**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**ЛАБОРАТОРНАЯ РАБОТА №4**

**по дисциплине «Операционные системы»**

**Тема: “Межпроцессное взаимодействие”**

| Студент |  |  |
| --- | --- | --- |
| Преподаватель |  | Тимофеев А. В. |

Санкт-Петербург

2024

**Цель работы:** исследовать инструменты и механизмы взаимодействия процессов в Windows.

**Задание 4.1.** Реализация решения задачи о читателях-писателях.

**Указания к выполнению**:

1. Выполнить решение задачи о читателях-писателях, для чего необходимо разработать консольные приложения «Читатель» и «Писатель»:

* Одновременно запущенные экземпляры процессов-читателей и процессов-писателей должны совместно работать с буферной памятью в виде проецируемого файла:

1. размер страницы буферной памяти равен размеру физической страницы оперативной памяти;
2. число страниц буферной памяти равно сумме цифр в номере студенческого билета без учета первой цифры.

* Страницы буферной памяти должны быть заблокированы в оперативной памяти (функция **VirtualLock**);
* Длительность выполнения процессами операций «чтения» и «записи» задается случайным образом в диапазоне от 0,5 до 1,5 сек.;
* Для синхронизации работы процессов необходимо использовать объекты синхронизации типа «семафор» и «мьютекс»;
* Процессы-читатели и процессы-писатели ведут свои журнальные файлы, в которые регистрируют переходы из одного «состояния» в  другое (начало ожидания, запись или чтение, переход к освобождению) с указанием кода времени (функция **TimeGetTime**).

Для состояний «запись» и «чтение» необходимо также запротоколировать номер рабочей страницы.

2. Запустите приложения читателей и писателей, суммарное количество одновременно работающих читателей и писателей должно быть не менее числа страниц буферной памяти. Проверьте функционирование приложений, проанализируйте журнальные файлы процессов, постройте сводные графики смены «состояний» для не менее  5 процессов-читателей и 5 процессов-писателей, дайте свои комментарии относительно переходов процессов из одного состояния в другое. Постройте графики занятости страниц буферной памяти (проецируемого файла) во времени, дайте свои комментарии.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

**Реализация программы**

Нужно реализовать решение задачи о писателях и читателях. Для этого необходимо организовать правильную работу приложений «писателя» и «читателя» с общей областью памяти, построить графики смены состояний писателей и читателей, а также графики занятости страниц памяти.

Принцип работы и взаимодействия писателей и читателей следующий: создаются два массива семафоров (размер = количеству страниц). Эти семафоры отслеживают в какие можно записывать, и какие страницы следует прочитать. Семафоры, отслеживающие для записи, сразу доступны для использования, т.е. счетчик равен 1, а для чтения равен 0. Когда в страницу что-то записывается, процесс повышает счетчик семафора для чтения на 1, аналогично для чтения.

Также для каждого процесса создан свой файл. В этот файл записывается время начала ожидания процесса свободной страницы, время взятия свободной страницы и время освобождения страницы.

Номер студ. билета – 130725 => **Количество страниц = 3 + 7 + 2 + 5 = 17.**

**Проецирование файла:**

file= CreateFile("filename", GENERIC\_WRITE | GENERIC\_READ, 0, nullptr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0),  
mapFile=CreateFileMapping(file, nullptr, PAGE\_READWRITE, 0, pageSize \* pageCount, "filemap");  
LPVOID viewFile= MapViewOfFile(mapFile,FILE\_MAP\_ALL\_ACCESS,0,0,pageCount\*pageSize);

**Создание семафоров:**

for (int i=0;i<pageCount;i++){  
 wSemaphore[i]= CreateSemaphoreA(nullptr,1,1,("wSemaphore"+std::to\_string(i)).c\_str());  
 rSemaphore[i]=CreateSemaphoreA(nullptr,0,1,("rSemaphore"+std::to\_string(i)).c\_str());  
}

**Создание процессов «писателя»:**

for (int i=0;i<processCount;i++){  
 STARTUPINFO si;  
 PROCESS\_INFORMATION pi;  
 SECURITY\_ATTRIBUTES secureAttr = { sizeof(secureAttr), nullptr, TRUE };  
 ZeroMemory( &si, sizeof(si) );  
 ZeroMemory( &pi, sizeof(pi) );  
 si.cb = sizeof(si);  
 si.cb = sizeof(si);  
 si.dwFlags |= STARTF\_USESTDHANDLES;  
 si.hStdInput = nullptr;  
 si.hStdError = nullptr;  
 si.hStdOutput=CreateFile(("writeLog" + std::to\_string(i) + ".txt").c\_str(), GENERIC\_WRITE | GENERIC\_READ, 0, &secureAttr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0);;  
 int process= CreateProcess((LPCTSTR)"writer.exe", // Process handle not inheritable  
 nullptr,  
 nullptr, // Process handle not inheritable  
 nullptr, // Thread handle not inheritable  
 true, // Set handle inheritance to FALSE  
 0, // No creation flags  
 nullptr, // Use parent's environment block  
 nullptr, // Use parent's starting directory  
 &si, // Pointer to STARTUPINFO structure  
 &pi);  
 if (!process) processHandles[i]=pi.hProcess;  
}

**Создание процессов «читателя»:**

for (int i=0;i<processCount;i++){  
 STARTUPINFO si;  
 PROCESS\_INFORMATION pi;  
 SECURITY\_ATTRIBUTES secureAttr = { sizeof(secureAttr), nullptr, TRUE };  
 ZeroMemory( &si, sizeof(si) );  
 ZeroMemory( &pi, sizeof(pi) );  
 si.cb = sizeof(si);  
 si.dwFlags |= STARTF\_USESTDHANDLES;  
 si.hStdInput = nullptr;  
 si.hStdError = nullptr;  
 si.hStdOutput=CreateFile(("readLog" + std::to\_string(i) + ".txt").c\_str(), GENERIC\_WRITE | GENERIC\_READ, 0, &secureAttr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0);  
 int process= CreateProcess((LPCTSTR)"reader.exe",   
 nullptr,  
 nullptr,   
 nullptr,   
 true,   
 0,   
 nullptr,   
 nullptr,   
 &si, &pi);  
 if (!process) processHandles[processCount+i]=pi.hProcess;  
}

**Реализация процесса «писателя»:**

*Открываем проецируемый файл:*

mappingHandle = OpenFileMapping(FILE\_MAP\_READ | FILE\_MAP\_WRITE, FALSE, (LPCTSTR)"filemap");

*Сам процесс «писателя»:*

std::string str ="Process with id " + ProccessId + " waiting :" + std::to\_string(GetTickCount()) + "\n";  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
  
int sem = WaitForMultipleObjects(pageCount, wSemaphore, false, INFINITE);  
str = "Process with id " + ProccessId + " take page:" + std::to\_string(sem) + ":" +  
 std::to\_string(GetTickCount()) + "\n";  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
  
  
Sleep( rand() % 1000+ 500);  
  
str = "Process with id " + ProccessId + " free page " + std::to\_string(sem) + ":" +  
 std::to\_string(GetTickCount()) + "\n\n";  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
  
ReleaseSemaphore(rSemaphore[sem], 1, nullptr);

Процесс ожидает семафор из массива wSemaphore, получает индекс страницы и в конце работы повышает счетчик для страницы с полученным индексом массива rSemaphore.

**Реализация процесса «читателя»:**

*Открываем проецируемый файл:*

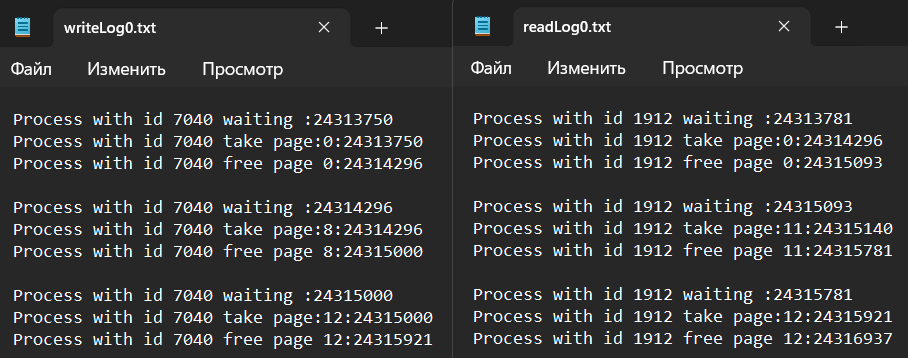
mappingHandle = OpenFileMapping(FILE\_MAP\_READ | FILE\_MAP\_WRITE, FALSE, (LPCTSTR)"filemap");

*Сам процесс «читателя»:*

std::string str ="Process with id " + ProccessId + " waiting :" + std::to\_string(GetTickCount()) + "\n";  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
  
int sem = WaitForMultipleObjects(pageCount, rSemaphore, false, INFINITE);  
str = "Process with id " + ProccessId + " take page:" + std::to\_string(sem) + ":" +  
 std::to\_string(GetTickCount()) + "\n";  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
Sleep( rand() % 1000+500);  
  
str = "Process with id " + ProccessId + " free page " + std::to\_string(sem) + ":" +  
 std::to\_string(GetTickCount()) + "\n\n";  
  
WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);  
  
ReleaseSemaphore(wSemaphore[sem], 1, nullptr);

Процесс ожидает семафор из массива rSemaphore, получает индекс страницы и в конце работы повышает счетчик для страницы с полученным индексом массива wSemaphore.

**Примеры файлов работы:**

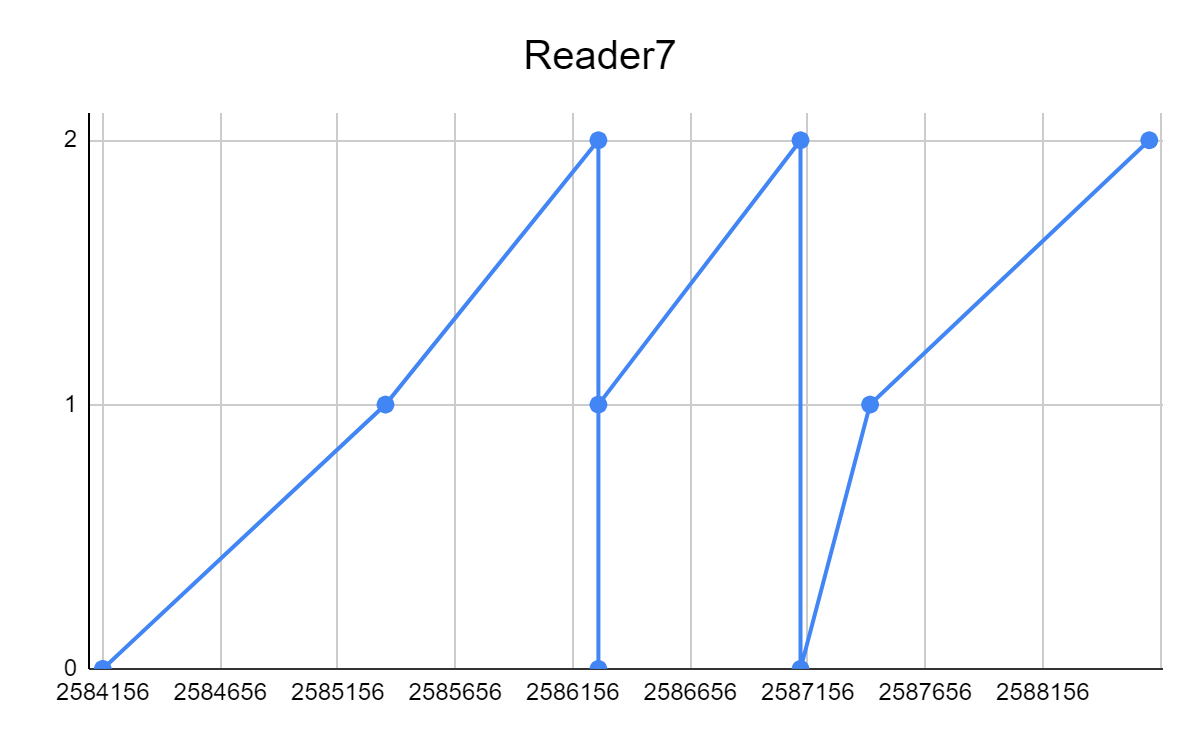
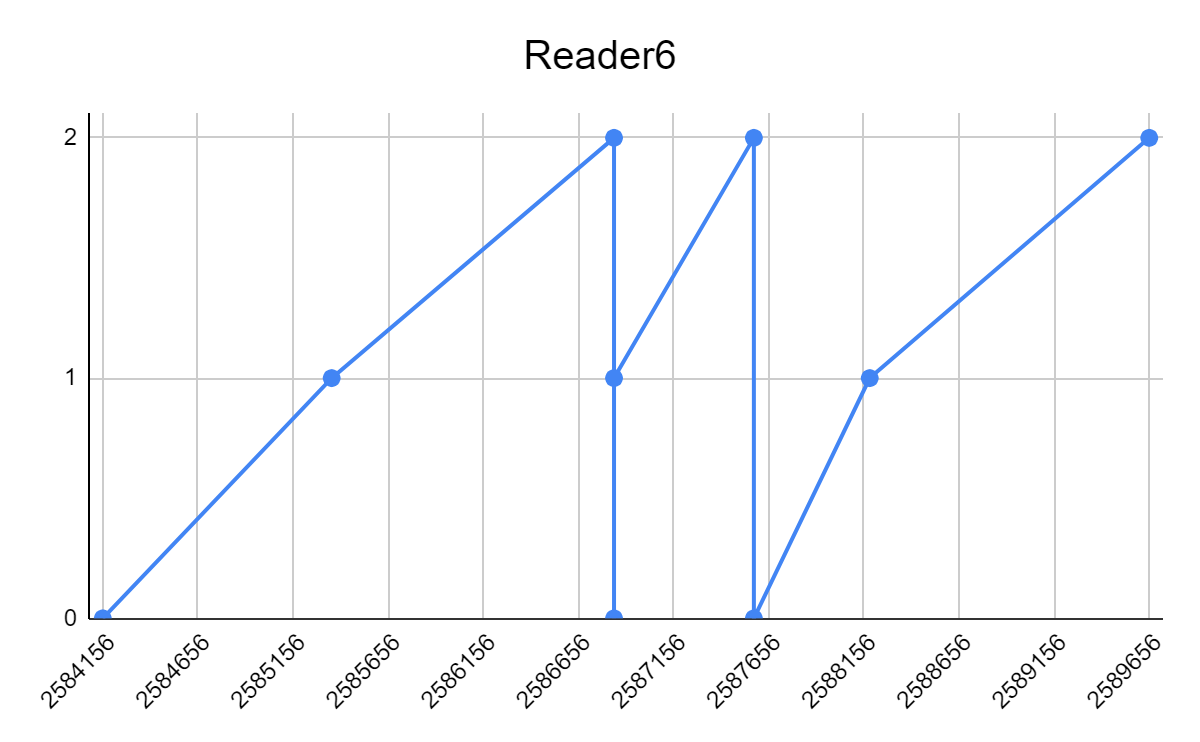
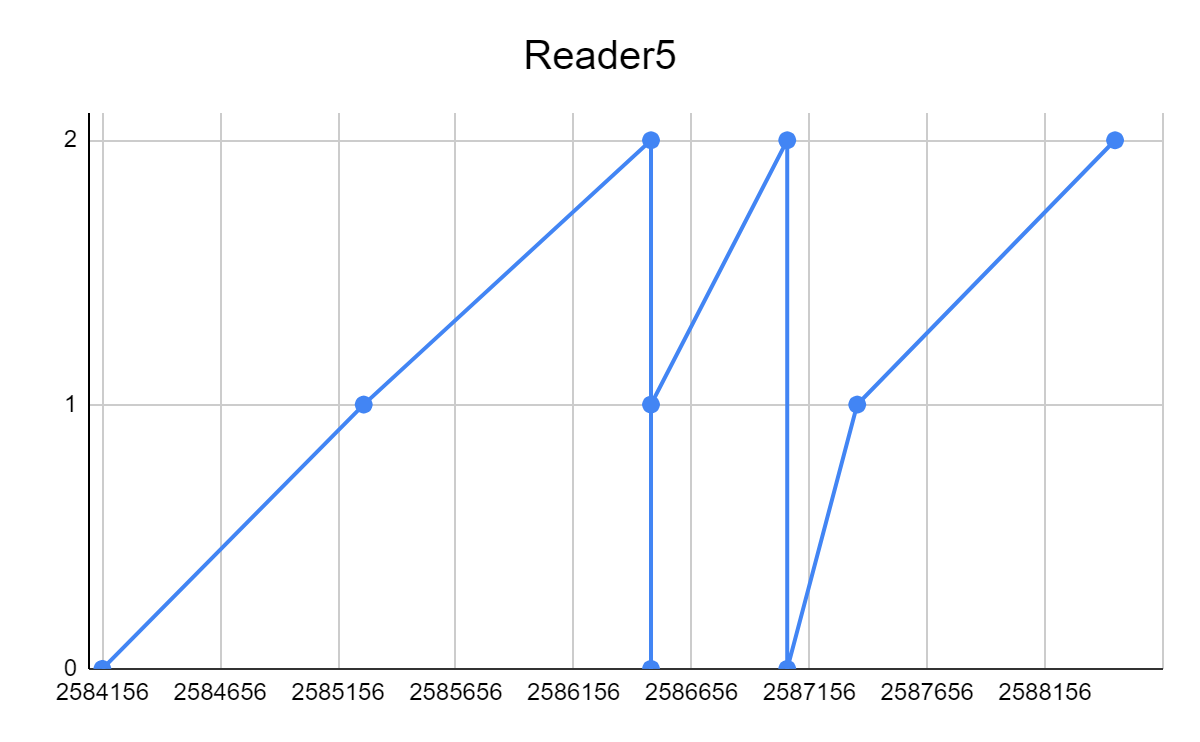
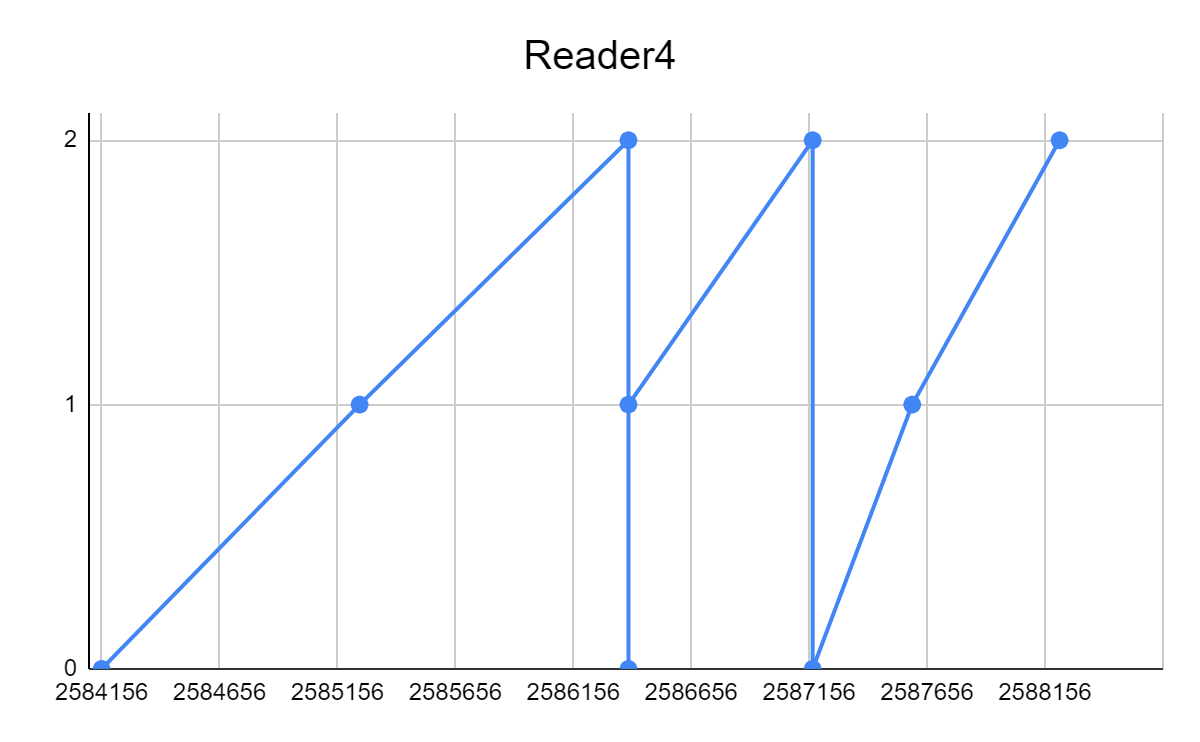
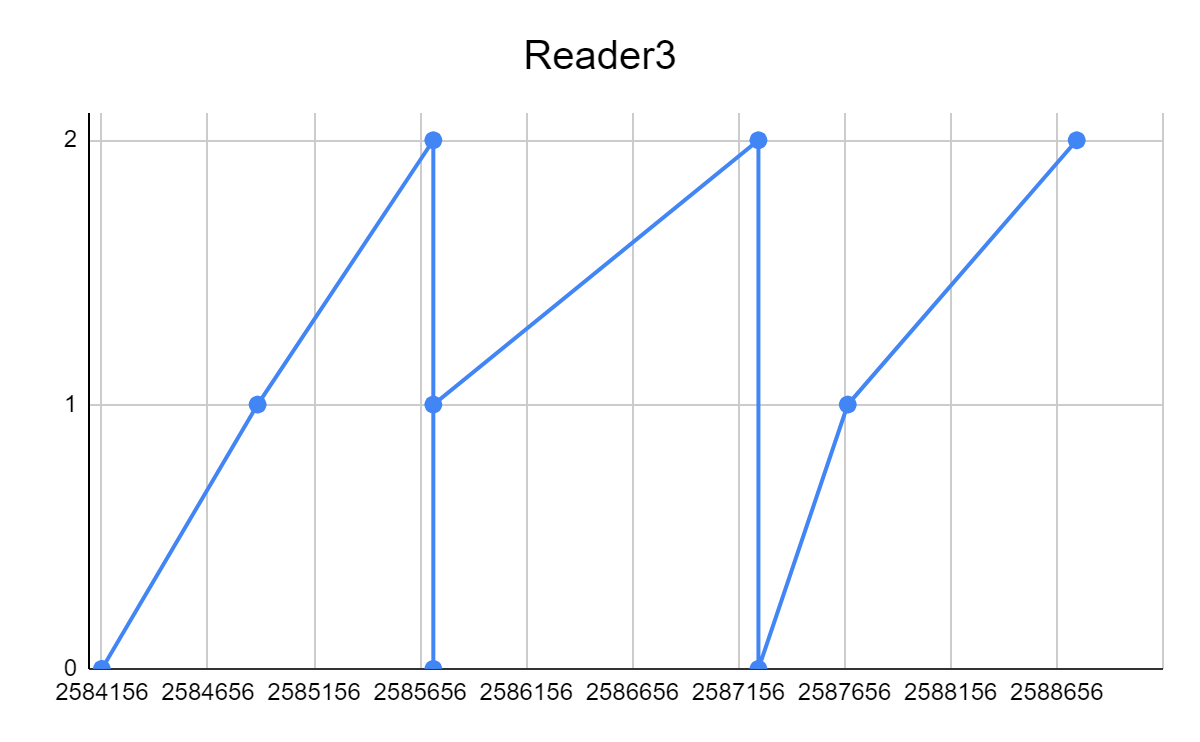
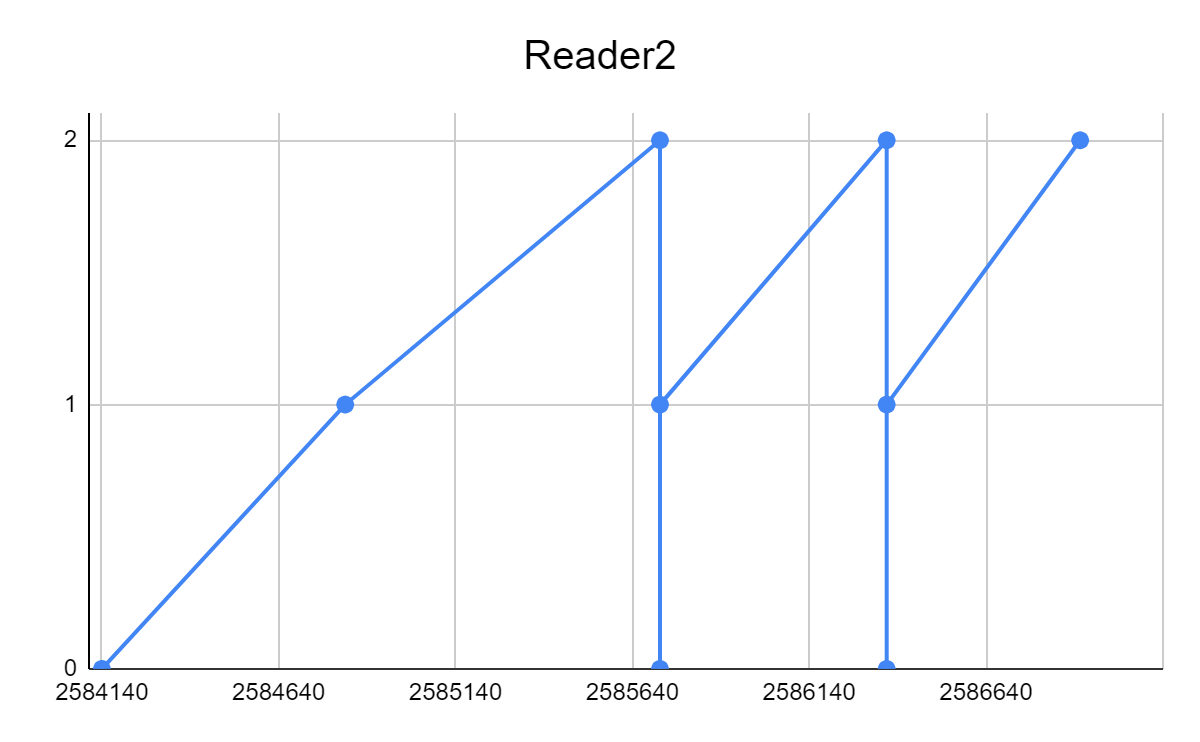
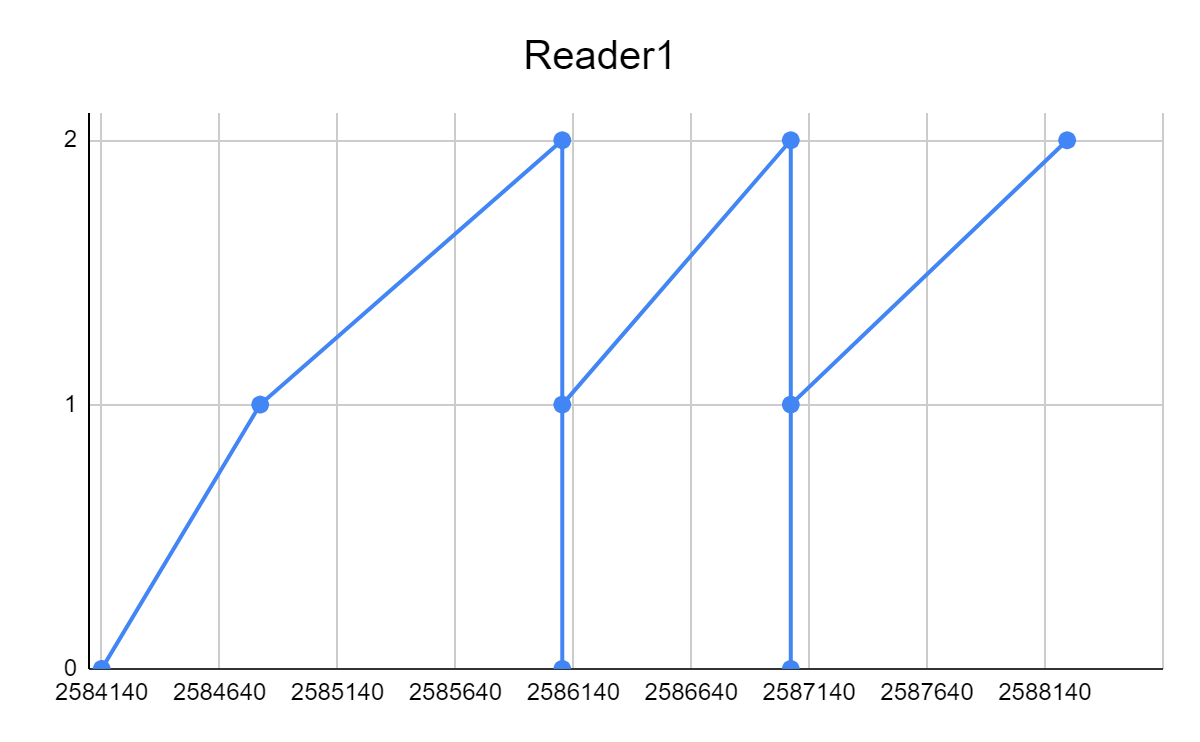
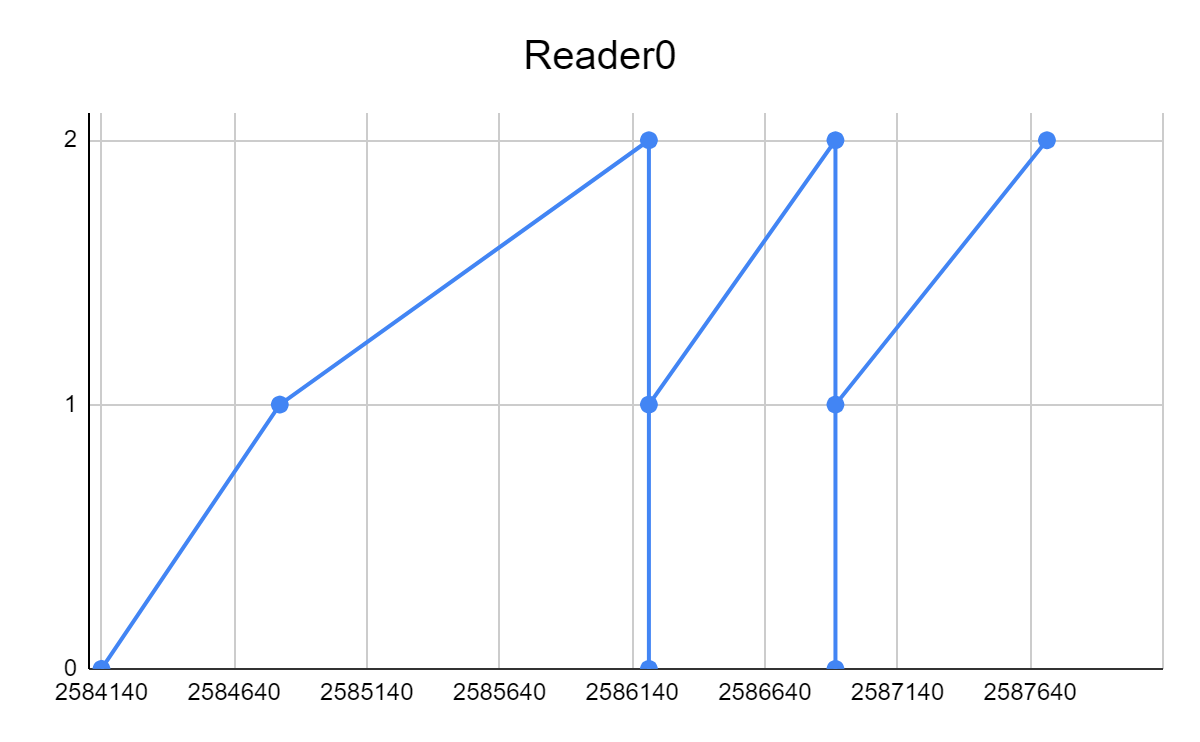
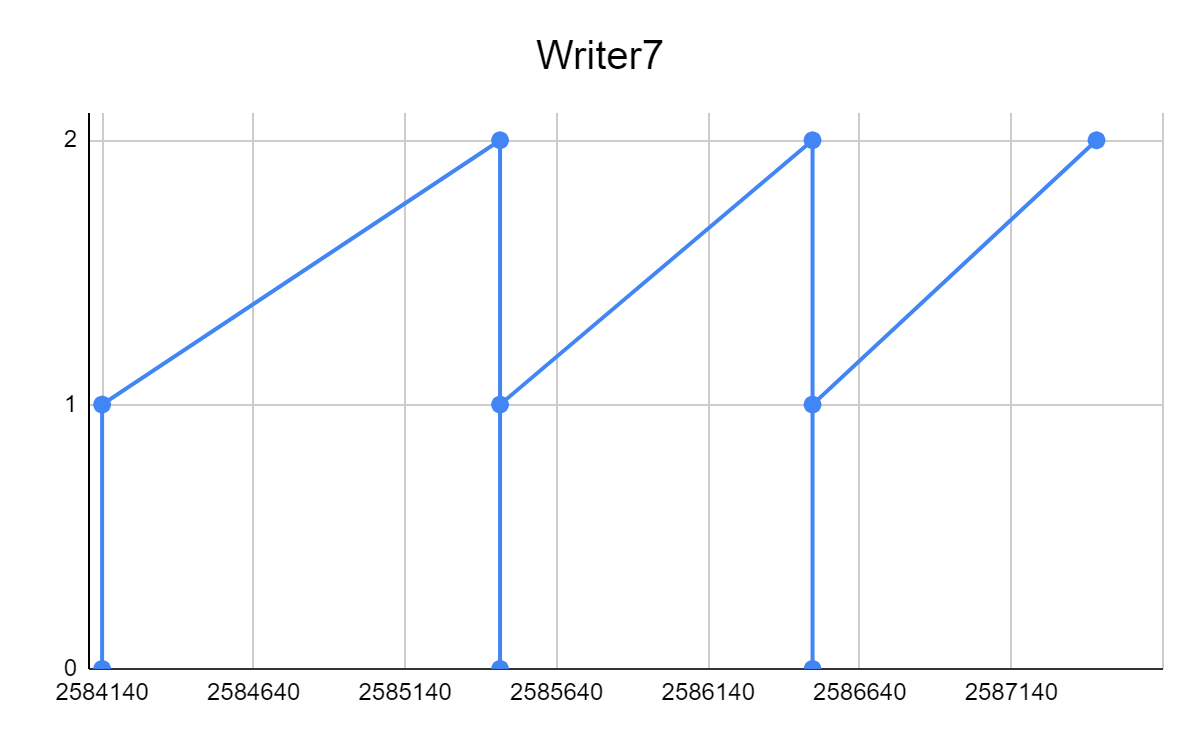
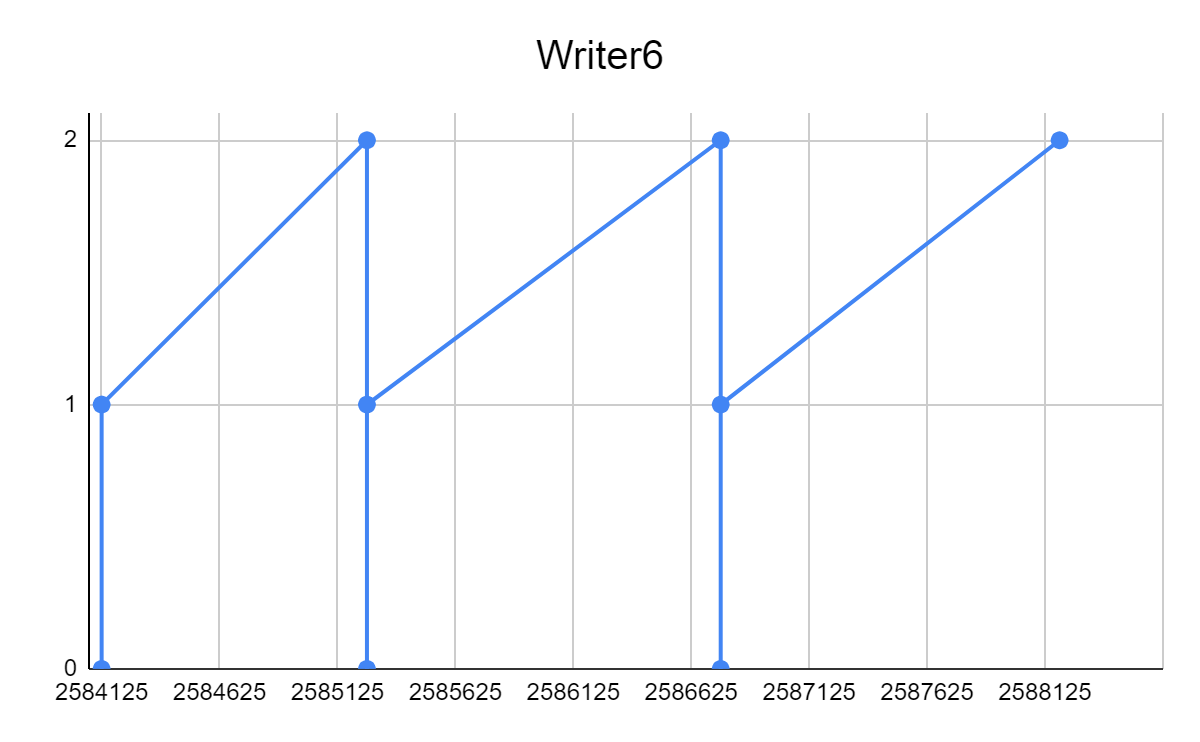
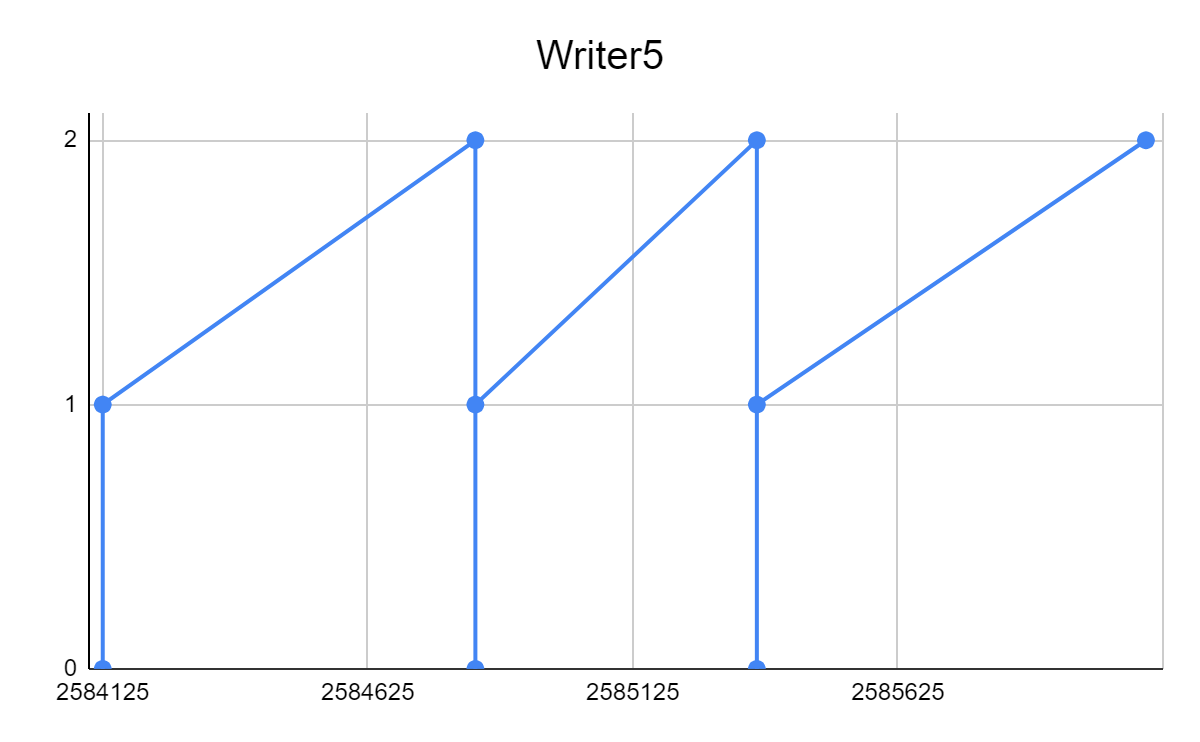
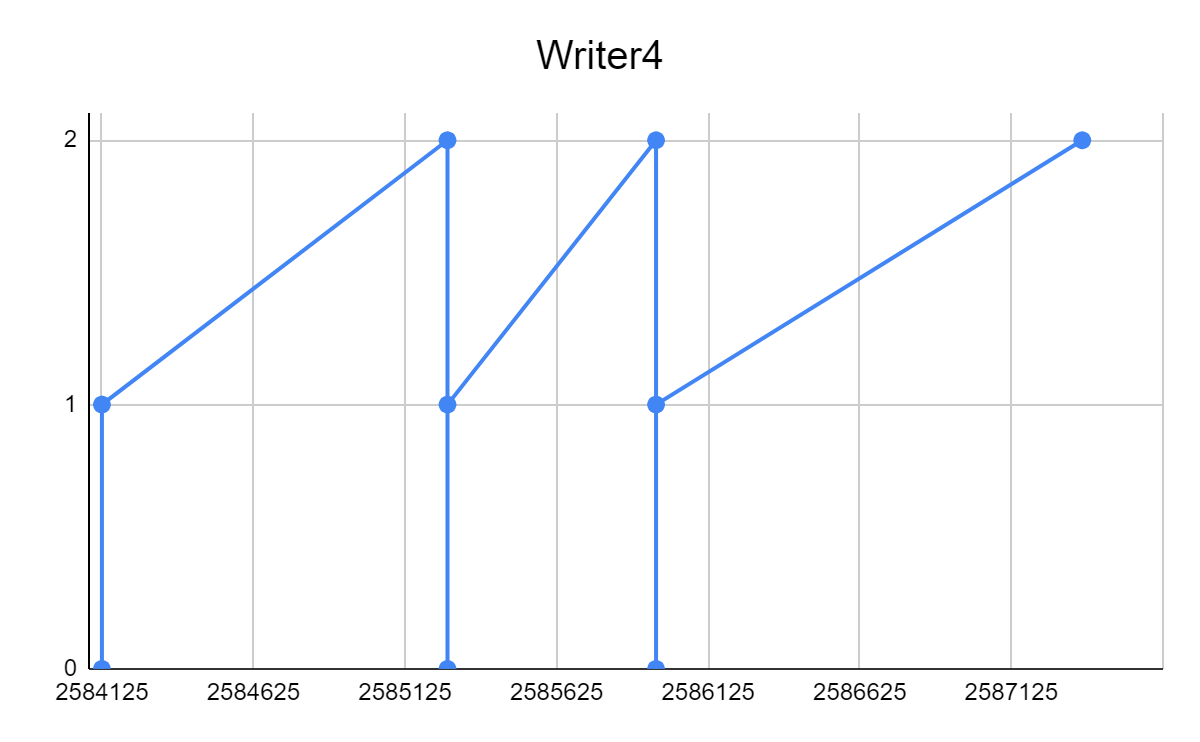
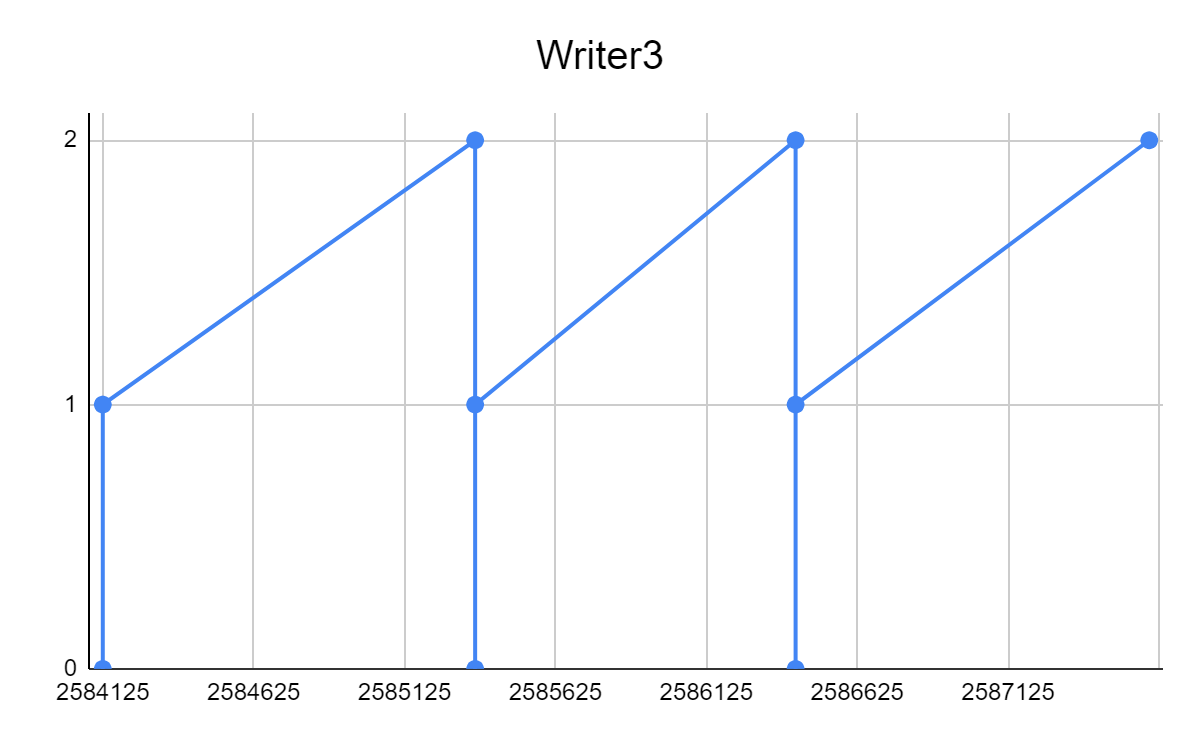
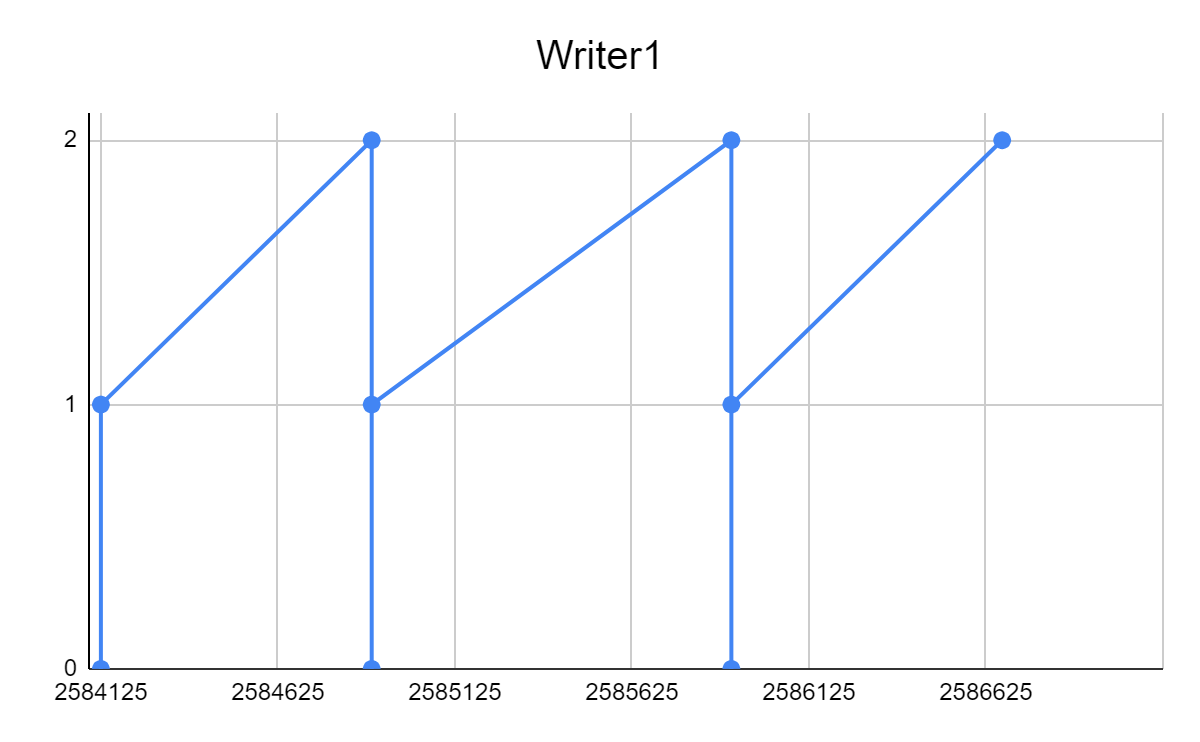
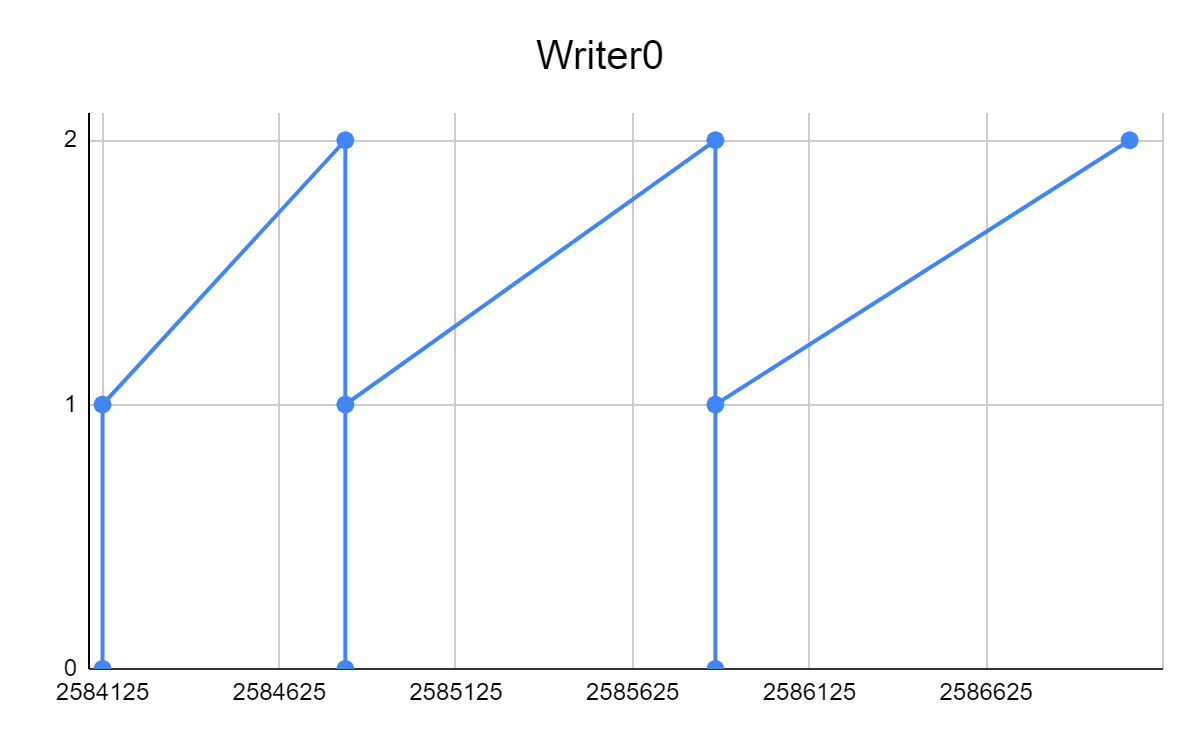


**Графики смены состояний процессов-«писателей» и -«читателей»:**

8 «писателей» и 8 «читателей».

Каждый «писатель» и «читатель» имеет 3 состояния:

* 0 – ожидание семафора на запись (или чтение)
* 1 – запись (или чтение)
* 2 – освобождение семафора

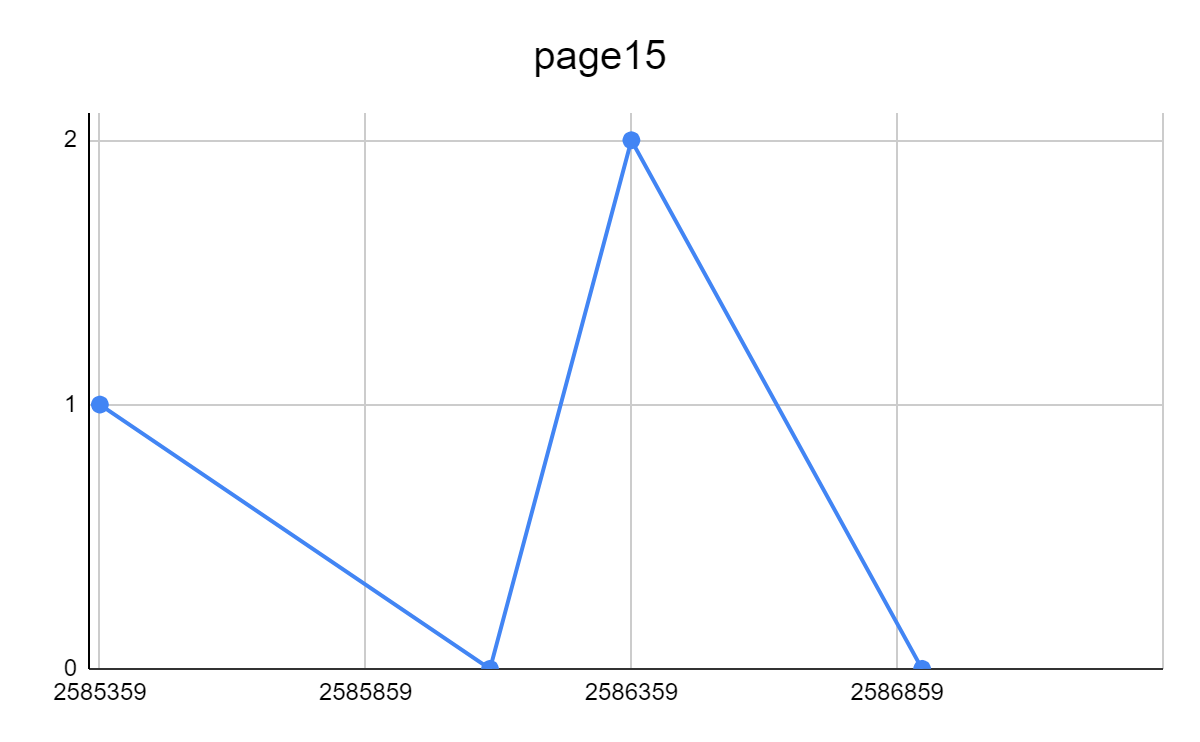
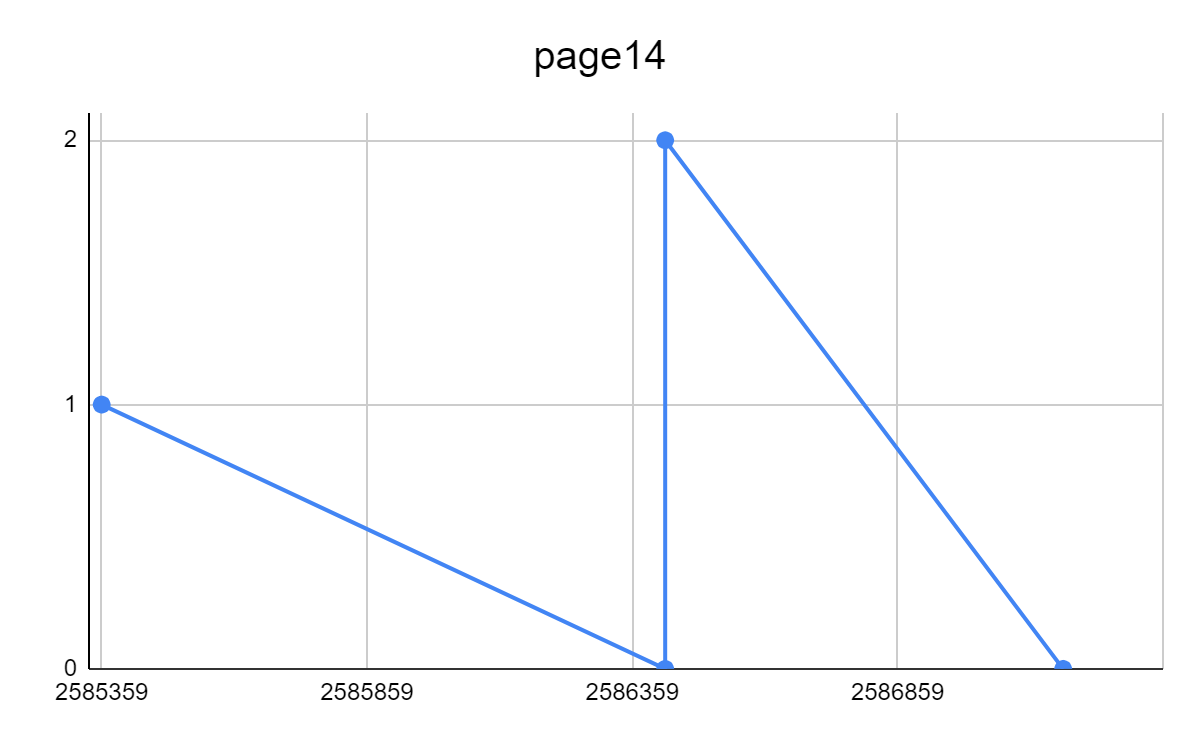
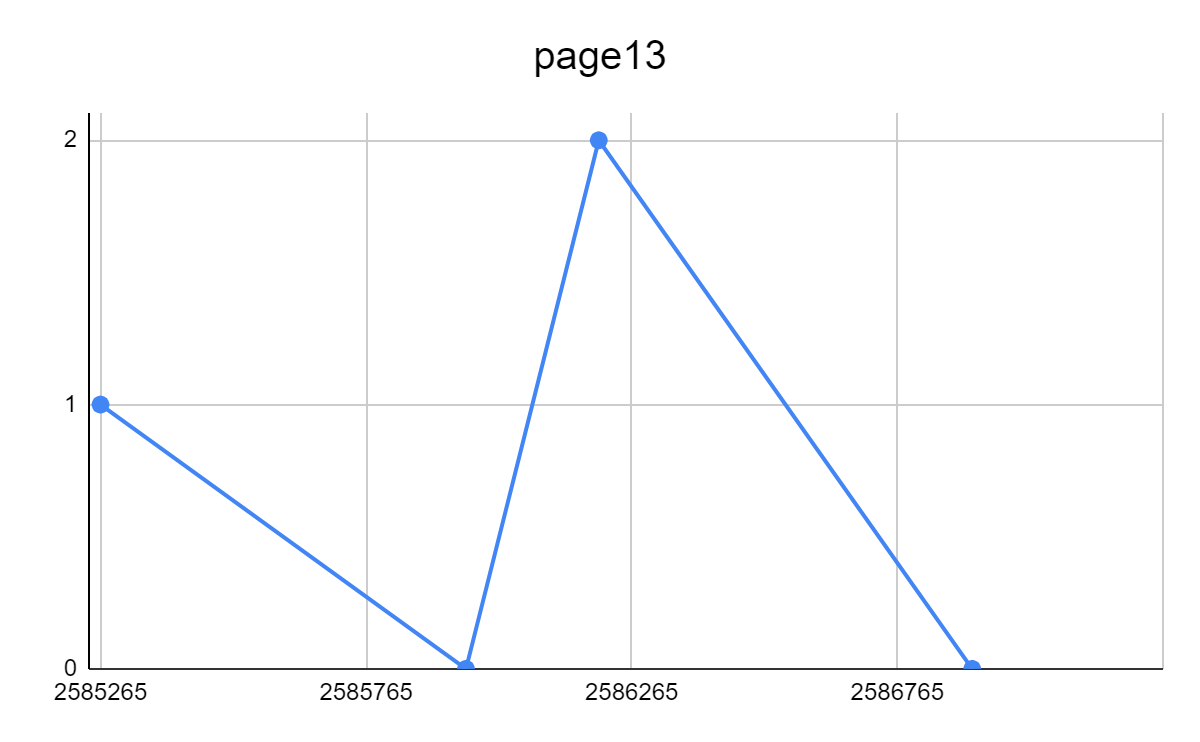
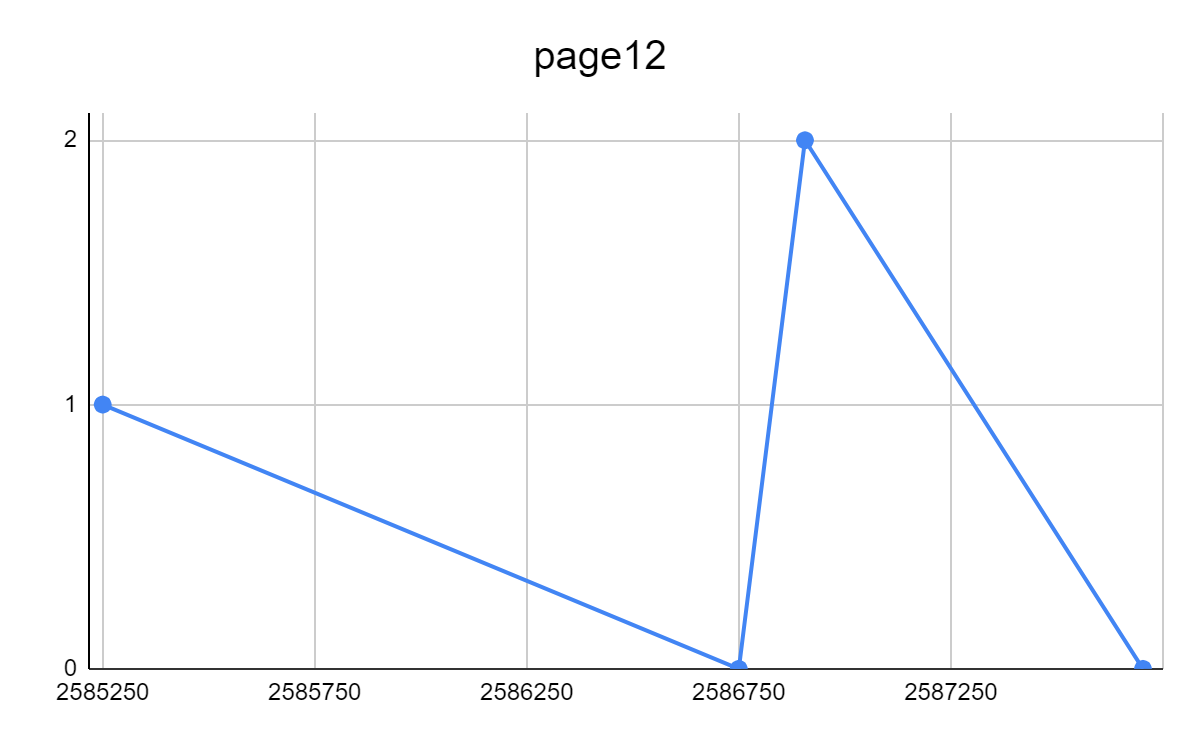
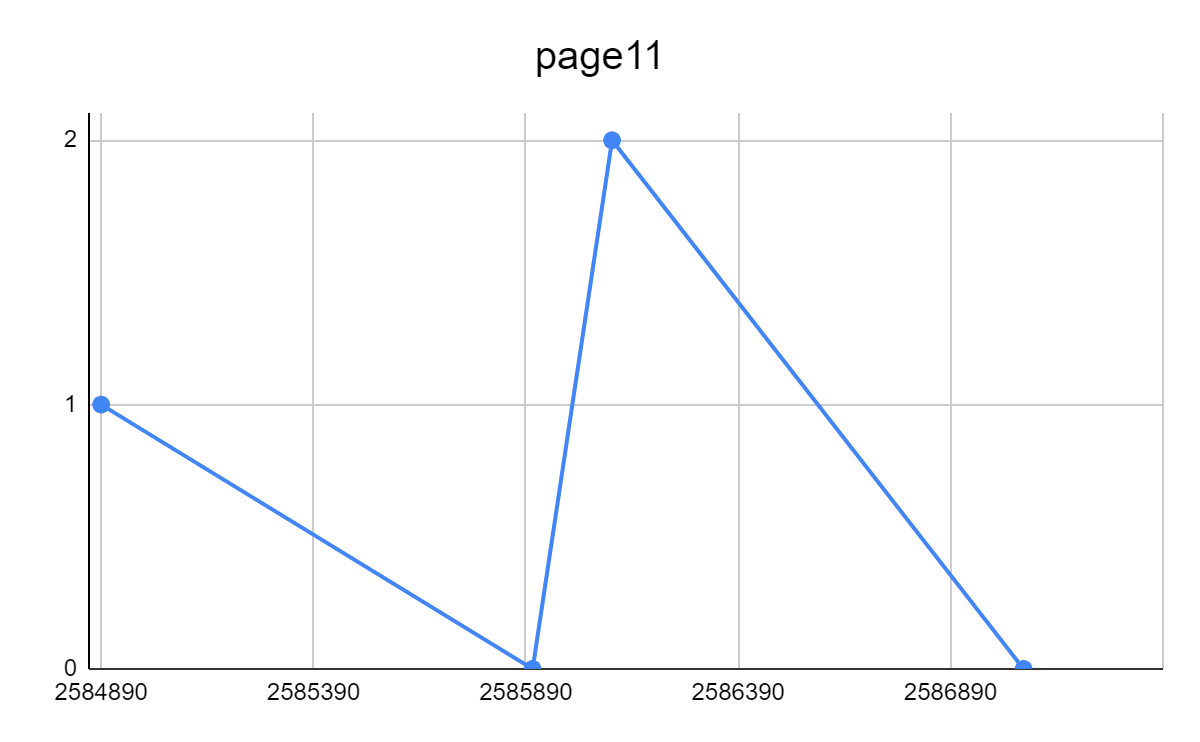
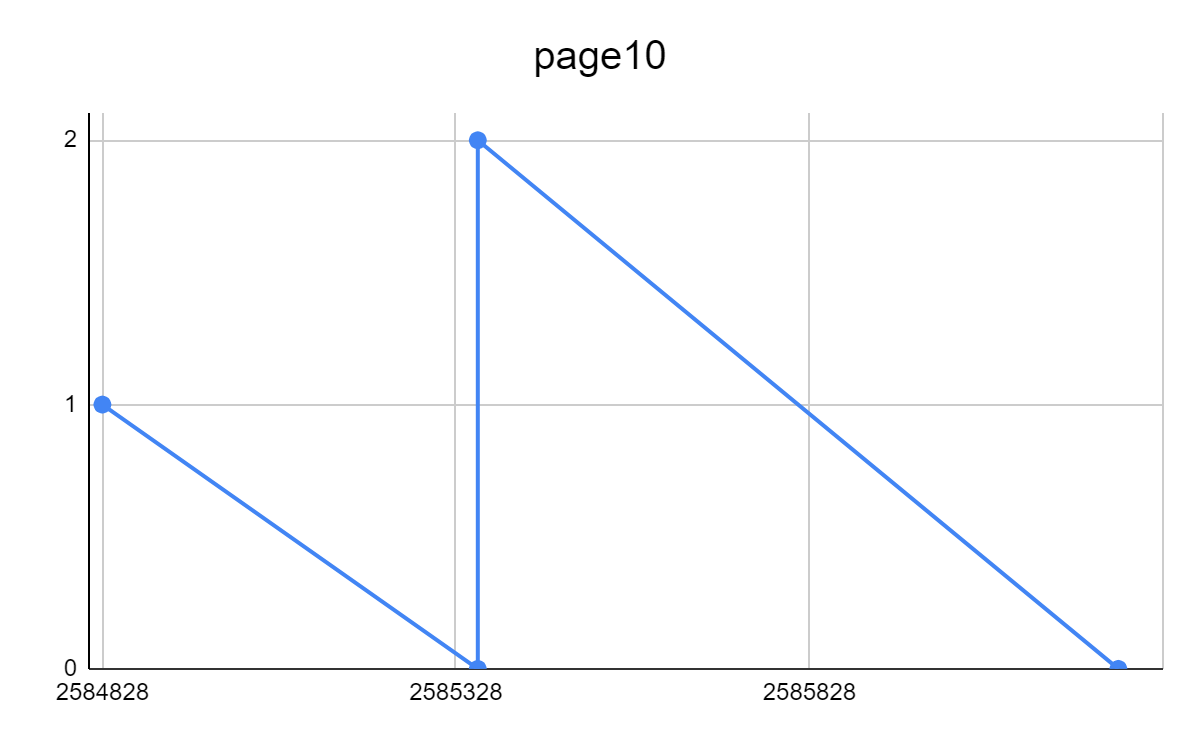
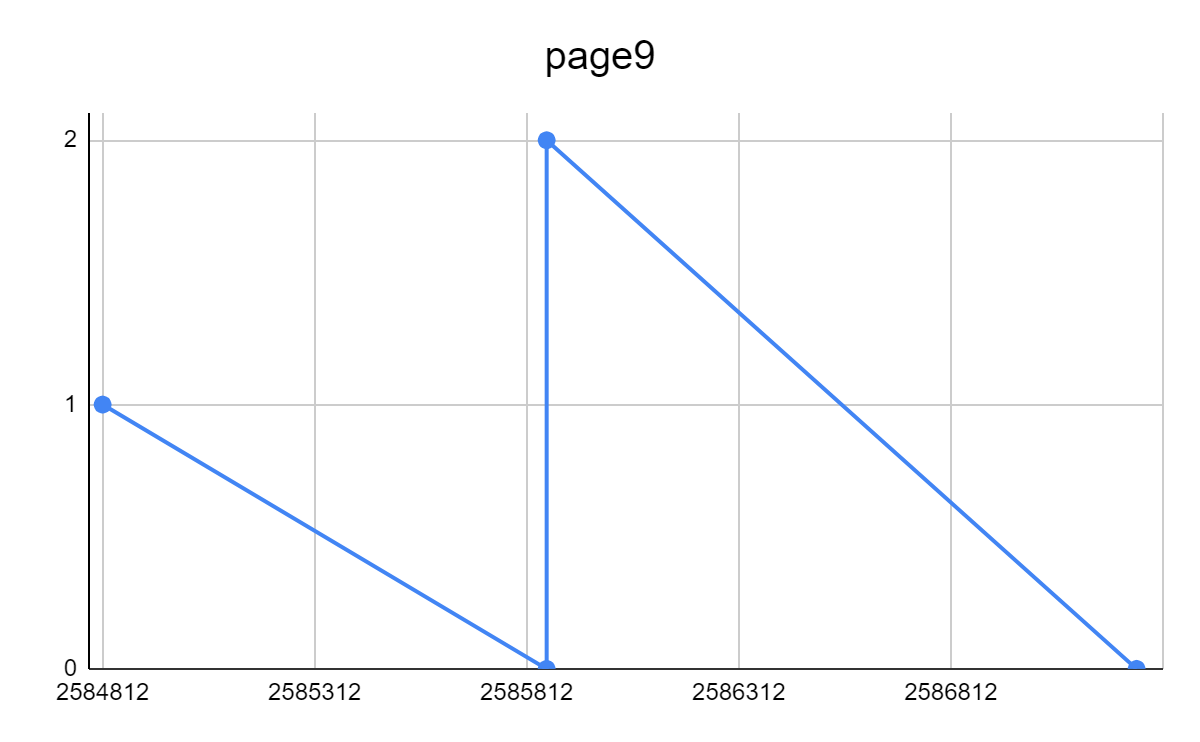
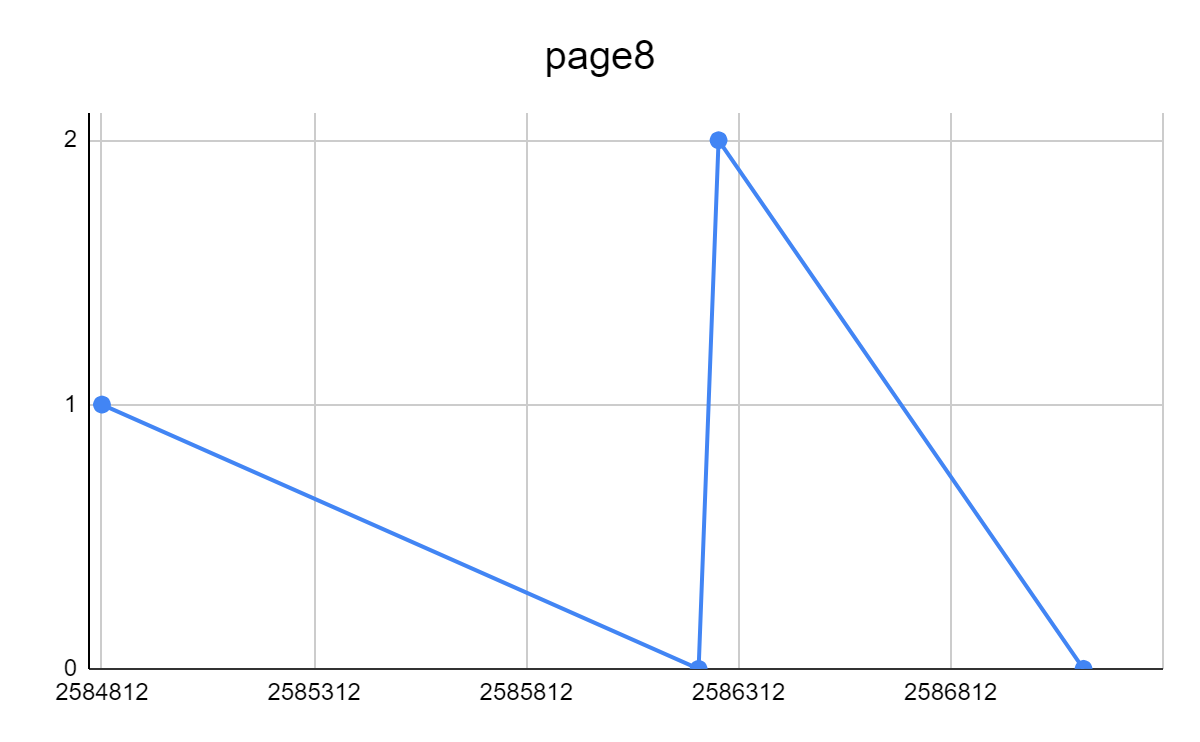
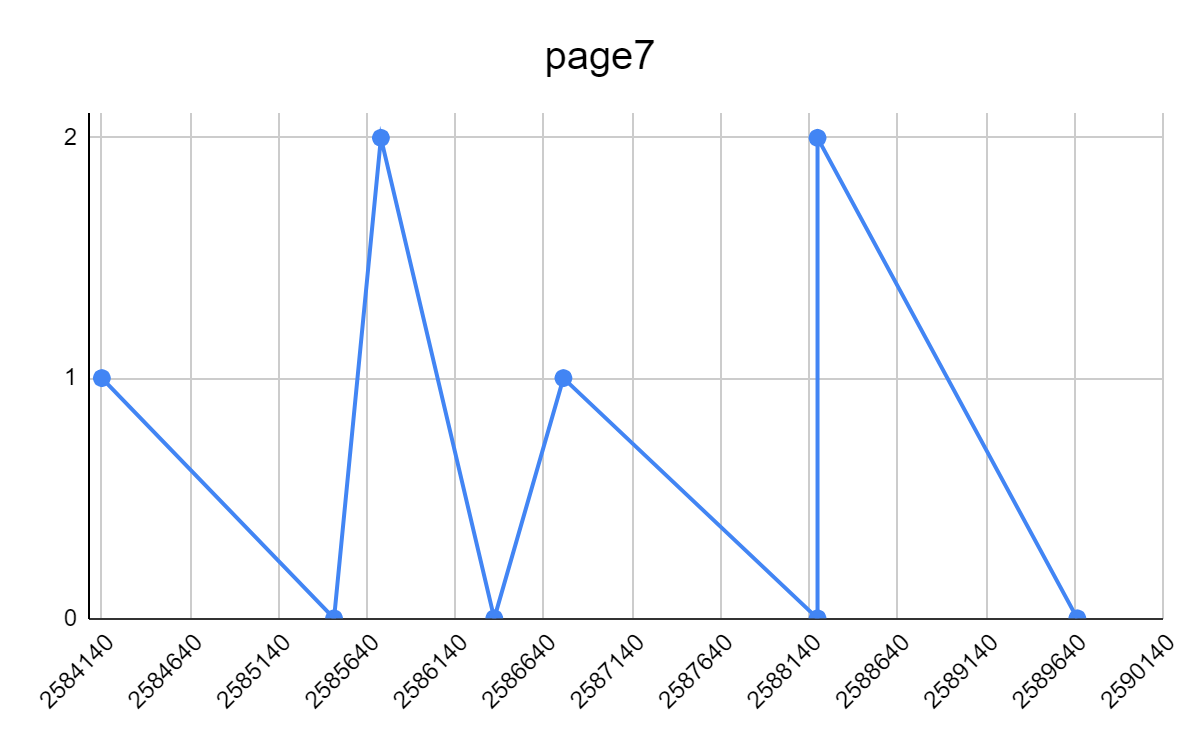
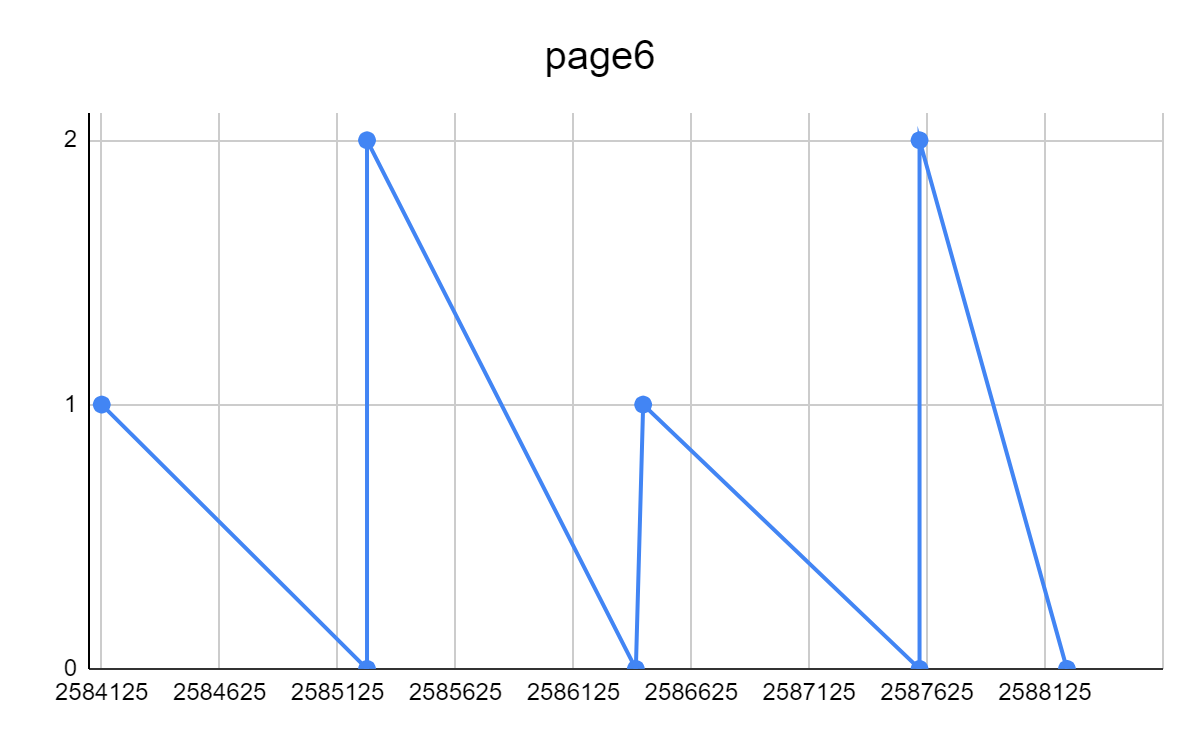
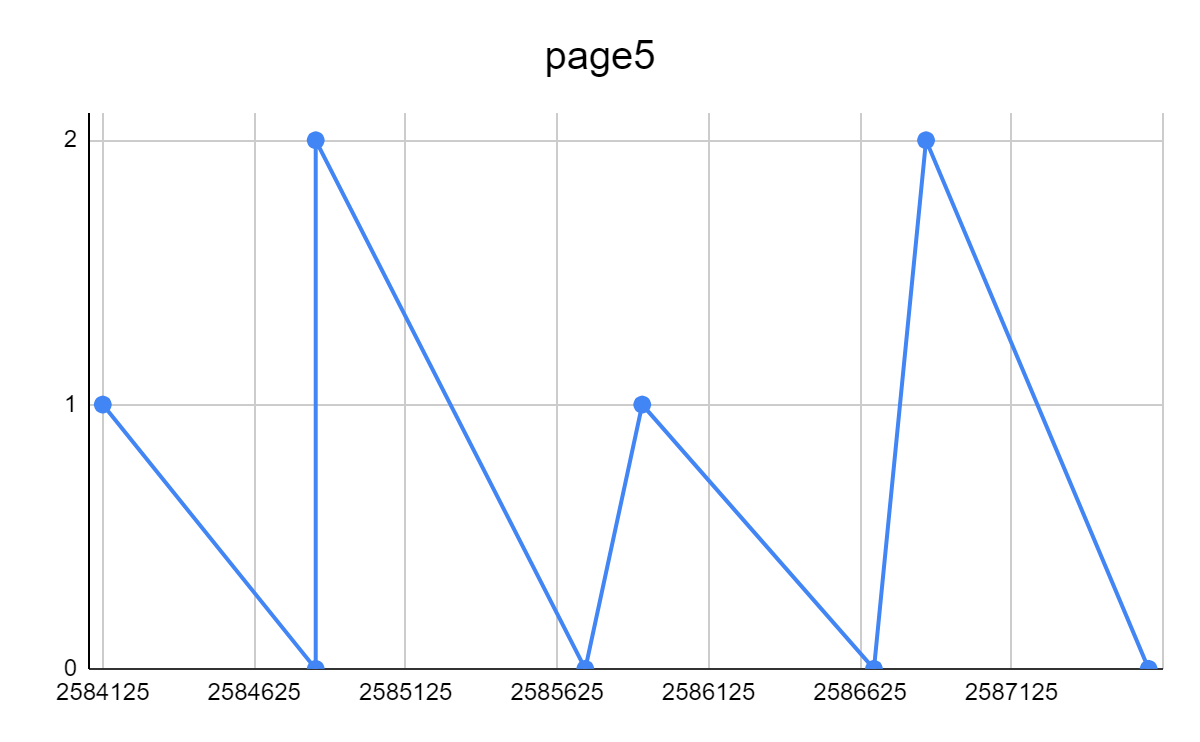
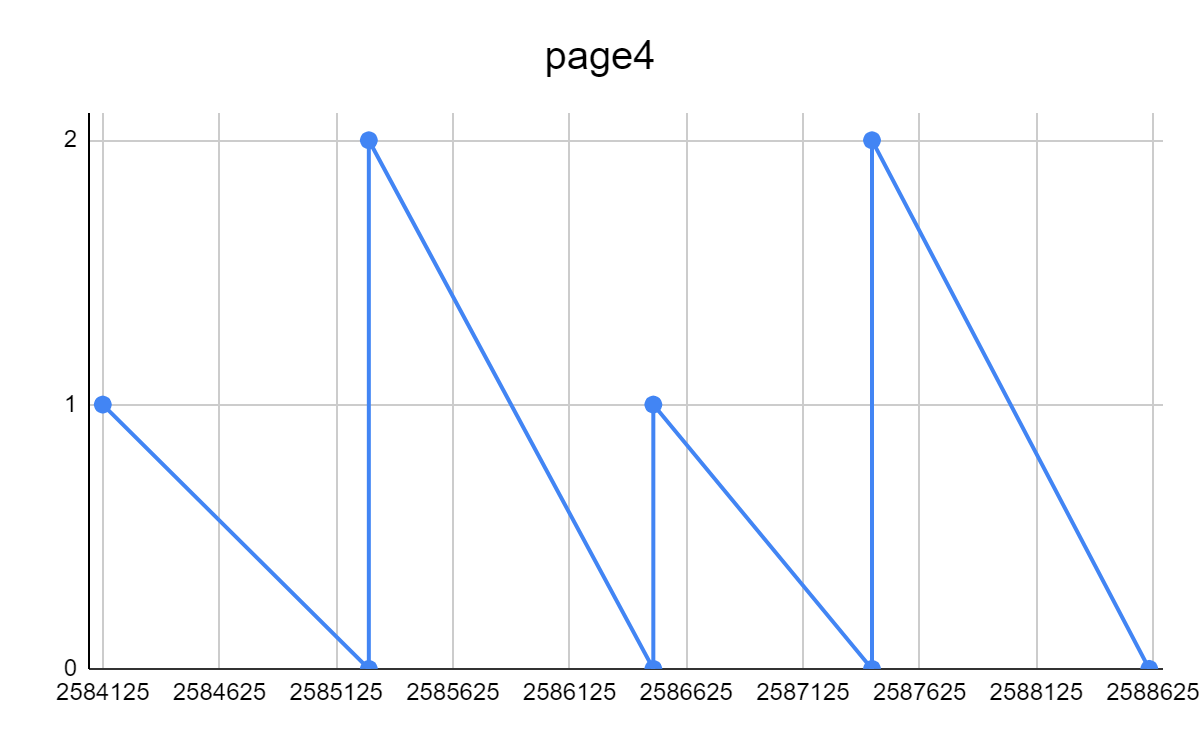
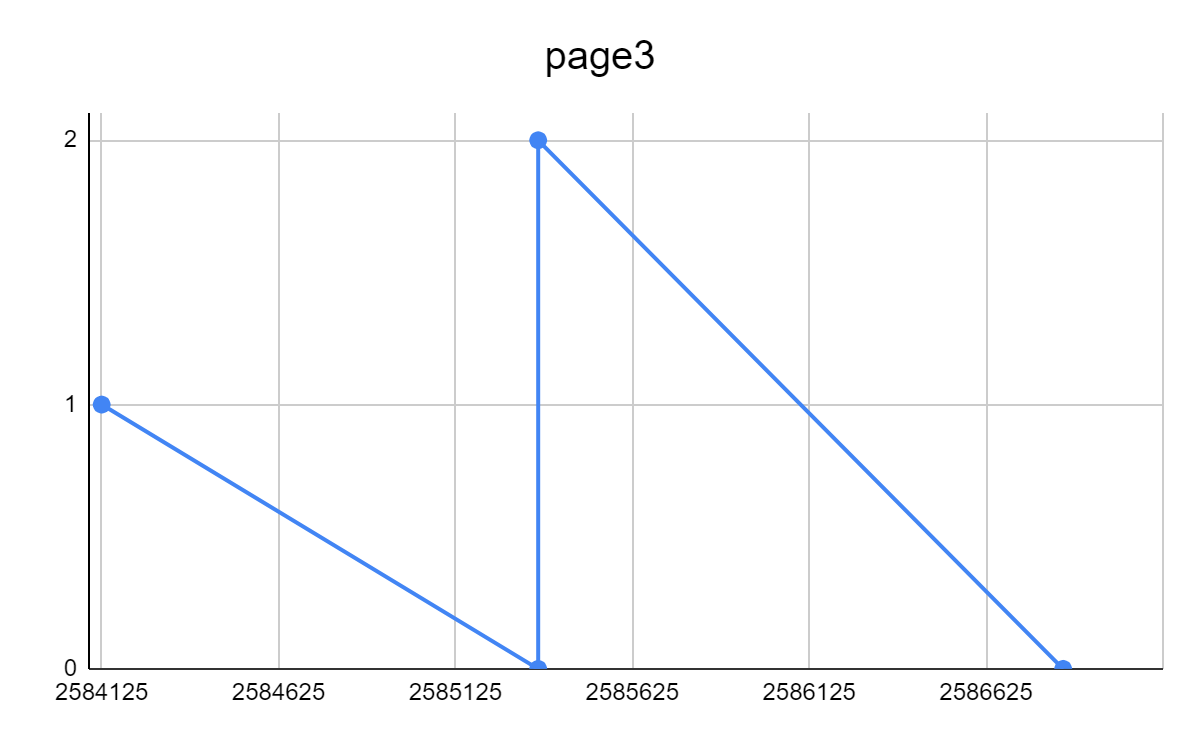
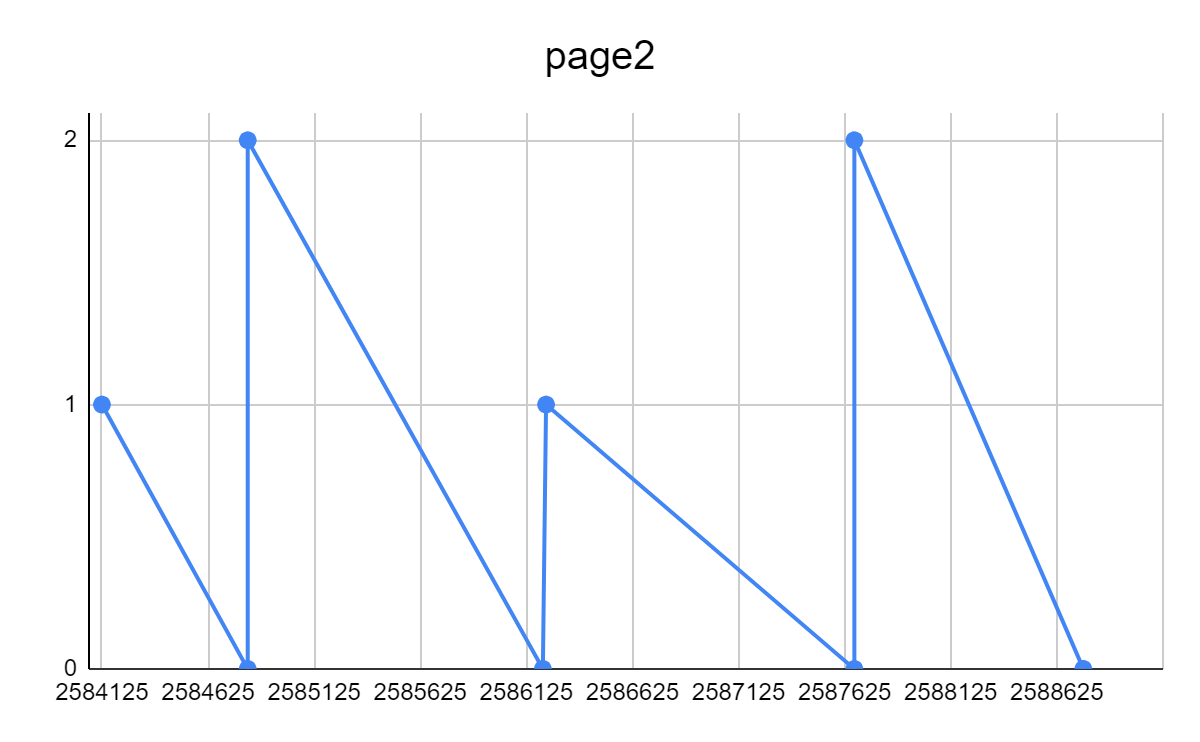
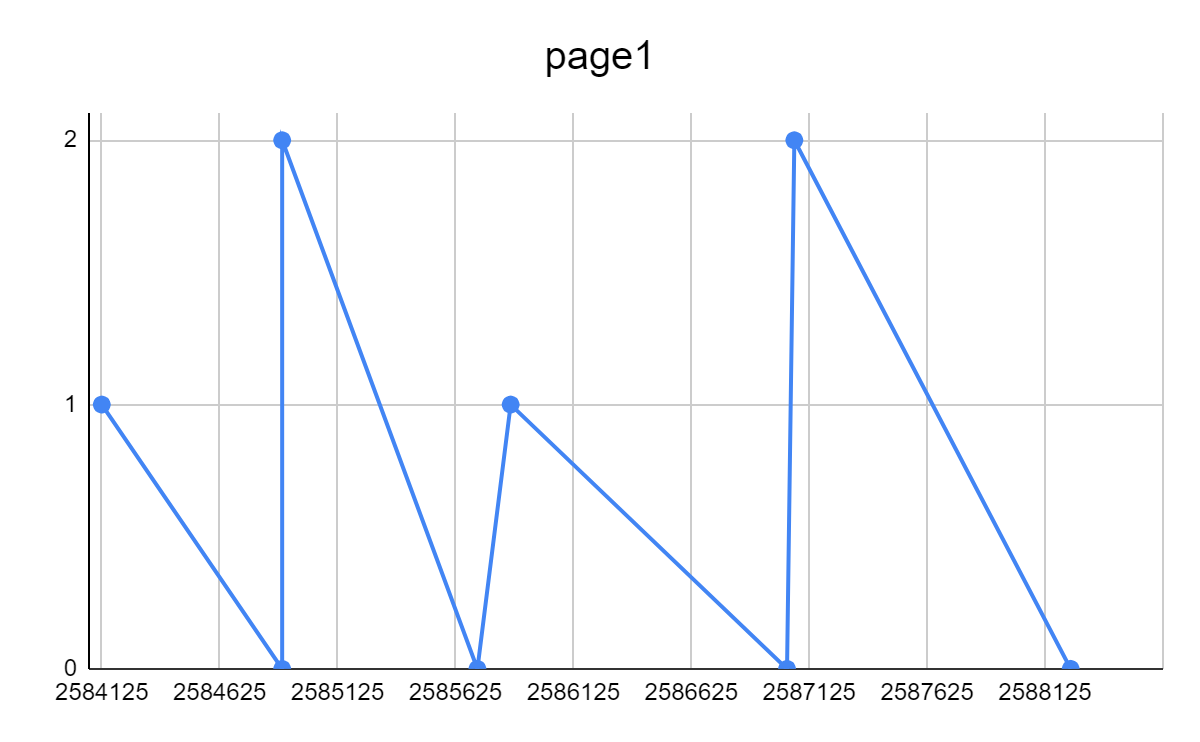
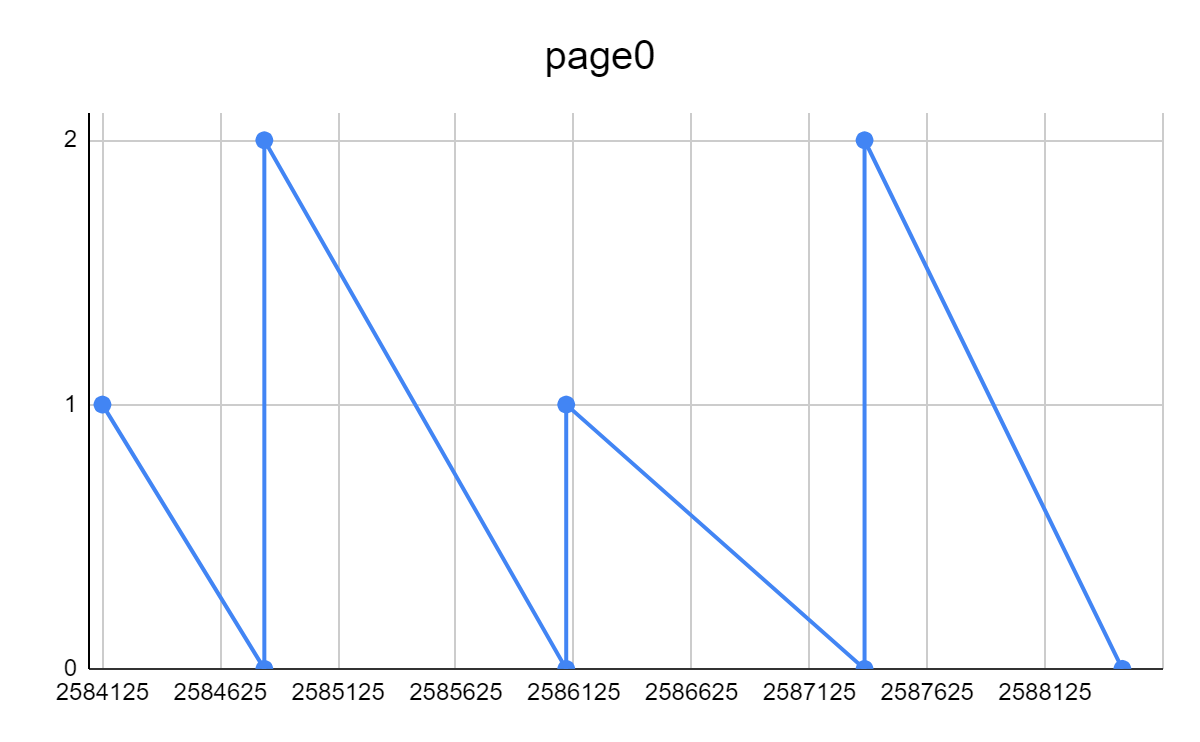


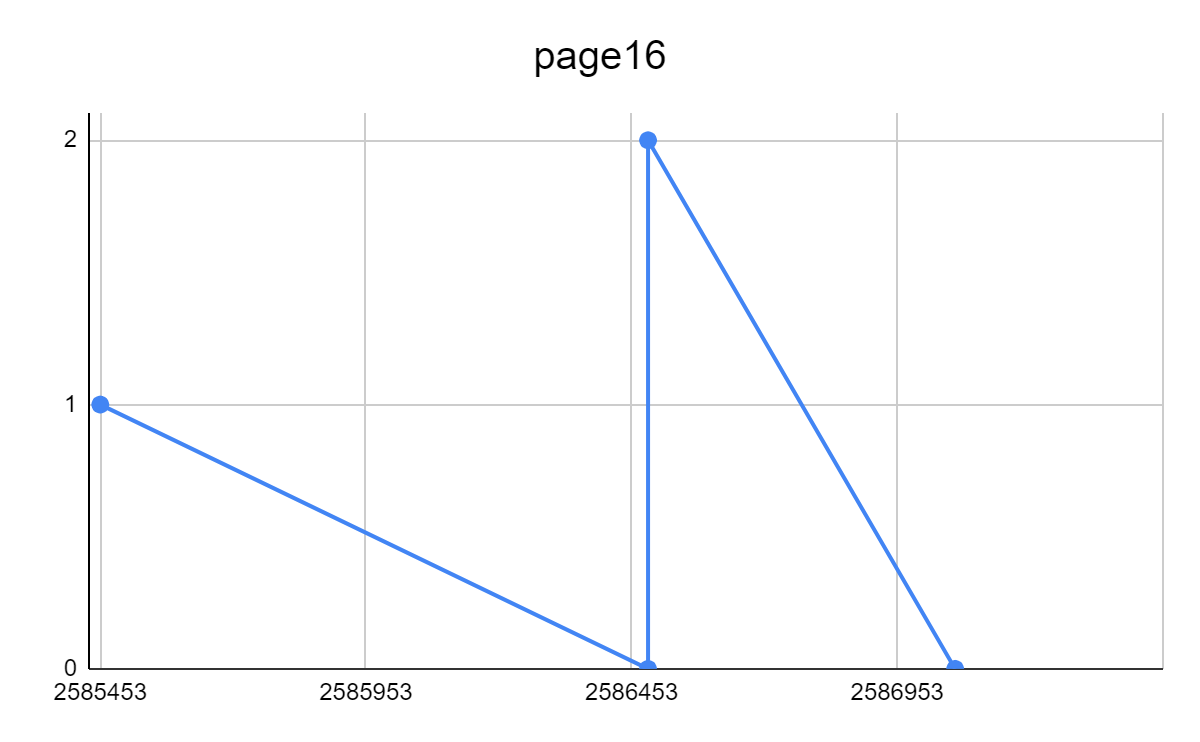
**Графики занятости страниц памяти:**

16 страниц.

Каждая занятость страницы имеет 3 состояния:

* + 0 – страница освобождена
  + 1 – страница взята на запись
  + 2 – страница взята на чтение



****

**Вывод:**

В ходе лабораторной работы были исследованы инструменты и механизмы взаимодействия процессов в Windows.

Была рассмотрена задача о писателях и читателях, которая затрагивает множество процессов (писателей и читателей). Они работают с общей областью памяти. Для обеспечения корректной работы нужно, чтобы:

* процессы не работали с одной и той же страницей памяти одновременно;
* данные, записанные писателем в некоторую страницу, были прочитаны каким-либо из читателей, иначе при новой попытке записи в то же место, они могут быть утеряны.

**Семафор** – это объект, который используется для контроля доступа нескольких потоков до общего ресурса. В общем случае это какая-то переменная, состояние которой изменяется каждым из потоков. Текущее состояние переменной определяет доступ к ресурсам.

**Семафор** *не имеет владельца*, он может быть изменен любым потоком. (это нужно понимать, особенно, если вы использовали бинарные семафоры)

**Мьютекс** (mutual exclusion — «взаимное исключение») — это базовый механизм синхронизации. Он предназначен для организации взаимоисключающего доступа к общим данным для нескольких потоков с использованием барьеров памяти.

**Мьютекс** *имеет владельца* - только поток, захвативший мьютекс, может его освободить.

**Захват мьютекса:** поток запрашивает монопольное использование общих данных, защищаемых мьютексом. Дальше два варианта развития событий: происходит захват мьютекса этим потоком (и в этом случае ни один другой поток не сможет получить доступ к этим данным) или поток блокируется (если мьютекс уже захвачен другим потоком).

**Освобождение мьютекса:** когда ресурс больше не нужен, текущий владелец должен вызвать функцию разблокирования, чтобы и другие потоки могли получить доступ к этому ресурсу. Когда мьютекс освобождается, доступ предоставляется одному из ожидающих потоков.

Ниже подробно расписан список функций использованных в работе, а также их параметры и назначение.

BOOL VirtualLock(

LPVOID lpAddress,

SIZE\_T dwSize

);

*VirtualLock* – блокирует указанную область виртуального адресного пространства процесса в физической памяти, гарантируя, что последующий доступ к этой области не приведет к сбою страницы. Если функция выполнена успешно, возвращаемое значение ненулевое, в противном – нулевое.

*lpAddress* – указатель на базовый адрес области страниц, которые должны быть заблокированы.

*dwSize* – размер области, подлежащей блокировке, в байтах. Область затронутых страниц включает все страницы, которые содержат один или несколько байтов в диапазоне от параметра *lpAddress* до (*lpAddress+dwSize*). Это означает, что диапазон в 2 байта, охватывающий границу страницы, приводит к блокировке обеих страниц.

**Семафоры :**

HANDLE CreateSemaphore(

LPSECURITY\_ATTRIBUTES lpSemaphoreAttributes,.

LONG lInitialCount,

LONG lMaximumCount,

LPCTSTR lpName

);

*CreateSemaphore* – cоздает или открывает именованный или неименованный объект семафора. В случае успешного выполнения функции возвращаемое значение представляет собой дескриптор объекта semaphore, в противном – NULL. Если именованный объект семафора существовал до вызова функции, функция GetLastError возвращает ERROR\_ALREADY\_EXISTS. В противном случае GetLastError возвращает 0.

*lInitialCount –* указатель на структуру SECURITY\_ATTRIBUTES . Если этот параметр имеет значение NULL, дескриптор не может наследоваться дочерними процессами.Начальное число для объекта семафора. Это значение должно быть больше или равно нулю и меньше или равно lMaximumCount. Состояние семафора сигнализируется, когда его число больше нуля, и без знака, когда оно равно нулю. Количество уменьшается на один раз, когда функция ожидания освобождает поток, ожидающий семафора. Количество увеличивается на указанную величину путем вызова функции ReleaseSemaphore.

*lMaximumCount –* максимальное число объектов семафора. Это значение должно быть больше нуля.

*lpName –* имя объекта семафора. При сравнении имен учитывается регистр. Если lpName соответствует имени существующего именованного объекта семафора, эта функция запрашивает право доступа SEMAPHORE\_ALL\_ACCESS . В этом случае параметры lInitialCount и lMaximumCount игнорируются, так как они уже заданы процессом создания. Если параметр lpSemaphoreAttributes не равен NULL, он определяет, можно ли наследовать дескриптор, но его член дескриптора безопасности игнорируется. Если lpName имеет значение NULL, объект семафора создается без имени.

BOOL ReleaseSemaphore(

HANDLE hSemaphore,

LONG lReleaseCount,

LPLONG lpPreviousCount

);

*ReleaseSemaphore* – увеличивает количество указанного объекта семафора на указанную величину. Если функция выполняется успешно, возвращается ненулевое значение, в противном – нулевое.

*hSemaphore* – дескриптор объекта семафора. Функция CreateSemaphore или OpenSemaphore возвращает этот дескриптор

*lReleaseCount –* сумма, на которую необходимо увеличить текущее число объекта семафора. Это значение должно быть больше нуля. Если указанное значение приведет к тому, что число семафора превысит максимальное число, указанное при создании семафора, счетчик не изменяется, и функция возвращает значение FALSE.

*lpPreviousCount –* указатель на переменную для получения предыдущего количества семафора. Этот параметр может иметь значение NULL , если предыдущее число не требуется.

HANDLE OpenSemaphore(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCTSTR lpName

);

*OpenSemaphore* – открывает существующий именованный объект семафора. В случае успешного выполнения функции возвращаемое значение является дескриптором объекта semaphore, в противном – NULL.

*dwDesiredAccess –* доступ к объекту семафора. Функция завершается ошибкой, если дескриптор безопасности указанного объекта не разрешает запрошенный доступ для вызывающего процесса.

*bInheritHandle* – если это значение равно TRUE, процессы, созданные этим процессом, наследуют дескриптор. В противном случае процессы не наследуют этот дескриптор.

*lpName –* имя открываемого семафора. Сравнение имен учитывает регистр

**Мьютексы :**

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes,

BOOL bInitialOwner,

LPCTSTR lpName

);

CreateMutex – создает или открывает именованный или неименованный объект мьютекса. Если функция выполняется успешно, возвращаемое значение является дескриптором для вновь созданного объекта мьютекса, в противном – NULL.

*lpMutexAttributes* – указатель на структуру SECURITY\_ATTRIBUTES. Если этот параметр имеет значение NULL, дескриптор не может быть унаследован дочерними процессами. Элемент lpSecurityDescriptor структуры задает дескриптор безопасности для нового мьютекса. Если lpMutexAttributes имеет значение NULL, мьютекс получает дескриптор безопасности по умолчанию.

*bInitialOwner* – если это значение равно TRUE и вызывающий объект создал мьютекс, вызывающий поток получает начальное владение объектом мьютекса. В противном случае вызывающий поток не получает права владения мьютексом.

*lpName* – имя объекта мьютекса. Имя ограничено MAX\_PATH символами. Сравнение имен учитывает регистр. Если lpName соответствует имени существующего именованного объекта мьютекса, эта функция запрашивает право доступа MUTEX\_ALL\_ACCESS . В этом случае параметр bInitialOwner игнорируется, так как он уже задан процессом создания. Если параметр lpMutexAttributes не равен NULL, он определяет, можно ли наследовать дескриптор, но его член дескриптора безопасности игнорируется. Если lpName имеет значение NULL, объект мьютекса создается без имени.

BOOL ReleaseMutex(

HANDLE hMutex

);

*ReleaseMutex* – освобождает владение указанным объектом мьютекса. Если функция выполнена успешно, возвращаемое значение ненулевое, в противном – нулевое.

*hMutex* – дескриптор объекта мьютекса. CreateMutex или функция OpenMutex возвращает этот дескриптор.

HANDLE OpenMutex(

DWORD dwDesiredAccess,

BOOL bInheritHandle,

LPCTSTR lpName

);

*OpenMutex* – открывает существующий именованный объект мьютекса. В случае успешного выполнения функции возвращаемое значение представляет собой дескриптор объекта мьютекса, в противном – NULL.

*dwDesiredAccess* – доступ к объекту мьютекса. Для использования мьютекса требуется только право доступа SYNCHRONIZE ; Чтобы изменить безопасность мьютекса, укажите MUTEX\_ALL\_ACCESS. Функция завершается ошибкой, если дескриптор безопасности указанного объекта не разрешает запрошенный доступ для вызывающего процесса.

*bInheritHandle* – если это значение равно TRUE, процессы, созданные этим процессом, наследуют дескриптор. В противном случае процессы не наследуют этот дескриптор.

*lpName* – имя открываемого мьютекса. Сравнение имен учитывает регистра.

**Задание 4.2.** Использование именованных каналов для реализации

сетевого межпроцессного взаимодействия.

**Указания к выполнению**:

1. Создайте два консольных приложения с меню (каждая выполняемая функция и/или операция должна быть доступна по отдельному пункту меню), которые выполняют:

* Приложение-сервер создает именованный канал (функция Win32 API – **CreateNamedPipe**), выполняет установление и отключение соединения (функции Win32 API – **ConnectNamedPipe**,  **DisconnectNamedPipe**), создает объект «событие» (функция Win32  API – **CreateEvent**) осуществляет ввод данных с клавиатуры и их  асинхронную запись в именованный канал (функция Win32 API – **WriteFile**), выполняет ожидание завершения операции ввода-вывода (функция Win32 API – **WaitForSingleObject**);
* Приложение-клиент подключается к именованному каналу (функция Win32 API – **CreateFile**), в асинхронном режиме считывает содержимое из именованного канала файла (функция Win32 API – **ReadFileEx**) и отображает на экран.

2. Запустите приложения и проверьте обмен данных между процессами. Запротоколируйте результаты в отчет. Дайте свои комментарии в отчете относительно выполнения функций Win32 API.

3. Подготовьте итоговый отчет с развернутыми выводами по заданию.

**Реализация программы**

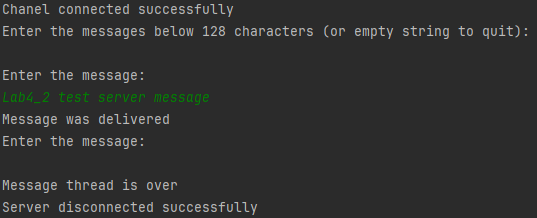
*Со стороны сервера:*

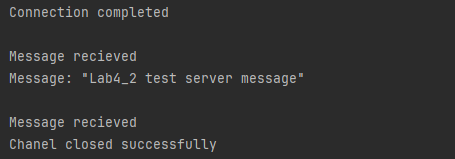
1. Создать именованный канал, функция **– CreateNamedPipe.**
2. Создать событие, функция **– CreateEvent.**
3. Вызвать функцию **ConnectNamedPipe**, которая ожидает соединения со стороны клиента.
4. Далее при соединении с клиентом помещаем сообщение в канал, функция **- WriteFile**. (первым параметром – дескриптор канала).
5. После обработки запроса вызываем функции для закрытия канала - **DisconnectNamedPipe**, **CloseHandle**.

*Со стороны клиента:*

1. Ожидать освобождения сервера, функция - **WaitNamedPipe**.
2. Вызвать функцию **CreateFile**, в качестве первого параметра имя канала, а второй тип доступа – чтение.
3. Получить ответ, функция **- ReadFileEx**. (первым параметром – дескриптор канала).
4. Закрыть канал после того, как обмен сообщениями закончен - функция **CloseHandle.**

**Пример работы :**





**Вывод 4.2:**

В ходе выполнения второй части лабораторной работы именованные каналы были использованы для реализации сетевого межпроцессного взаимодействия.

Именованный канал (Named Pipe) — это механизм взаимодействия между процессами в операционных системах, позволяющий передавать данные между ними. Они обеспечивают именованное сетевое соединение между процессами, даже если они работают на одном компьютере или на разных компьютерах в сети. Это особенно полезно в случаях, когда нужно передавать данные между процессами, которые работают в разных средах выполнения или в разных языках программирования.

Событие — гарантия, что только один процесс или поток сможет работать с каналом в определенный момент времени. Перед тем, как работать с каналом, процесс или поток проверят состояние события. Если оно сигнализирует о доступности канала, то процесс или поток получают доступ к каналу. В противном случае они ожидают, пока событие сигнализирует об освобождении канала другим процессом или потоком.

Для того, чтобы увидеть обмен информацией, нужно было одновременно запустить приложения сервера и клиента. Каждый процесс должен был подключиться каналу. Затем осуществлялся ввод данных в приложении сервера, которые затем считывались и выводились на экран в приложении клиента.

Ниже подробно расписан список функций использованных в работе, а также их параметры и назначение.

**Именованные каналы:**

HANDLE CreateNamedPipe(

LPCTSTR lpName,

DWORD dwOpenMode,

DWORD dwPipeMode,

DWORD nMaxInstances,

DWORD nOutBufferSize,

DWORD nInBufferSize,

DWORD nDefaultTimeOut,

LPSECURITY\_ATTRIBUTES lpSecurityAttributes

);

*CreateNamedPipe* – создает экземпляр именованного канала и возвращает дескриптор для последующих операций с каналом. Серверный процесс именованного канала использует эту функцию либо для создания первого экземпляра определенного именованного канала и установления его основных атрибутов, либо для создания нового экземпляра существующего именованного канала.

*lpName* – уникальное имя канала. Эта строка должна иметь следующий вид: *\\.\pipe\имя канала*

Часть имени канала может содержать любой символ, кроме обратной косой черты, включая цифры и специальные символы. Длина всей строки имени канала может составлять до 256 символов. Имена каналов не чувствительны к регистру.

*dwOpenMode* – режим открытия. Функция завершается с ошибкой, если dwOpenMode указывает что-либо, отличное от 0, или флаги, перечисленные в следующих таблицах. Этот параметр должен указывать один из режимов доступа к каналу. Один и тот же режим должен быть указан для каждого экземпляра канала. В нашем случае PIPE\_ACCESS\_OUTBOUND (0x00000002) – поток данных в канале передается только от сервера к клиенту. Этот режим предоставляет серверу эквивалент GENERIC\_WRITE доступ к каналу. Клиент должен указать доступ GENERIC\_READ при подключении к каналу.

*dwPipeMode* – режим канала. Функция завершается с ошибкой, если dwPipeMode указывает что-либо, отличное от 0 или некоторых флагов. В нашем случае указываются флаги – PIPE\_TYPE\_MESSAGE (0x00000004), PIPE\_READMODE\_MESSAGE (0x00000002) и PIPE\_WAIT (0x00000000).

PIPE\_TYPE\_MESSAGE *–* данные записываются в канал в виде потока сообщений. Канал обрабатывает байты, записанные во время каждой операции записи, как единицу сообщения.

PIPE\_READMODE\_MESSAGE *–* данные считываются из канала в виде потока сообщений. Этот режим можно использовать, только если также указан PIPE\_TYPE\_MESSAGE.

PIPE\_WAIT *–* включен режим блокировки. Когда дескриптор канала указан в функции ReadFile, WriteFile или ConnectNamedPipe, операции не завершаются до тех пор, пока не появятся данные для чтения, не будут записаны все данные или не будет подключен клиент.

*nMaxInstances –* максимальное количество экземпляров, которые могут быть созданы для этого канала. Это значение может быть указано в первом экземпляре канала; такое же количество должно быть указано для других экземпляров канала. Допустимые значения находятся в диапазоне от 1 до PIPE\_UNLIMITED\_INSTANCES (255).

*nOutBufferSize* – количество байт, которое нужно зарезервировать для выходного буфера. Обсуждение размера буферов именованного канала смотрите в следующем разделе Замечаний.

*nInBufferSize* – количество байт, которые необходимо зарезервировать для входного буфера. Обсуждение размера буферов именованного канала см. В следующем разделе "Примечания".

nDefaultTimeOut – значение тайм-аута по умолчанию в миллисекундах, если функция WaitNamedPipe указывает NMPWAIT\_USE\_DEFAULT\_WAIT. Каждый экземпляр именованного канала должен указывать одно и то же значение. Нулевое значение приведет к тайм-ауту по умолчанию в 50 миллисекунд.

lpSecurityAttributes – указатель на структуру SECURITY\_ATTRIBUTES, которая задает дескриптор безопасности для нового именованного канала и определяет, могут ли дочерние процессы наследовать возвращаемый дескриптор. Если lpSecurityAttributes равно NULL, именованный канал получает дескриптор безопасности по умолчанию, и дескриптор не может быть унаследован. Списки управления доступом в дескрипторе безопасности по умолчанию для именованного канала предоставляют полный контроль учетной записи LocalSystem, администраторам и владельцу-создателю. Они также предоставляют доступ на чтение членам группы "Все" и анонимной учетной записи.

BOOL ConnectNamedPipe(

HANDLE hNamedPipe,

LPOVERLAPPED lpOverlapped

);

*ConnectNamedPipe* – позволяет серверному процессу именованного канала ожидать, пока клиентский процесс подключится к экземпляру именованного канала. Клиентский процесс подключается, вызывая либо CreateFile или CallNamedPipe. Если операция выполняется синхронно, ConnectNamedPipe не возвращает результат до завершения операции. Если функция выполнена успешно, возвращаемое значение ненулевое. Если операция выполняется синхронно, ConnectNamedPipe не возвращает результат до завершения операции. Если функция выполнена успешно, возвращаемое значение ненулевое. Если клиент подключается до вызова функции, функция возвращает ноль, а GetLastError возвращает ERROR\_PIPE\_CONNECTED .

*hNamedPipe* – дескриптор серверного конца экземпляра именованного канала. Этот дескриптор возвращается функцией CreateNamedPipe. Указатель на OVERLAPPED структуру. Если hNamedPipe был открыт с помощью FILE\_FLAG\_OVERLAPPED, параметр lpOverlapped не должен быть равен NULL. Он должен указывать на допустимую структуру OVERLAPPED . Если hNamedPipe был открыт с помощью FILE\_FLAG\_OVERLAPPED и lpOverlapped имеет значение NULL, функция может неправильно сообщить о завершении операции подключения. Если hNamedPipe был создан с помощью FILE\_FLAG\_OVERLAPPED, а lpOverlapped не равен NULL, то OVERLAPPED структура должна содержать дескриптор объекта события ручного сброса (который сервер может создать с помощью функции CreateEvent ). Если hNamedPipe не был открыт с помощью FILE\_FLAG\_OVERLAPPED, функция не возвращается до тех пор, пока не будет подключен клиент или не возникнет ошибка. Успешные синхронные операции приводят к тому, что функция возвращает ненулевое значение, если клиент подключается после вызова функции.

BOOL DisconnectNamedPipe(

[in] HANDLE hNamedPipe

);

*DisconnectNamedPipe* – отключает серверную часть экземпляра именованного канала от клиентского процесса. Возвращаемое значение –

eсли функция выполняется успешно, возвращается ненулевое значение, в противном случае – нулевое.

*hNamedPipe* – дескриптор экземпляра именованного канала. Этот дескриптор должен быть создан функцией CreateNamedPipe.

**События:**

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpEventAttributes,

BOOL bManualReset,

BOOL bInitialState,

LPTSTR lpName

);

*CreateEvent –* функция CreateEvent создает именованный или неназванный объект события. Дескриптор объекта event указывает на успех. Если именованный объект event существовал до вызова функции, функция возвращает дескриптор существующего объекта и GetLastError возвращает ERROR\_ALREADY\_EXISTS . Значение NULL указывает на сбой.

*lpEventAttributes* – указатель на структуру SECURITY\_ATTRIBUTES. Если этот параметр имеет значение NULL, дескриптор не может быть унаследован дочерними процессами. Член структуры lpSecurityDescriptor задает дескриптор безопасности для нового события. Если значение lpEventAttributes равно NULL, событие получает дескриптор безопасности по умолчанию.

*bManualReset* – если этот параметр имеет значение TRUE, функция создает объект события с ручным сбросом, который требует использования функции ResetEvent для установки состояния события на nonsignaled . Если этот параметр имеет значение FALSE, функция создает объект события с автоматическим сбросом, и система автоматически сбрасывает состояние события на nonsignaled после того, как был выпущен один ожидающий поток.

*bInitialState* – если этот параметр имеет значение TRUE, сигнализируется начальное состояние объекта event; в противном случае оно не сигнализируется.

*lpName* – имя объекта события. При сравнении имен учитывается регистр. Если lpName совпадает с именем существующего именованного объекта event, эта функция запрашивает право доступа EVENT\_ALL\_ACCESS. В этом случае параметры bManualReset и bInitialState игнорируются, поскольку они уже были установлены в процессе создания. Если параметр lpEventAttributes не имеет значения NULL, он определяет, может ли дескриптор быть унаследован, но его элемент дескриптора безопасности игнорируется. Если lpName равно NULL, объект события создается без имени.

**Код программ:**

#include <ctime>

#include <iostream>

#include <windows.h>

const int pageCount = 16, processCount=8, pageSize=4096;

int main(){

std::string mutexname="IOmutex";

HANDLE wSemaphore[pageCount],

rSemaphore[pageCount],

file= CreateFile("filename", GENERIC\_WRITE | GENERIC\_READ, 0, nullptr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0),

mapFile=CreateFileMapping(file, nullptr, PAGE\_READWRITE, 0, pageSize \* pageCount, "filemap");

LPVOID viewFile= MapViewOfFile(mapFile,FILE\_MAP\_ALL\_ACCESS,0,0,pageCount\*pageSize);

for (int i=0;i<pageCount;i++){

wSemaphore[i]= CreateSemaphoreA(nullptr,1,1,("wSemaphore"+std::to\_string(i)).c\_str());

rSemaphore[i]=CreateSemaphoreA(nullptr,0,1,("rSemaphore"+std::to\_string(i)).c\_str());

}

VirtualLock(viewFile, pageSize \* pageCount);

HANDLE processHandles[processCount\*2];

for (int i=0;i<processCount;i++){

STARTUPINFO si;

PROCESS\_INFORMATION pi;

SECURITY\_ATTRIBUTES secureAttr = { sizeof(secureAttr), nullptr, TRUE };

ZeroMemory( &si, sizeof(si) );

ZeroMemory( &pi, sizeof(pi) );

si.cb = sizeof(si);

si.cb = sizeof(si);

si.dwFlags |= STARTF\_USESTDHANDLES;

si.hStdInput = nullptr;

si.hStdError = nullptr;

si.hStdOutput=CreateFile(("writeLog" + std::to\_string(i) + ".txt").c\_str(), GENERIC\_WRITE | GENERIC\_READ, 0, &secureAttr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0);;

int process= CreateProcess((LPCTSTR)"writer.exe", // Process handle not inheritable

nullptr,

nullptr, // Process handle not inheritable

nullptr, // Thread handle not inheritable

true, // Set handle inheritance to FALSE

0, // No creation flags

nullptr, // Use parent's environment block

nullptr, // Use parent's starting directory

&si, // Pointer to STARTUPINFO structure

&pi);

if (!process) processHandles[i]=pi.hProcess;

}

for (int i=0;i<processCount;i++){

STARTUPINFO si;

PROCESS\_INFORMATION pi;

SECURITY\_ATTRIBUTES secureAttr = { sizeof(secureAttr), nullptr, TRUE };

ZeroMemory( &si, sizeof(si) );

ZeroMemory( &pi, sizeof(pi) );

si.cb = sizeof(si);

si.dwFlags |= STARTF\_USESTDHANDLES;

si.hStdInput = nullptr;

si.hStdError = nullptr;

si.hStdOutput=CreateFile(("readLog" + std::to\_string(i) + ".txt").c\_str(), GENERIC\_WRITE | GENERIC\_READ, 0, &secureAttr, CREATE\_ALWAYS, FILE\_ATTRIBUTE\_NORMAL, 0);

int process= CreateProcess((LPCTSTR)"reader.exe", // Process handle not inheritable

nullptr,

nullptr, // Process handle not inheritable

nullptr, // Thread handle not inheritable

true, // Set handle inheritance to FALSE

0, // No creation flags

nullptr, // Use parent's environment block

nullptr, // Use parent's starting directory

&si, // Pointer to STARTUPINFO structure

&pi);

if (!process) processHandles[processCount+i]=pi.hProcess;

}

Sleep(15000);

WaitForMultipleObjects(processCount\*2,processHandles,true,INFINITE);

for (int i=0;i<processCount\*2;i++){

CloseHandle(processHandles[i]);

}

VirtualUnlock(viewFile, pageSize \* pageCount);

UnmapViewOfFile(viewFile);

CloseHandle(mapFile);

CloseHandle(file);

CloseHandle(rSemaphore);

CloseHandle(wSemaphore);

return 0;

}

#include <ctime>

#include <iostream>

#include <windows.h>

const int pageCount = 16, processCount=8;

int main() {

srand(GetCurrentProcessId());

std::string mutexname="IOmutex";

HANDLE wSemaphore[pageCount] ,

rSemaphore[pageCount],

mappingHandle = OpenFileMapping(FILE\_MAP\_READ | FILE\_MAP\_WRITE, FALSE, (LPCTSTR)"filemap");

std::string ProccessId = std::to\_string (GetCurrentProcessId());

HANDLE handleStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

for (int i=0;i<pageCount;i++){

wSemaphore[i]=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE,("wSemaphore"+std::to\_string(i)).c\_str());

rSemaphore[i]=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE,("rSemaphore"+std::to\_string(i)).c\_str());

}

if (mappingHandle){

for (int i=0;i<3;i++) {

std::string str ="Process with id " + ProccessId + " waiting :" + std::to\_string(GetTickCount()) + "\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

int sem = WaitForMultipleObjects(pageCount, wSemaphore, false, INFINITE);

str = "Process with id " + ProccessId + " take page:" + std::to\_string(sem) + ":" +

std::to\_string(GetTickCount()) + "\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

Sleep( rand() % 1000+ 500);

str = "Process with id " + ProccessId + " free page " + std::to\_string(sem) + ":" +

std::to\_string(GetTickCount()) + "\n\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

ReleaseSemaphore(rSemaphore[sem], 1, nullptr);

}

}

for (int i=0;i<pageCount;i++){

CloseHandle(rSemaphore[i]);

CloseHandle(wSemaphore[i]);

}

CloseHandle(mappingHandle);

CloseHandle(handleStdOut);

return 0;

}

#include <ctime>

#include <iostream>

#include <windows.h>

const int pageCount = 16, processCount=8;

int main() {

srand(GetCurrentProcessId());

std::string mutexname="IOmutex";

HANDLE wSemaphore[pageCount] ,

rSemaphore[pageCount],

mappingHandle = OpenFileMapping(FILE\_MAP\_READ | FILE\_MAP\_WRITE, FALSE, (LPCTSTR)"filemap");

std::string ProccessId = std::to\_string (GetCurrentProcessId());

HANDLE handleStdOut = GetStdHandle(STD\_OUTPUT\_HANDLE);

for (int i=0;i<pageCount;i++){

wSemaphore[i]=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE,("wSemaphore"+std::to\_string(i)).c\_str());

rSemaphore[i]=OpenSemaphore(SEMAPHORE\_ALL\_ACCESS, FALSE,("rSemaphore"+std::to\_string(i)).c\_str());

}

if (mappingHandle) {

for (int i = 0; i < 3; i++) {

std::string str ="Process with id " + ProccessId + " waiting :" + std::to\_string(GetTickCount()) + "\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

int sem = WaitForMultipleObjects(pageCount, rSemaphore, false, INFINITE);

str = "Process with id " + ProccessId + " take page:" + std::to\_string(sem) + ":" +

std::to\_string(GetTickCount()) + "\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

Sleep( rand() % 1000+500);

str = "Process with id " + ProccessId + " free page " + std::to\_string(sem) + ":" +

std::to\_string(GetTickCount()) + "\n\n";

WriteFile(handleStdOut, str.c\_str(), str.length(), nullptr, nullptr);

ReleaseSemaphore(wSemaphore[sem], 1, nullptr);

}

}

for (int i=0;i<pageCount;i++){

CloseHandle(rSemaphore[i]);

CloseHandle(wSemaphore[i]);

}

CloseHandle(mappingHandle);

CloseHandle(rSemaphore);

CloseHandle(wSemaphore);

CloseHandle(handleStdOut);

return 0;

}

#include <iostream>  
#include <windows.h>  
using std::cout;  
const int size = 128;  
  
bool ConnectToPipe(HANDLE& hPipe, HANDLE& hEvent){  
 hPipe = CreateNamedPipe(TEXT("\\\\.\\pipe\\pipename"), PIPE\_ACCESS\_OUTBOUND, PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT,  
 PIPE\_UNLIMITED\_INSTANCES, size, size, 0, nullptr);  
  
 hEvent = CreateEvent(nullptr, false, false, nullptr);  
 OVERLAPPED overlapped = OVERLAPPED();  
 bool isConnected = false;  
 if (hPipe != INVALID\_HANDLE\_VALUE && hEvent != INVALID\_HANDLE\_VALUE){  
 overlapped.hEvent = hEvent;  
 isConnected = ConnectNamedPipe(hPipe, &overlapped);  
 WaitForSingleObject(hEvent, INFINITE);  
 }  
 return isConnected;  
}  
  
bool SendMyMessage(HANDLE& hPipe, HANDLE& hEvent, bool& endFlag){  
 std::string message;  
 OVERLAPPED overlapped = OVERLAPPED();  
 overlapped.hEvent = hEvent;  
  
 cout << "Enter the message:\n";  
 std::getline(std::cin, message);  
 if (message.length() > size){  
 printf("Message is above %d characters\n", size);  
 return false;  
 }  
 if (!strcmp(message.c\_str(), "")) endFlag = true;  
 return (WriteFile(hPipe, message.c\_str(), size, nullptr, &overlapped) && WaitForSingleObject(overlapped.hEvent, INFINITE) == WAIT\_OBJECT\_0);  
}  
  
  
int main(){  
 HANDLE pipeHandle = nullptr, eventHandle = nullptr;  
 bool isConnected, endFlag = false;  
  
 isConnected = ConnectToPipe(pipeHandle, eventHandle);  
  
 if (isConnected){  
 cout << "Chanel connected successfully\n";  
 printf("Enter the messages below %d characters (or empty string to quit):\n\n", size);  
 while(!endFlag){  
 isConnected = SendMyMessage(pipeHandle, eventHandle, endFlag);  
 if (endFlag) cout << "Message thread is over\n";  
 else if (isConnected) cout << "Message was delivered\n";  
 else cout << "Message wasn't delivered\n";  
 }  
  
 if (DisconnectNamedPipe(pipeHandle))  
 cout << "Server disconnected successfully\n";  
 else cout << "Server disconnected unsuccessfully\n";  
  
 if (pipeHandle != INVALID\_HANDLE\_VALUE) CloseHandle(pipeHandle);  
 if (eventHandle!= INVALID\_HANDLE\_VALUE) CloseHandle(eventHandle);  
  
 system("pause");  
 }  
 else cout << "Chanel wasn't connected\n";  
}

#include <iostream>  
#include <windows.h>  
using std::cout;  
const int size = 128;  
HANDLE pipeHandle = nullptr;  
  
void WINAPI callback(DWORD dwErrorCode) {  
 if (!dwErrorCode) cout << "\nMessage recieved\n";  
 else cout << "\nMessage haven't recieved\n";  
}  
  
bool ConnectToPipe(HANDLE& hPipe){  
 WaitNamedPipe(R"(\\.\pipe\pipename)", NMPWAIT\_WAIT\_FOREVER);  
 hPipe = CreateFile(TEXT("\\\\.\\pipe\\pipename"), GENERIC\_READ, 0, nullptr, OPEN\_EXISTING, FILE\_FLAG\_OVERLAPPED, nullptr);  
 return hPipe != INVALID\_HANDLE\_VALUE;  
}  
  
bool GetMyMessage(HANDLE& hPipe, char message[]){  
 OVERLAPPED overlapped = OVERLAPPED();  
 bool flag = ReadFileEx(hPipe, message, size, &overlapped,  
 reinterpret\_cast<LPOVERLAPPED\_COMPLETION\_ROUTINE>(callback));  
 SleepEx(INFINITE, true);  
 if (!strcmp(message, "")) flag = false;  
 return flag;  
}  
  
  
int main(){  
 char message[size];  
  
 if (ConnectToPipe(pipeHandle)){  
 cout << "Connection completed\n";  
 while(GetMyMessage(pipeHandle, message))  
 printf("Message: \"%s\"\n", message);  
 }  
 else cout << "Connection failed\n";  
 if (CloseHandle(pipeHandle))  
 cout << "Chanel closed successfully\n";  
 else cout << "Chanel wasn't closed\n";  
 system("pause");  
}