1. Нулевой блок:
   1. Win32 API

Для использования данного API в языке c++ подключаются библиотеки Windows.h, conio.h, process.h. В них определено множество деректив #define, необходимых для работы с API (напр. переопределены базовые типы данных)

* 1. Потоком в Windows называется объект ядра, которому операционная система выделяет процессорное время для выполнения приложения. Каждому потоку принадлежат следующие ресурсы:
* код исполняемой функции;
* набор регистров процессора;
* стек для работы приложения;
* стек для работы операционной системы;
* блок окружения, который содержит служебную информацию для работы потока.

Разделяют рабочие потоки (working threads) – потоки, с которыми может работать пользователь, и потоки интерфейса пользователя (user interface threads) – потоки, работающие без участия пользователя.

Для создания потоков используются функции:

HANDLE CreateThread(

LPSECURITY\_ATTRIBUTES lpThreadAttributes, // атрибуты защиты

DWORD dwStackSize, // размер стека потока в байтах

LPTHREAD\_START\_ROUTINE lpStartAddress, // адрес исполняемой функции

LPVOID lpParameter, // адрес параметра

DWORD dwCreationFlags, // флаги создания потока

LPDWORD lpThreadId // идентификатор потока

);

uintptr\_t \_beginthreadex(

void \*security,

unsigned stack\_size,

unsigned ( \_\_stdcall \*start\_address )( void \* ),

void \*arglist,

unsigned initflag,

unsigned \*thrdaddr

);

Примеры использования (лаб. 1):

HANDLE hThread;

DWORD IDThread;

auto param = new Array(a, n);

hThread = (HANDLE)\_beginthreadex(NULL, 0, &worker, (void\*)param, 0, (unsigned int\*)&IDThread);

if (hThread == NULL)

return GetLastError();

//hThread = CreateThread(nullptr, 0, worker, (LPVOID)param, 0, &IDThread); //another way to open thread

Вызываемая ф-ция:

//DWORD WINAPI worker(LPVOID param) //name of function, if we use "another way"

unsigned \_\_stdcall worker(void\* param){…}

* 1. Мьютекс – объект ядра, который находится в сигнальном состоянии, если он не принадлежит ни одному потоку. В противном случае мьютекс находится в несигнальном состоянии. Одновременно мьютекс может принадлежать только одному потоку.

Создается мьютекс вызовом функции CreateMutex, которая имеет следующий прототип:

HANDLE CreateMutex(

LPSECURITY\_ATTRIBUTES lpMutexAttributes, // атрибуты защиты

BOOL bInitialOwner, // начальный владелец мьютекса

LPCTSTR lpName // имя мьютекса

);Мьютекс захватывается потоком посредством любой функции ожидания, а освобождается функцией ReleaseMutex, которая имеет следующий прототип:

BOOL ReleaseMutex(

HANDLE hMutex // дескриптор мьютекса

);

Для того

чтобы получить доступ к уже созданному мьютексу, поток может также использовать функцию OpenMutex,

которая имеет следующий прототип:

HANDLE OpenMutex(

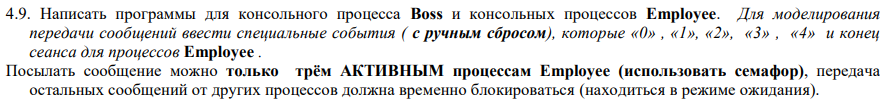
DWORD dwDesiredAccess, // доступ к мьютексу

BOOL bInheritHandle // свойство наследования

LPCTSTR lpName // имя мьютекса

);

К сожалению, в моём варианте лабораторной работы изпользовались семафоры 😊



* 1. Событием называется оповещение о некотором выполненном действии. В программировании события используются для оповещения одного потока о том, что другой поток выполнил некоторое действие.

В операционных системах Windows события описываются объектами ядра Events. При этом различают два типа событий:

* события с ручным сбросом;
* события с автоматическим сбросом.

Различие между этими типами событий заключается в том, что событие с ручным сбросом можно перевести в несигнальное состояние только посредством вызова функции ResetEvent, а событие с автоматическим сбросом переходит в несигнальное состояние как при помощи функции ResetEvent, так и при помощи функции ожидания. При этом отметим, что если события с автоматическим сбросом ждут несколько потоков, используя

функцию WaitForSingleObject, то из состояния ожидания освобождается только один из этих потоков.

Создаются события вызовом функции CreateEvent, которая имеет следующий прототип:

HANDLE CreateEvent(

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, // атрибуты защиты

BOOL bManualReset, // тип события

BOOL bInitialState, // начальное состояние события

LPCTSTR lpName // имя события

);

Примеры использования (лаб. 3):

HANDLE hWorkEnterSectionEvent;

hWorkEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

WaitForSingleObject(hWorkEnterSectionEvent, INFINITE);

ResetEvent(hWorkEnterSectionEvent);

//другой поток

SetEvent(hWorkEnterSectionEvent);

(лаб. 4):

Процесс 1:

HANDLE\* events;

events = new HANDLE[6];

std::string s = "Event\_0";

for (int i = 0; i < 6; ++i)

{

events[i] = CreateEvent(NULL, TRUE, FALSE, (LPCWSTR)s.c\_str());

s[s.size() - 1]++;

}

//использование

switch (WaitForMultipleObjects(6, events, FALSE, INFINITE) - WAIT\_OBJECT\_0)

{…}

//закрытие

for (int i = 0; i < 6; ++i)

{

CloseHandle(events[i]);

}

delete[] events;

Процесс 2:

HANDLE\* events;

events = new HANDLE[6];

std::string s = "Event\_0";

for (int i = 0; i < 6; ++i)

{

events[i] = OpenEvent(EVENT\_ALL\_ACCESS, TRUE, (LPCWSTR)s.c\_str());

s[s.size() - 1]++;

}

//использование

SetEvent(events[i]);

//закрытие

delete[] events;

1. Первый блок:
2. Второй блок:
   1. ООП декомпозиция — это процесс разделения сложной системы на более мелкие и управляемые части, что позволяет создавать более доступные для разработки, тестирования и поддержки компоненты. Этот подход также включает разделение функциональности на классы и методы для улучшения читаемости и использования кода.
   2. Статический полиморфизм — это возможность языка программирования определить, какой метод использовать на основе типов аргументов, переданных в метод, еще на этапе компиляции. То есть компилятор определяет, какой метод вызвать до запуска программы.
   3. Инкапсуляция — это основной принцип объектно-ориентированного программирования, который позволяет скрывать внутреннюю реализацию объекта от пользователя и предоставлять только необходимый интерфейс для взаимодействия с ним. Пользователь не знает, как объект управляет своими данными и функциональностью, но может использовать его методы для выполнения задач. Инкапсуляция способствует созданию безопасного и модульного кода, который может быть изменен без влияния на другие части программы.
3. Третий блок:
   1. Паттерн строитель (Builder Pattern) представляет собой метод проектирования, позволяющий пошагово создавать сложные объекты с различными конфигурациями. Он включает в себя создание класса-строителя (Builder), ответственного за сборку объекта путем установки его свойств посредством методов. Для создания объекта вызывается метод build(). Применение этого паттерна можно увидеть, например, при формировании заказа в интернет-магазине или создании SQL-запросов, где классы-строители устанавливают соответствующие свойства объектов, позволяя создавать разные конфигурации без создания множества конструкторов. Это способствует более читаемому и поддерживаемому коду, инкапсулируя процесс создания сложных объектов.
   2. Паттерн декоратор (Decorator Pattern) предоставляет возможность добавлять дополнительную функциональность к объектам без изменения их основного кода. Он основывается на создании класса-декоратора (Decorator), содержащего ссылку на декорируемый объект и добавляющего новые функции через методы, аналогичные методам декорируемого объекта. Применительно к приложениям, например, для обработки изображений или текстовых документов, декораторы, такие как фильтры для изображений или форматирование текста, могут добавлять новые функции без изменения исходного объекта. Это делает код более гибким и легко поддерживаемым, позволяя создавать уникальные комбинации функциональности через множество декораторов для объекта.