//1 вопрос

Он также создает 50 потоков, где четные потоки выполняют депозит, а нечетные - снятие денег. После выполнения всех потоков выводится конечный баланс.

Однако, у вас есть несколько ошибок в коде:

Вы создаете массив HANDLE\* handles с 49 элементами, но в цикле запускаете 50 потоков. Это может привести к неопределенному поведению или ошибкам. Используйте 50 элементов в массиве handles.

В функции DoWithdraw после проверки баланса вы вызываете Sleep(20);. Это может привести к неправильным результатам, так как другие потоки могут изменить баланс в это время. Уберите это участок кода или синхронизируйте его, чтобы избежать проблем.

В функции GetBalance вы читаете баланс из файла при каждом вызове. Это неэффективно. Лучше читать баланс один раз и хранить его в памяти до изменения.

Вы не освобождаете ресурсы, выделенные под массив handles. Используйте delete[] handles; после завершения работы с потоками.

Вы используете магические числа для депозита и снятия денег. Лучше создать константы или параметризовать их, чтобы код был более читаемым и настраиваемым

Для определения участков кода, которые требуют синхронизации, вы можете использовать следующий подход:

Анализ критических ресурсов: Идентифицируйте ресурсы или данные, к которым могут обращаться несколько потоков одновременно и которые могут привести к состояниям гонки или несогласованности. В вашем коде это файл balance.txt.

Исследование операций над ресурсами: Определите, какие операции выполняются с этими ресурсами, и выделите те участки кода, где происходит доступ к критическим ресурсам. В вашем коде это чтение и запись в файл в функциях ReadFromFile и WriteToFile.

Рассмотрение последовательности операций: Рассмотрите, какие операции выполняются в потоках над критическими ресурсами. Определите, какие последовательности операций могут привести к состояниям гонки или несогласованности. В вашем коде это операции чтения и записи файла.

Разработка синхронизации: Внесите изменения в код для обеспечения синхронизации доступа к критическим ресурсам. В вашем случае, вы используете CRITICAL\_SECTION для блокировки доступа к файлу balance.txt, обеспечивая, что только один поток имеет к нему доступ в определенный момент времени.

Тестирование: Запустите приложение с несколькими потоками или экземплярами и протестируйте его, чтобы убедиться, что синхронизация работает правильно и что не возникает состояний гонки или других проблем с доступом к критическим ресурсам.

Этот процесс обычно включает в себя анализ кода, выявление участков, которые могут потребовать синхронизации, и внесение изменений для обеспечения корректной работы при многозадачности.

CRITICAL\_SECTION и MUTEX - это два механизма синхронизации, предоставляемых операционной системой Windows для обеспечения многозадачности и согласованного доступа к ресурсам. Они имеют различия в своем использовании и производительности, и каждый из них подходит для определенных сценариев.

CRITICAL\_SECTION:

Производительность: CRITICAL\_SECTION, как правило, более быстрый, чем MUTEX, потому что он оптимизирован для использования в пределах одного процесса.

Применение: Он используется для синхронизации потоков внутри одного процесса. CRITICAL\_SECTION - это более легковесный механизм и хорошо подходит для ситуаций, где несколько потоков должны координировать доступ к общим ресурсам в пределах одного процесса.

Владение: CRITICAL\_SECTION не имеет владельца. Это означает, что тот же поток, который заблокировал критическую секцию, может её разблокировать, что делает его менее подходящим для некоторых сложных сценариев.

MUTEX:

Производительность: MUTEX может быть более медленным, чем CRITICAL\_SECTION, так как он может использоваться для синхронизации между процессами, и это требует дополнительной системной сложности.

Применение: MUTEX может использоваться для синхронизации как внутри одного процесса, так и между разными процессами. Он подходит для сценариев, где необходима синхронизация между различными процессами.

Владение: MUTEX обычно связан с владельцем, что означает, что только поток, который заблокировал MUTEX, может его разблокировать. Это может быть полезно в некоторых случаях, когда требуется более жесткий контроль над доступом к ресурсам.

В каких ситуациях их следует использовать:

Используйте CRITICAL\_SECTION, когда вам нужна синхронизация между потоками внутри одного процесса, и вы хотите достичь более высокой производительности.

Используйте MUTEX, когда вам нужна синхронизация между потоками, работающими в разных процессах, или когда вам нужно более жесткое управление доступом к ресурсам, включая владение мьютексом.

Оба механизма имеют свои преимущества и недостатки, и выбор зависит от конкретных требований вашего приложения.

В вашем коде FileLockingCriticalSection используется для синхронизации доступа к файлу balance.txt. Он является экземпляром CRITICAL\_SECTION, и, в этом конкретном случае, это может быть правильным способом синхронизации.

CRITICAL\_SECTION и Mutex оба могут использоваться для синхронизации доступа к ресурсам, включая файлам. Основное различие между ними заключается в их области видимости и производительности.

CRITICAL\_SECTION обычно более производительный, чем Mutex, когда речь идет о синхронизации между потоками в пределах одного процесса. Ваш код выполняет синхронизацию доступа к файлу balance.txt в рамках одного процесса, и, по-видимому, в этом случае использование CRITICAL\_SECTION является разумным выбором.

Если бы ваше приложение требовало синхронизации между разными процессами, то использование Mutex было бы более подходящим. Однако в данном контексте CRITICAL\_SECTION может хорошо справляться с задачей синхронизации доступа к файлу.

Важно правильно управлять и освобождать CRITICAL\_SECTION после использования, как вы делаете в функции DeleteCriticalSection(&FileLockingCriticalSection), чтобы избежать утечек ресурсов.