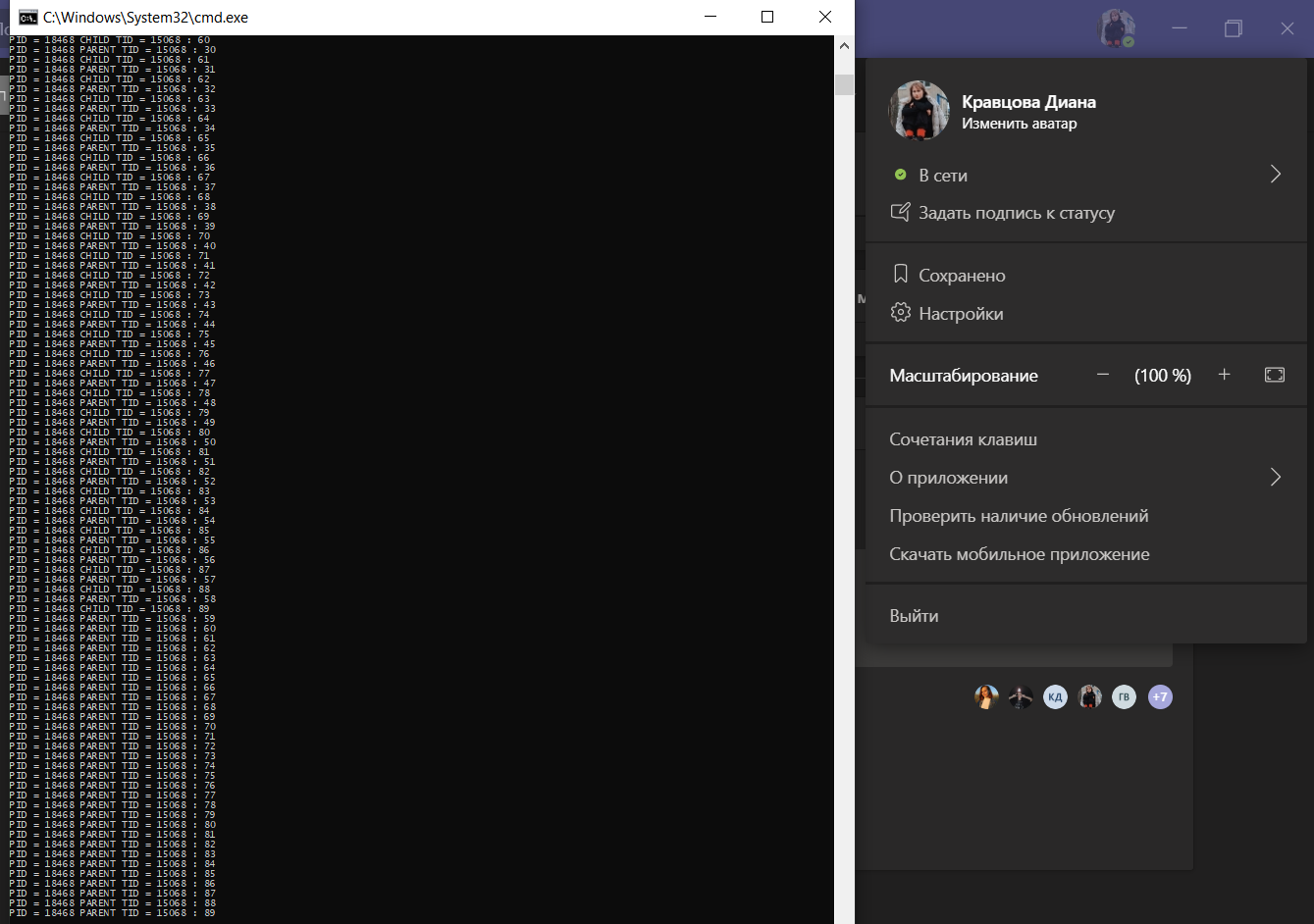
**Скриншот(ы) запуска приложения и результатов до 30 итерации.**

****

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

****

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

****

Объяснить принцип работы секций и расписать функции для работы с ними в С++.

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

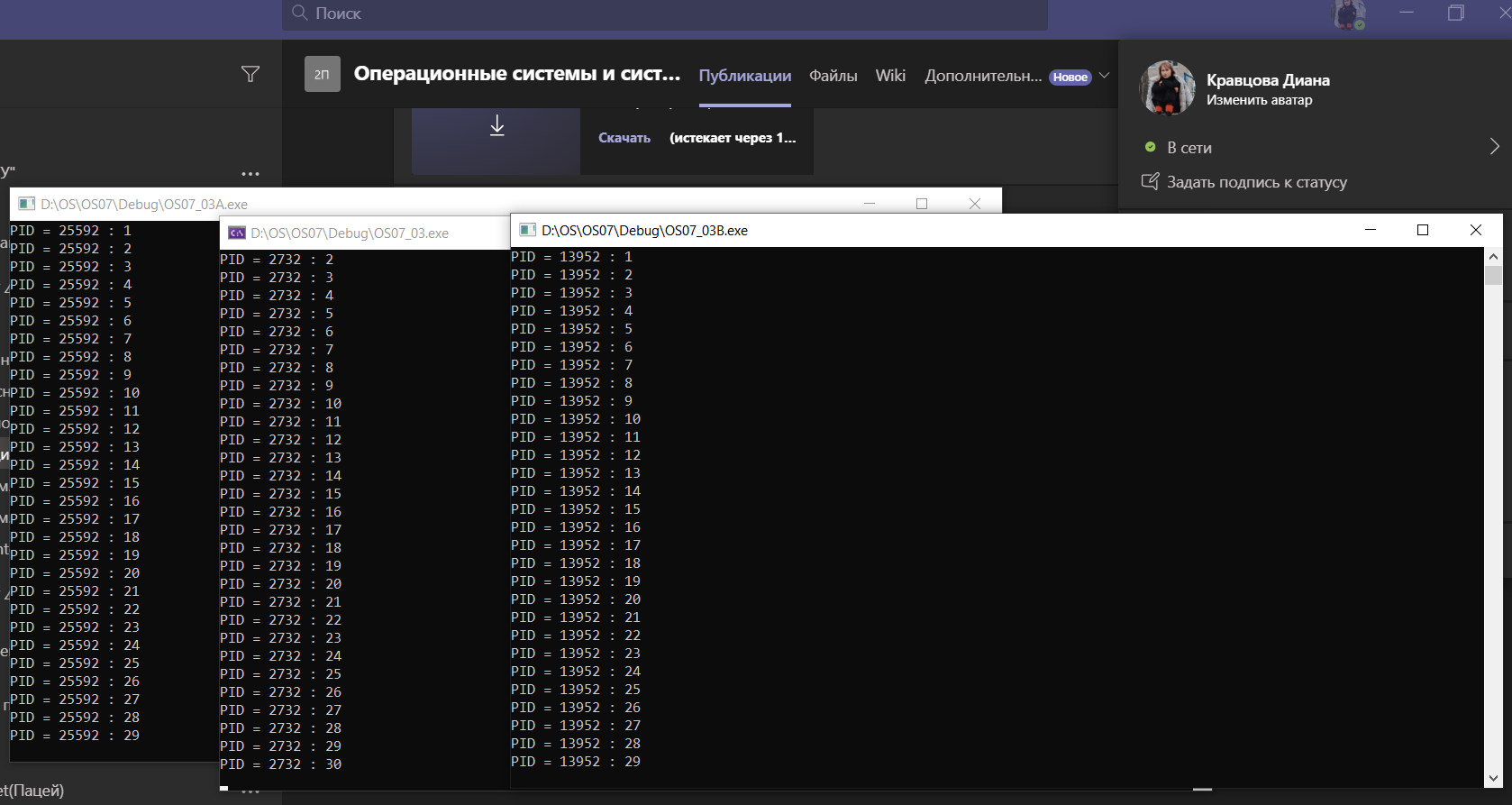
**EnterCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); //функция входа в критическую секция

**LeaveCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); //функция выхода из критической секции

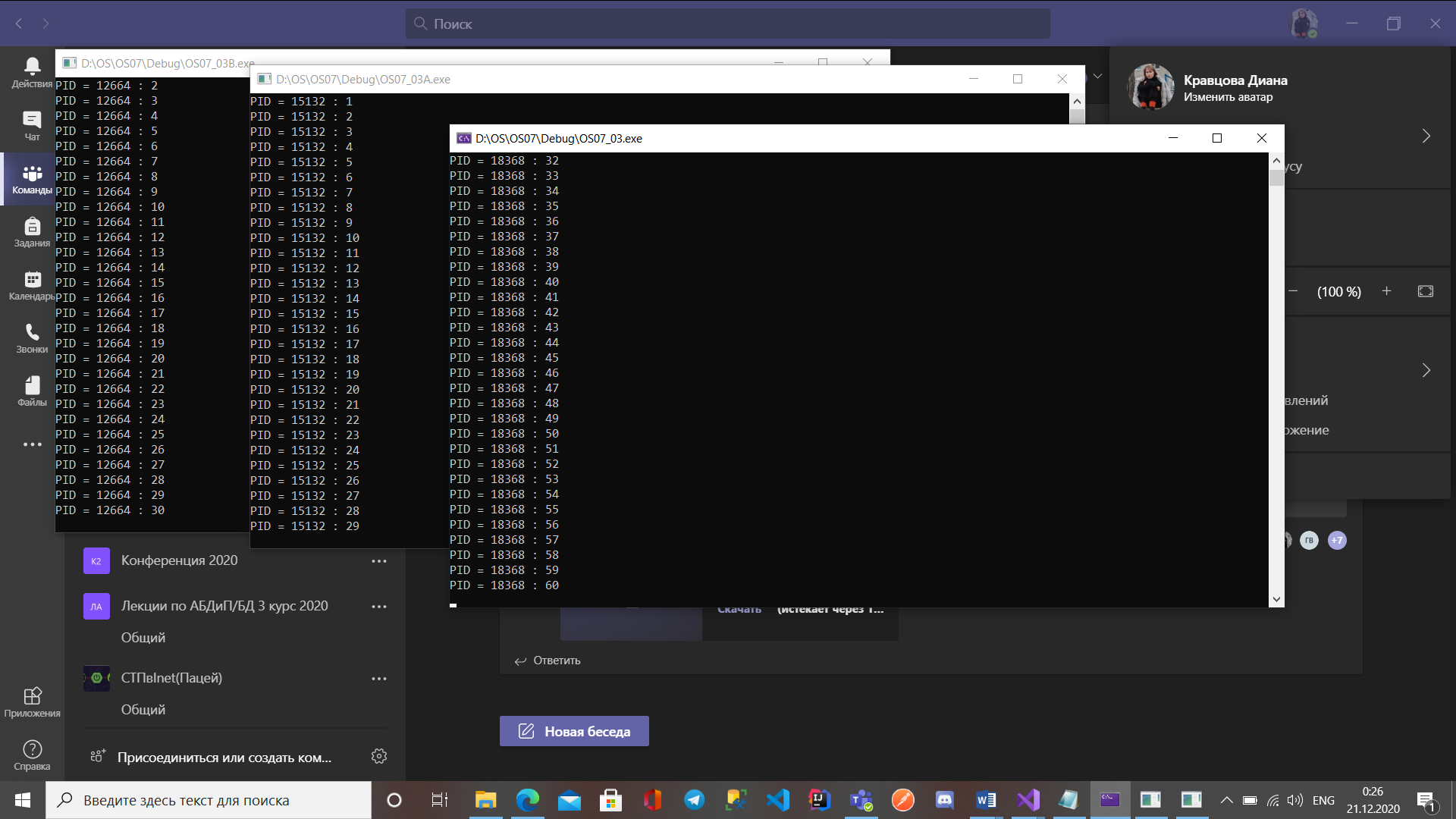
**DeleteCriticalSection**(LPCRITICAL\_SECTION lpCriticalSection); //Удаление(деинициализация) критической секции

**Задание 3.**

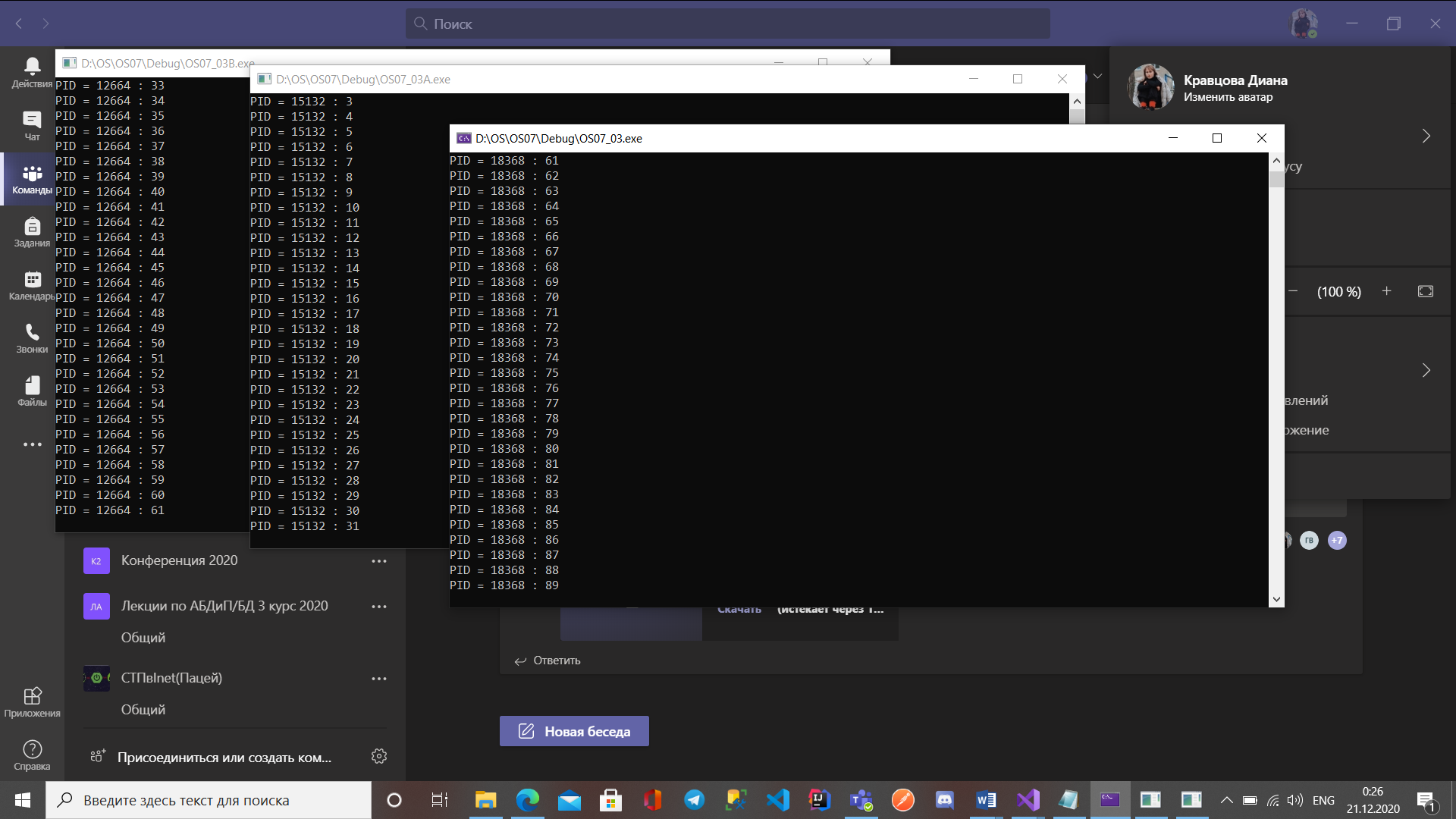
**Скриншот(ы) запуска приложения и результатов до 30 итерации.**



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



Скриншот(ы) работы приложения между после 60 итерации.



Объяснить принцип работы мьютекса и расписать функции для работы с ним в С++.

**Мьютекс** — это совместно используемая переменная, которая может находиться в одном из двух состояний: заблокированном или незаблокированном. Следовательно,

для их представления нужен только один бит, но на практике зачастую используется

целое число, при этом нуль означает незаблокированное, а все остальные значения —

заблокированное состояние. 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.

**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

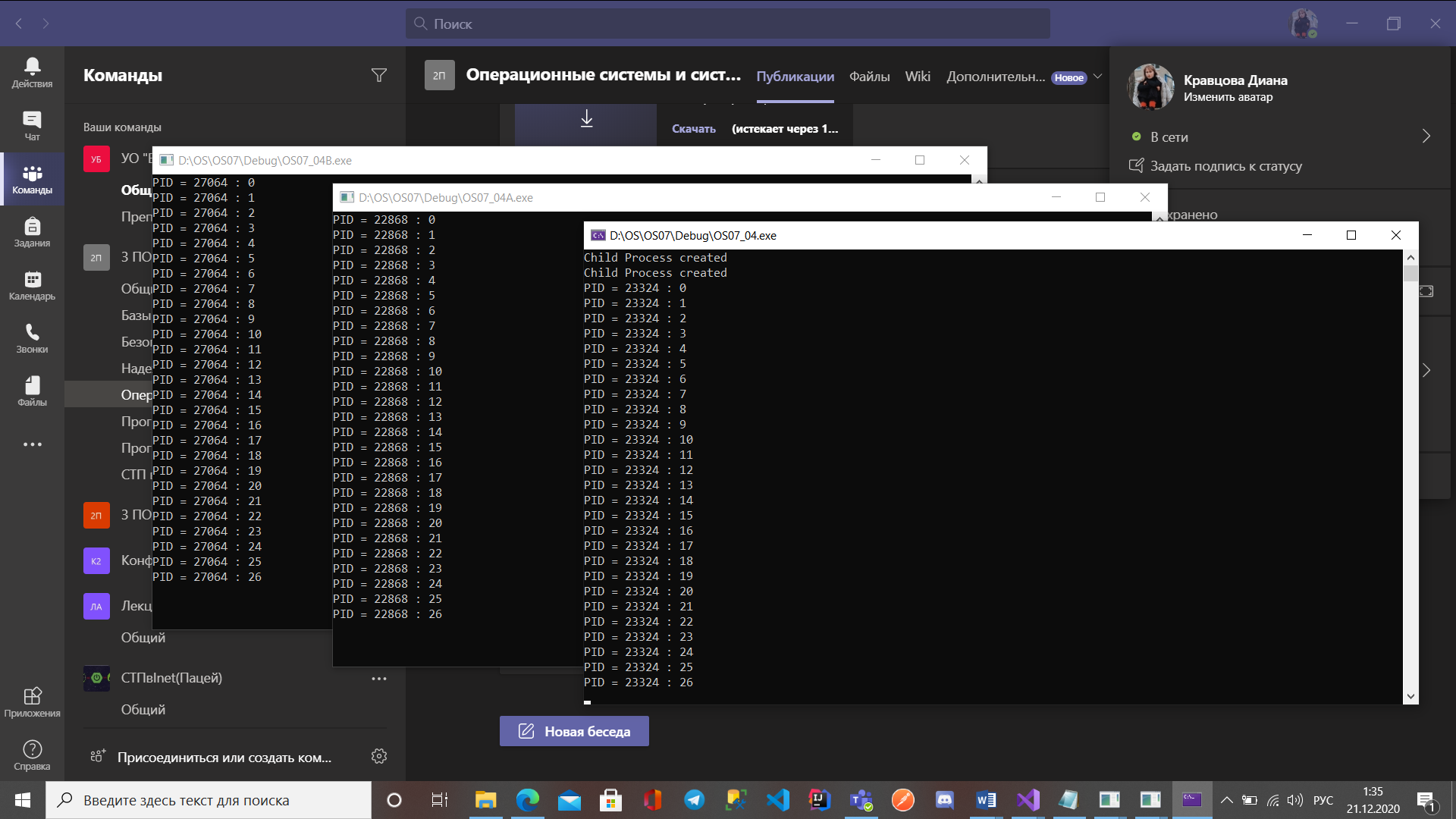
):

**Возвращение мьютекса в несигнальное состояние:**

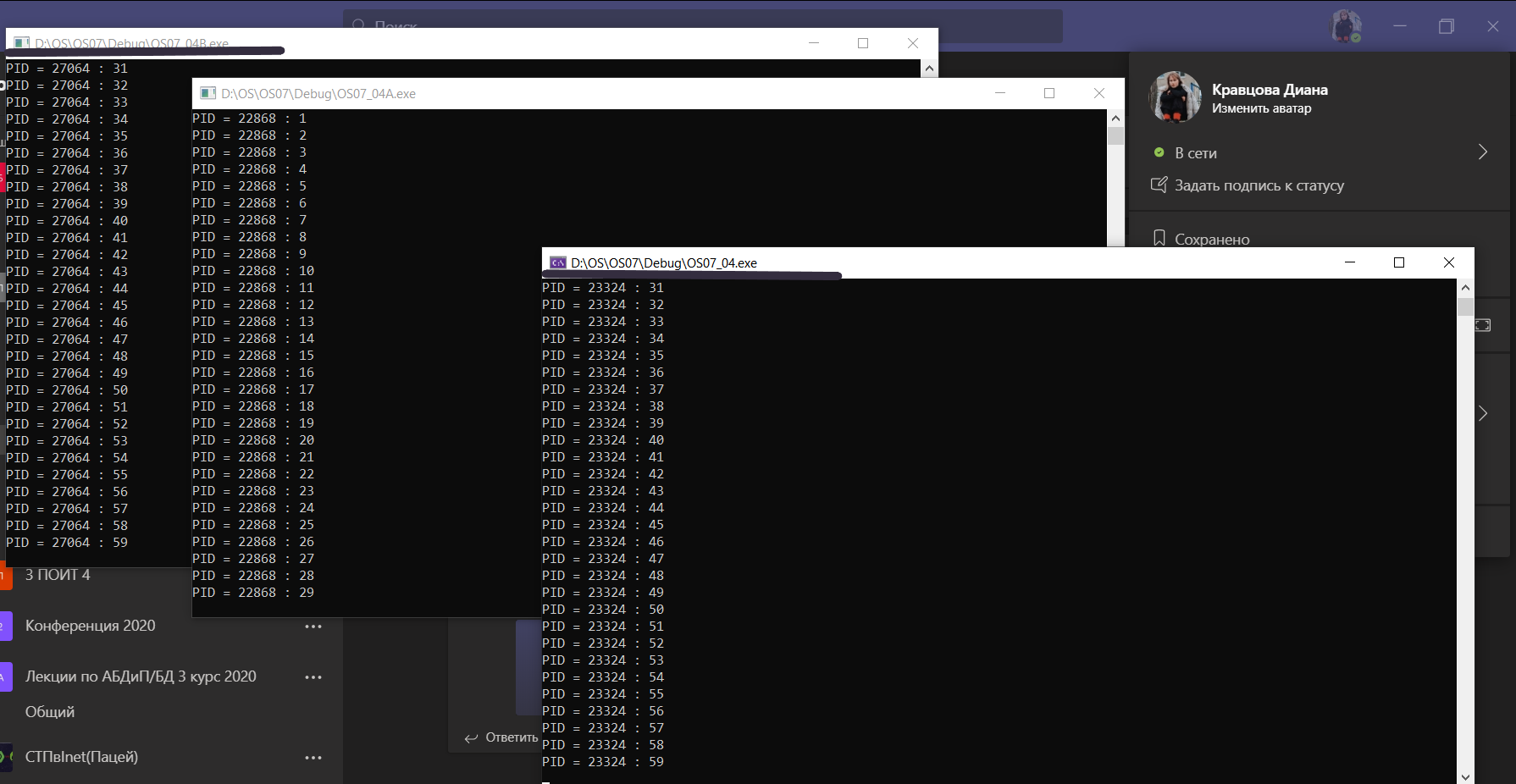
function **ReleaseMutex**(hMutex: THandle)

**Задание 4.**

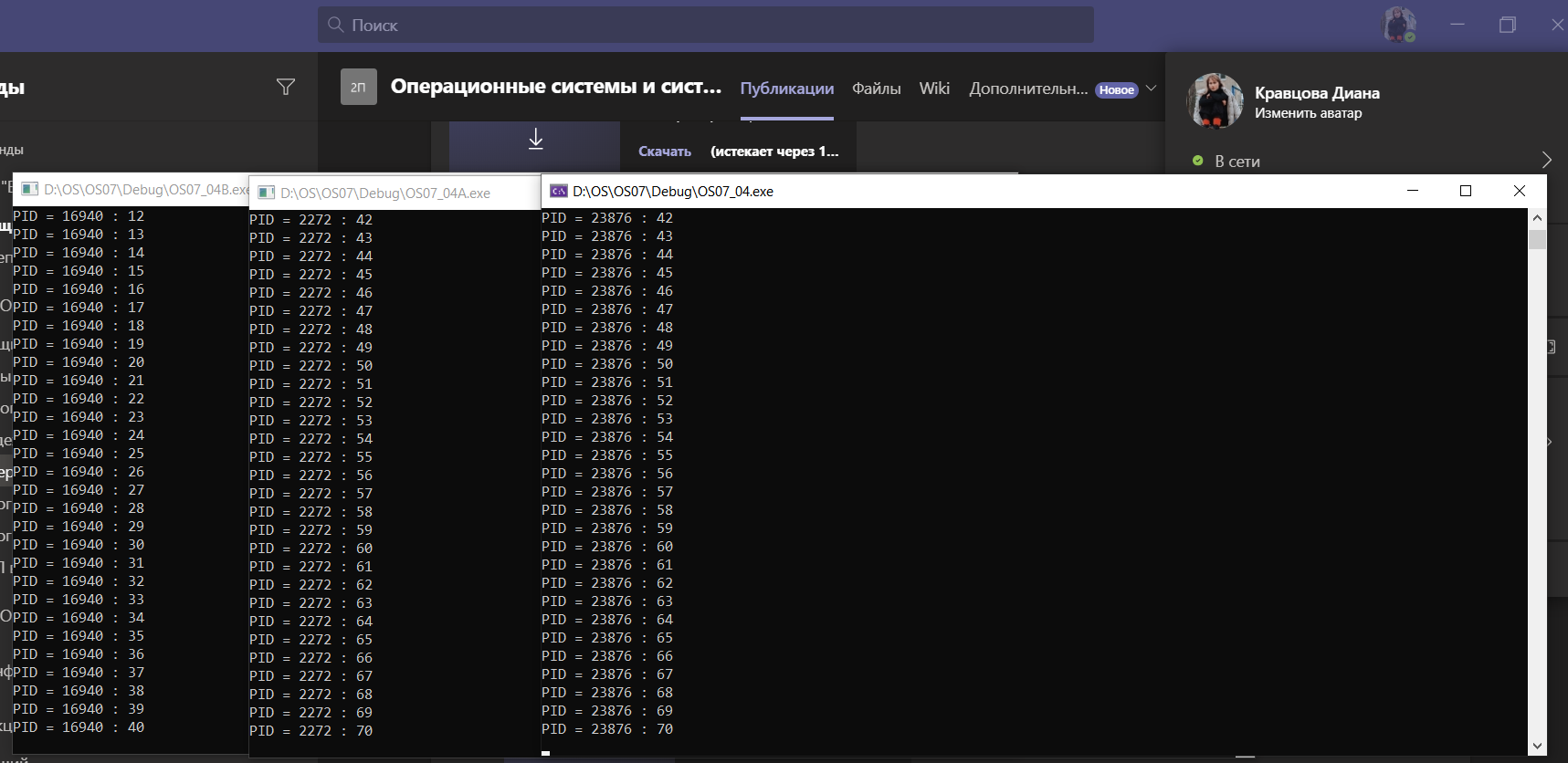
Скриншот(ы) запуска приложения и результатов до 30 итерации.



**Скриншот(ы) работы приложения между 30 и 60 итерацией. На скриншоте(ах) второго пункта выделить имена работающих одновременно двух процессов.**



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

****

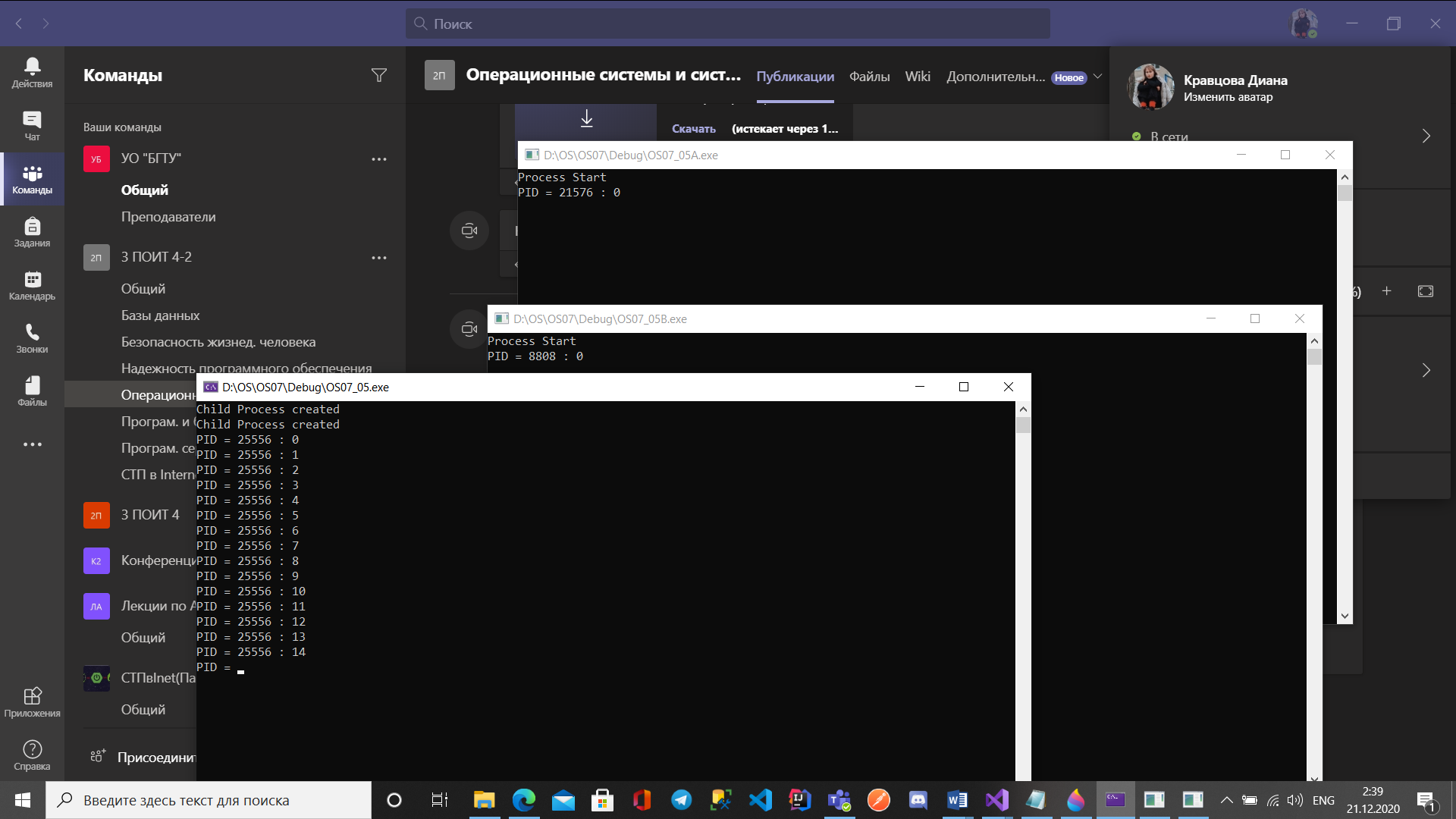
Объяснить принцип работы семафора и расписать функции для работы с ним в С++.

В основе семафора лежит счётчик, над которым можно производить две атомарные операции: 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**().

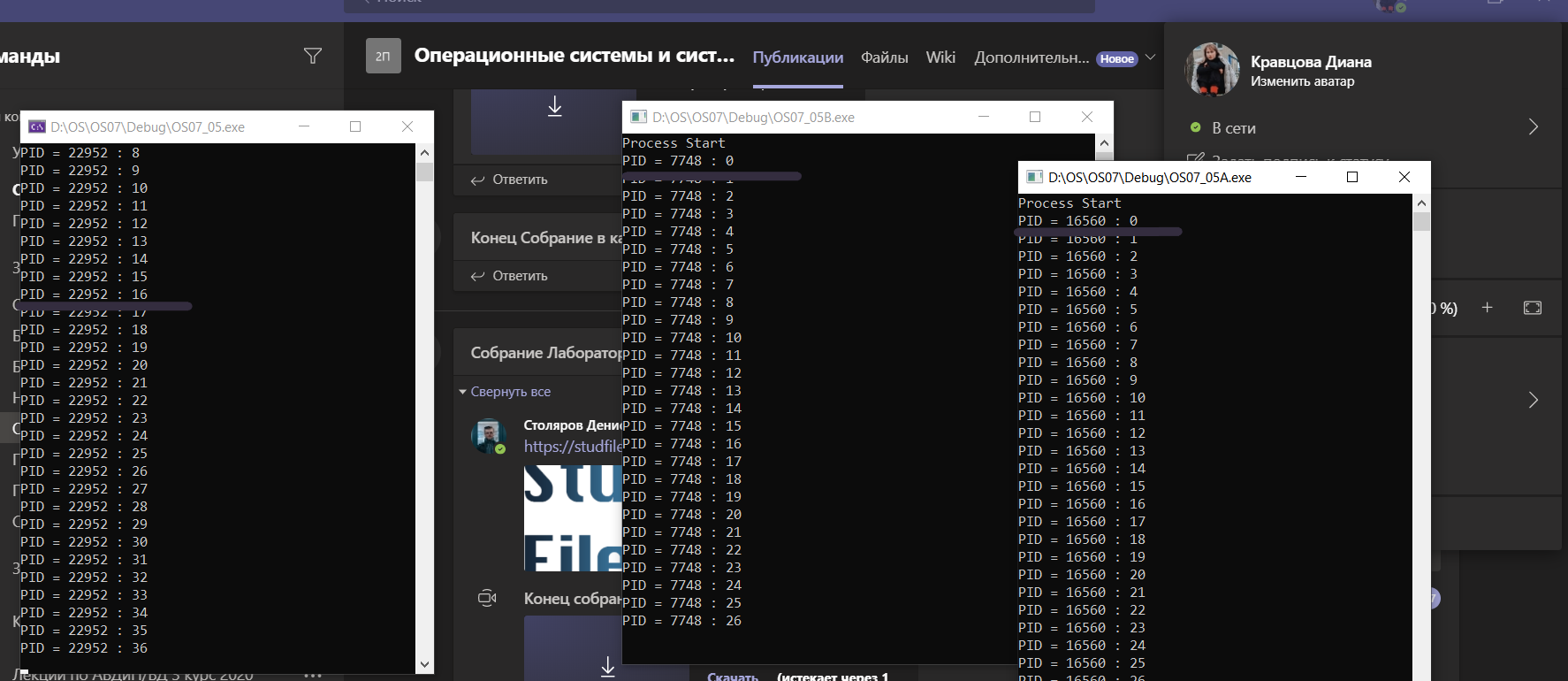
**Задание 5.**

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



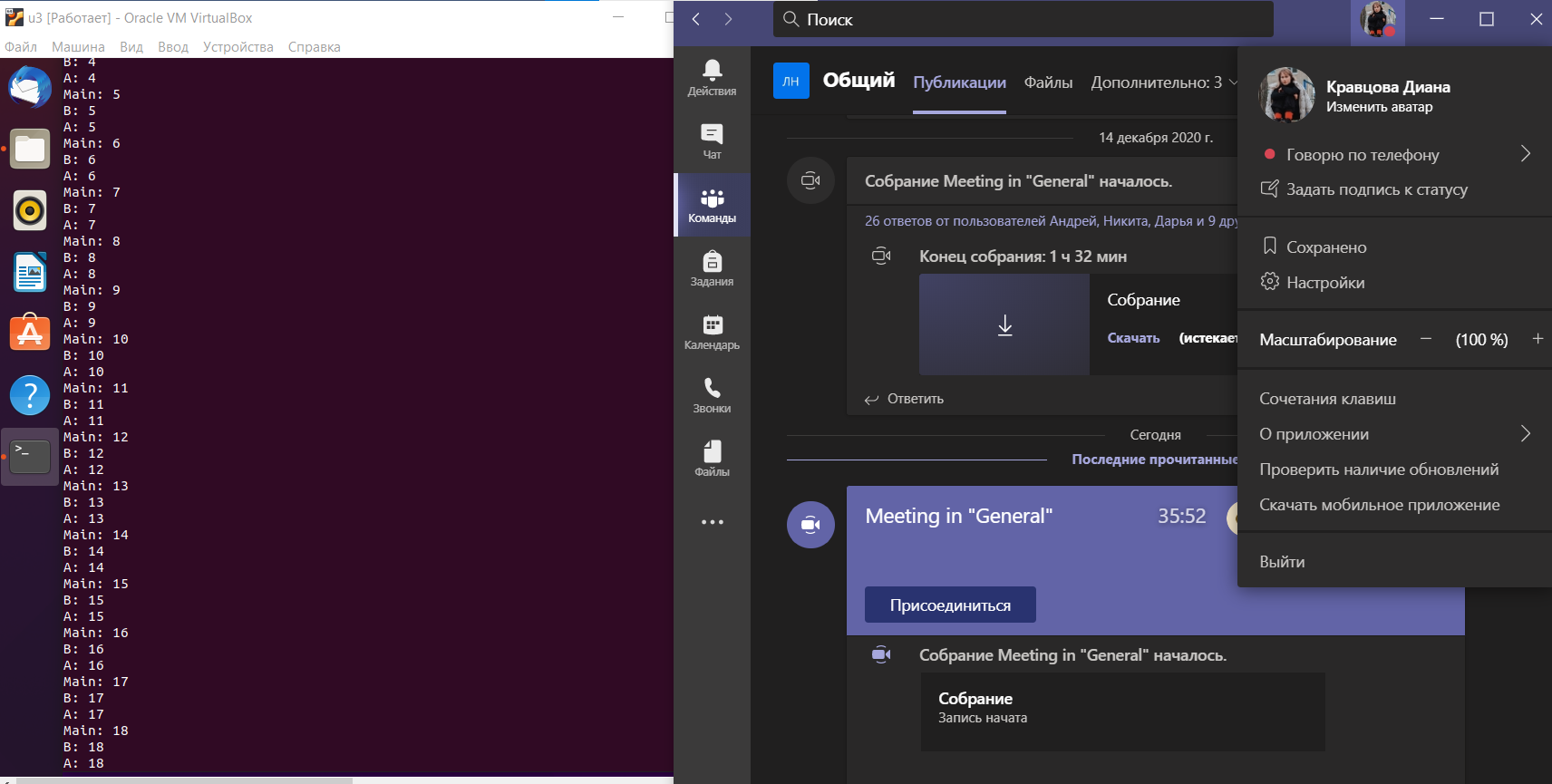
**Скриншот(ы) работы приложения между после 15 итерации.**

**На 2 скриншоте выделить момент начала работы двух других процессов.**

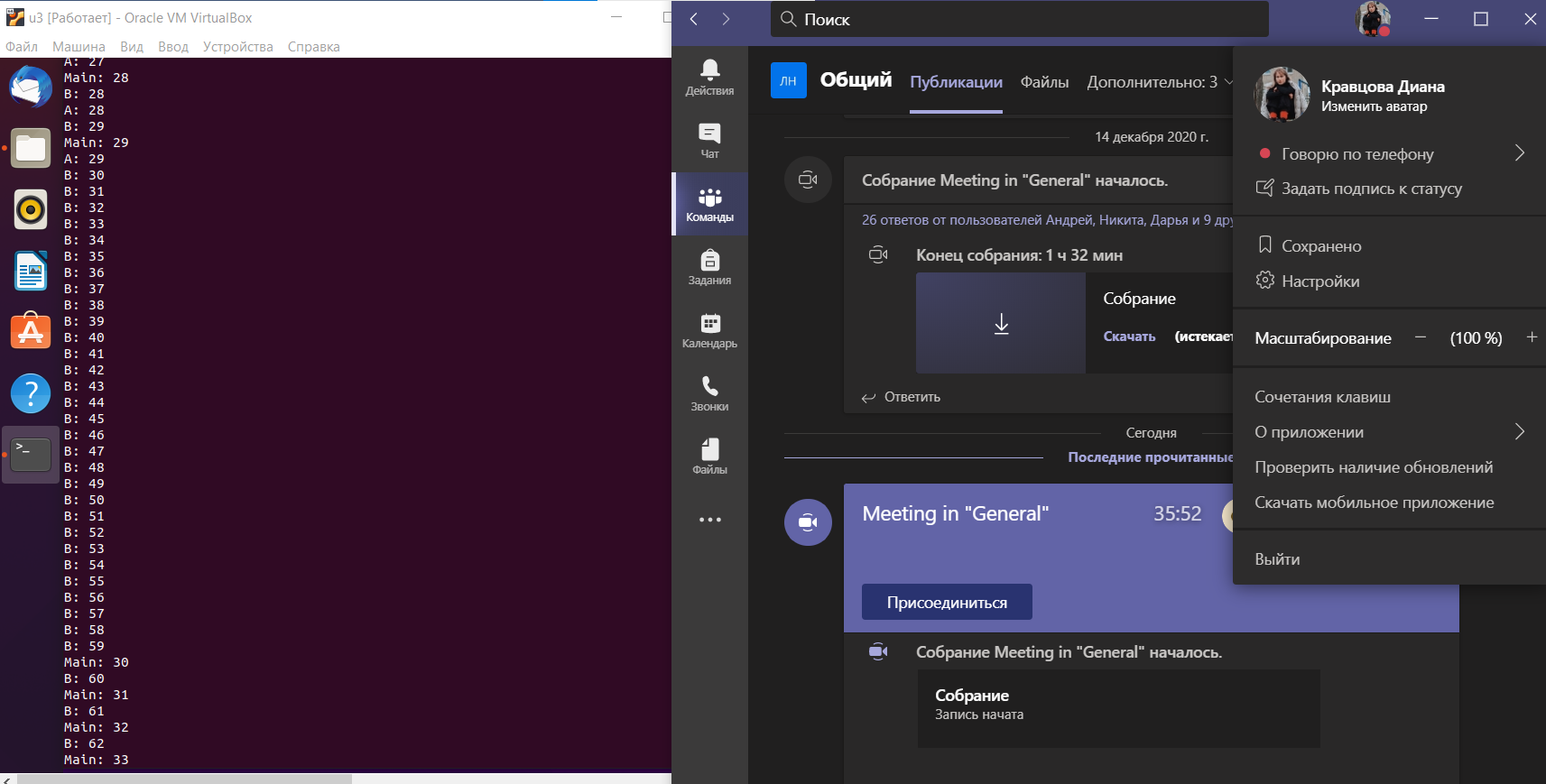


**Задание 6.**

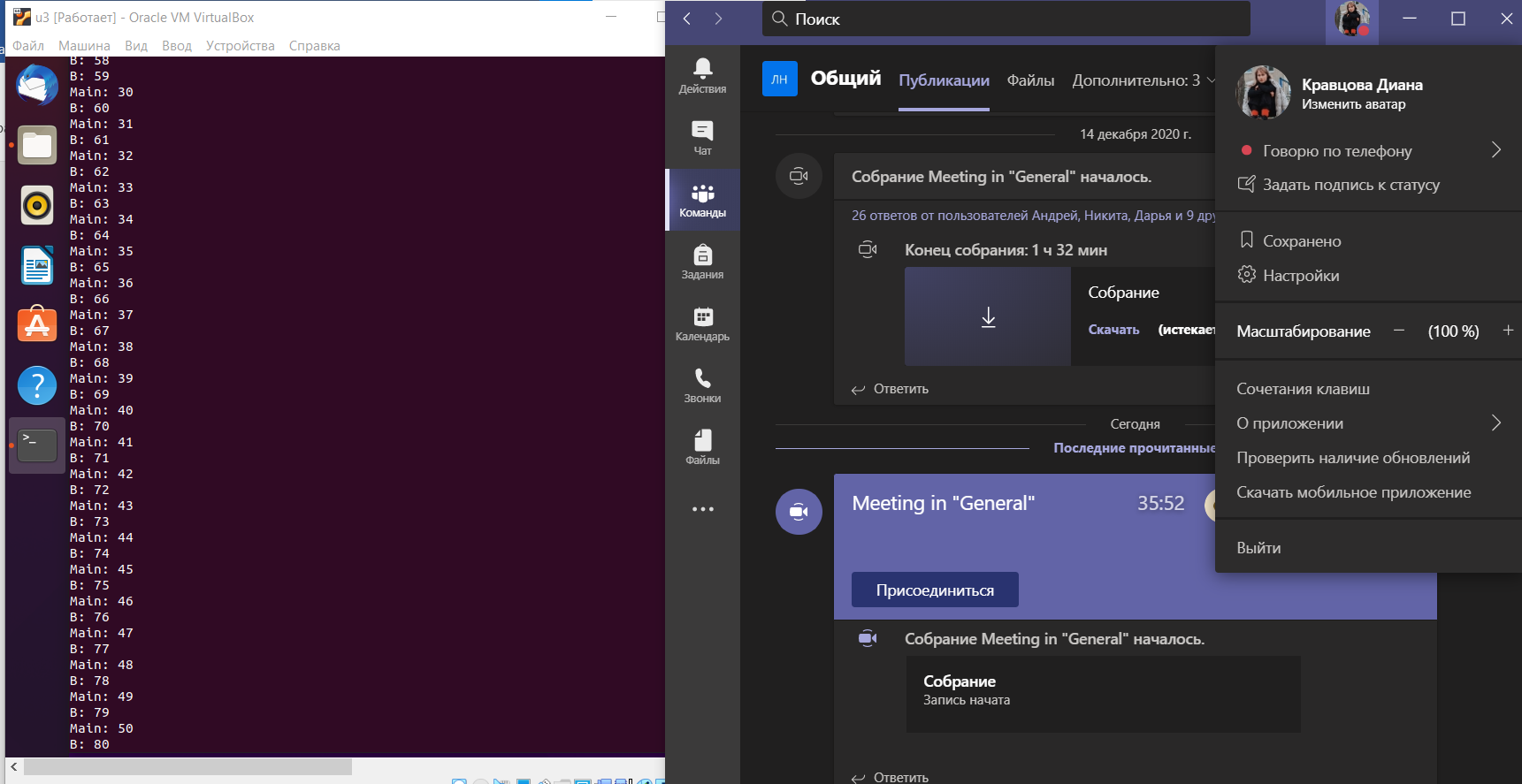
**Скриншот(ы) запуска приложения и результатов до 30 итерации.**

****

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

****

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

****

**pthread\_mutex\_init – функция, предназначеная для инициализации мьютекса. где первый аргумент – указатель на мьютекс, а второй – аттрибуты мьютекса.**

**После создания мьютекса он может быть захвачен с помощью функции pthread\_mutex\_lock**

**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**

**pthread\_mutex\_unlock**

**После использования мьютекса его необходимо уничтожить с помощью функции**

**pthread\_mutex\_destroy**