**Введение в системное программирование**

Системой называется отношение двух множеств. множество элементов и множество связей между этими элементами.

Программы называется набор инструкций, понятных для конечного исполнителя.

Программный код - это это зашифрованная версия программы; То есть набор инструкций, написанный на языке, понятном постановщику задачи.

Транслятор называется средство, которое переводит инструкции с языка постановщика на язык исполнителя.

Компилятор переводит все инструкции сразу, А интерпретатор осуществляет построчный перевод.

Программирование - Это процесс написания программного кода.

Особенности работы программиста заключаются в специфическом стиле мышления, Который необходим для написания программ.

Системные программисты пишут программные коды для функционирования системы

Операционная система Выступает посредником между пользователем и аппаратной частью компьютера.

Следовательно свойства операционной системы можно разделить на две группы:

* Машинно-зависимые свойства. Критичны по отношению к архитектуре
  + Управление памятью
  + управление аппаратными прерываниями
  + управление вводом выводом
  + выделение ресурсов процессам
  + и т.п.
* Машинно-независимые свойства
  + Управление потоками данных
  + защита данных
  + планирование заданий
  + и т.п.

Качественная операционная система должна также обладать следующим набором свойств:

* Надежность. Минимальное количество сбоев, А также наличие механизмов диагностирования ошибок.
* Защищённость. Система должна защищать процессы от негативного влияния друг на друга (а также от процессов извне).
* Предсказуемость. Поведение системы можно спрогнозировать. Выполнение типовых операций должно приводить к типовому результату
* Удобство. Выполнение типовых операций должно затрачивать оптимальное количество ресурсов пользователя
* Эффективность. Выполнение типовых операций должно затрачивать оптимальное количество ресурсов системы
* Оптимальное количество сервисов. Возможности системы должны покрывать базовые потребности среднестатистического пользователя.
* Гибкость. Параметры системы могут быть адаптированы под потребности конкретного пользователя.
* Расширяемость. Наращивание функционала системы мы без ущерба для базового функционала
* Ясность. Механизмы работы системы должны быть доступны для восприятия пользователем.

**Основные типы операционных систем**

1. Системы пакетной обработки данных
   * Пакет это последовательность команд, Объединённых в одну логическую структуру.
   * Оператор загружает пакет данных в систему, система обрабатывает этот пакет данных И выдает новый пакет с результатом
   * Недостатком такой системы является невозможность пользователя взаимодействовать с ней во время вычислений. Поэтому невозможно различить ситуацию зависания программы и работы программы (Отследить Прогресс выполнения).
   * Другим недостатком является невозможность распараллеливания операций
2. Системы реального времени. Создают имитацию параллельности выполнения процессов.
   * Здесь на выполнение каждого процесса отводится определенный квант времени.Затем инициируется аппаратное прерывание и и начинается отсчет нового Кванта
3. Многопроцессорные системы. Сочетают в себе несколько логических или физических процессов.
   * Каждый процессор может работать как в режиме реального времени, так и в режиме пакетной обработки.
4. Система мульти программирования. Позволяет оптимизировать затраты системных ресурсов даже в 1 поточном режим
5. Сетевые ос используют удалённый сервер для обработки данных.

**Общие принципы взаимодействия пользователя с операционной системой**

Взаимодействие осуществляется через интерфейс пользователя (это часть операционной системы).

Например выделяют графический или консольный интерфейс.

Интерфейс Может видоизменяться в зависимости от категории пользователей.

**Назначение и использование современных ПК**

Назначение компьютера - это автоматизация процессов, которые раньше выполняли с человеком.

Использование компьютера - Это работа с программным обеспечением

Программное обеспечение делится на три категории:

* Прикладное. Используется для решения прикладных задач.
* Системное. Используется для решения системных задач.
* Инструментальная. Используется для разработки программного обеспечения.

С этой позицией операционная система выступает совокупностью прикладных и системных программ.

В целом операционная система представляет из себя множество процессов. Каждый процесс это отдельно взятая запущенная программа.

Программа представляет из себя набор инструкций. Инструкция разделяется на две части: Область данных и область управления.

В ходе работы процессы потребляют ресурсы. Или, другими словами, ресурс - это то, что Необходимо процессу для работы.

Ресурсы бывают делимые и неделимые.

Неделимые могут быть использованы только одним процессом. Делимые используется несколькими процессами одновременно или параллельно.

Основные ресурсы, необходимые для работы процессов можно посмотреть в Диспетчере задач.

***Потребление процессора.***

Метрикой, который отвечает за потребление процессора является процессор на и время. Процессорное время - это количество тактов, которые необходимо для завершения процесса.

Однако на практике данные характеристика не является самой существенной. Гораздо важнее отслеживать процент загрузки процессора (Это-то Какое количество процессорного времени потребляет каждый процесс от максимально возможного)

***Потребление оперативной памяти.***

Данный показатель измеряется в мегабайтах. С практической точки зрения важно именно количественное соотношение (Это связано с подходами к динамическому выделению памяти).

Также интерес Могут представлять потребление сетевых ресурсов, частота обращения к внешнему накопителю и др.

Основная задача оптимизации программного кода сводятся непосредственно к к уменьшению процессорного времени и количеству потребляемой памяти при сохранении функционала.

Парадигма мультипрограммирования Способствует оптимизации потребление ресурсов, что сокращает время выполнения программ.

В однопрограммном режиме все процессы выполняются последовательно друг за другом. таким образом суммарное время на выполнение данных процессов увеличивается.

в мультипрограммном режиме используется имитация параллельности работы (Пока один процесс загружает данные, другой выполняется. И пока к первый процесс выполняется, другой загружает данные).

Мультипрограммный подход сокращает совокупное время выполнения процессов, Однако каждый процесс По отдельности выполняется дольше.

Отсюда можно понять, что механизм выделения ресурсов процессам зависит от ситуации. За распределение ресурсов отвечает специальный компонент операционной системы.

Ресурс может быть выделен в следующих случаях:

* Если он свободен. И отсутствуют запросы к нему от процессов с более высоким приоритетом.
* Если этот ресурс допускает совместное использование
* Если он занят процессом с низшим приоритетом

Ресурсы бывают не только аппаратного, но и программного характера.

Например программные модули.

Программные модули могут быть также делимыми, неделимыми. Но их ограничение на неделимость достигается программным путём (Программные ресурсы априори делимые).

Поэтому их Обычно классифицируют по возможности повторного доступа. Они делятся на привилегированные, не привилегированные и реентерабельные.

Привилегированные модули выполняется без прерываний.

Непривилегированные модули могут быть прерваны в любой момент.

Рентабельные модули имеют определенные точки прерывания. Они занимаются в промежуточную позицию между привилегированными и непривилегированными.

**Оптимизация потребления ресурсов**

Ресурсы компьютера всегда конечные. Поэтому всегда стоит задача оптимизации их количества (Чтобы 1этаж задача выполнялась с использованием меньшего количества ресурсов).

Процессорное время может быть сэкономлено путем оптимизации алгоритмов.

Экономия памяти производится за счет уменьшения количества переменных. Однако в данном случае приходится увеличивать количество операций присваивания, что увеличивает процессорное время.

Поэтому в рамках одного процесса экономия процессорного времени и памяти являются взаимно обратными операциями.

Однако в рамках системы мы можно сэкономить общее количество потребляемой памяти. Это достигается за счет динамического выделения памяти.

Объём где я ими памяти под переменную зависит от её типа. статические переменные занимает оперативную память пока процесс не завершится.

**Указатель как тип данных**

Указатель представляет из себя переменную, значением которой является адрес другой переменной.

Под указатель тоже выделяется место в оперативной памяти. Количество байт = разрядности программы.

Переменные однозначно идентифицируется двумя параметрами: Это её адрес (Номер 1 ячейки памяти) и Количество занимаемых ячеек.

int a = 10;

int\* p = &a;//в качестве значения указателя выступает адрес переменной

К указателям применима операции сложения и вычитания. Они интерпретируются как сдвиги на определённое количество ячеек памяти. Шаг равен количество байтов, на которые ссылается указатель.

**Массивы и указатели**

Фактически массив является указателем на последовательность элементов определенного типа. Имя массива является указателем на первый элемент. А объём памяти равен суммарному объему Всех элементов с учетом их типа.

Пример:

#define N 10

int a[N];

int main(void)

{

system("chcp 1251>nul");

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

{

\*(a+i) = i \* i;

}

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

{

printf("%d ", \*(a+i));

}

return 0;

}

**Динамическое выделение памяти**

За динамическое выделение памяти отвечает функция malloc();

В качестве аргумента данная функция принимает количество байт, которые нужно выделить.

За освобождение памяти отвечает функция free(). в качестве аргумента функция принимает указатель:

int\* p = malloc(4);

\*p = 10;

printf("%d", \*p);

free(p);

Функция sizeof() принимает в качестве аргументов идентификатор типа, А возвращается количество байт, которое он занимает.

Пример создания динамического массива:

int n;

printf("введите размерность массива\n");

scanf("%d", &n);

int\* a=malloc(n\*sizeof(int));

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

{

a[i] = i \* i;

}

for (int i = 0; i < n+5; i++)

{

printf("%d ",a[i]);

}

free(a);

**Указатели и функции**

Указатели можно передавать как аргументы функции. также Функция может возвращать указатель. пример:

void f1(int\* a)

{

\*a = (\*a) + 10;

}

int main(void)

{

system("chcp 1251>nul");

int n = 10;

printf("%d\n", n);

f1(&n);

printf("%d\n", n);

return 0;

}

**Обобщенные указатели**

Обобщенные указатели могут ссылаться на любой тип данных. Однако для обращения указателю его необходимо в явном виде привезти к тому типу, на который он ссылается. пример:

int i = 100500;

char c = 'W';

float f = 123.456;

void\* p;

p = &i;

printf("%d\n", \*(int\*)p);

p = &c;

printf("%c\n", \*(char\*)p);

p = &f;

printf("%f\n", \*(float\*)p);

**Указатель на указатель**

Поскольку показатель также является переменной, то она тоже имеет адрес. соответственно этот адрес можно сохранить в значении другого указателя.

указатель на указатель объявляется с двумя звездочками.

Пример массива указателей:  
 int i = 100500;

int n = 123456;

int\* p[2] = {&i,&n};

printf(" %d %d", \*p[0], \*p[1]);

**Оператор переименования типов**

typedef int\* p\_int;

typedef p\_int\* pp\_int;

int main(void)

{

system("chcp 1251>nul");

int i = 100500;

p\_int p = &i;

pp\_int pp = &p;

printf("%d %d %d", i,\*p,\*\*pp);

return 0;

}

**Указатели и двумерные массивы**

Двумерный массив определяется как линейный массив из указателей, каждый из которых также является линейным массивом.

int main(void)

{

system("chcp 1251>nul");

printf("введите размерность массива ");

int n;

scanf("%d", &n);//ввели размерность массива

int\*\* pp = malloc(n \* sizeof(int));//выделилил память под линейный массив указателей

int\*\* pp1 = pp;//сохраняем "голову" в двумерном массиве (запомнили, где начало массива)

int\* p;

//алгоритм заполнения динамического двумерного массива

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

{

\*pp = malloc((i + 1) \* sizeof(int));

p = \*pp;//запомнили положение головы во вложенном массиве

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

{

\*\*pp = j;

(\*pp)++;

}

\*pp = p;//вернули положение головы обратно

pp++;

}

pp = pp1;//вернули "голову" на место

//алгоритм вывода динамического двумерного массива

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

{

p = \*pp;//запомнили положение головы во вложенном массиве

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

{

printf("%d ", \*\*pp);

(\*pp)++;

}

printf("\n");

\*pp = p;//вернули положение головы обратно

pp++;

}

pp = pp1;//вернули "голову" на место

//очищаем выделенную под массив память

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

{

free(pp[i]);// очистили внутренние массивы

}

free(pp);//очистили внешний массив

return 0;

}

тот же код, только через индексаторы:

int main(void)

{

system("chcp 1251>nul");

printf("введите размерность массива ");

int n;

scanf("%d", &n);//ввели размерность массива

int\*\* pp = malloc(n \* sizeof(int));//выделилил память под линейный массив указателей

//алгоритм заполнения динамического двумерного массива

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

{

pp[i] = malloc((i + 1) \* sizeof(int));

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

{

pp[i][j] = j;

}

}

//алгоритм вывода динамического двумерного массива

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

{

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

{

printf("%d ", pp[i][j]);

}

printf("\n");

}

//очищаем выделенную под массив память

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

{

free(pp[i]);// очистили внутренние массивы

}

free(pp);//очистили внешний массив

return 0;

}

**Указатель на функцию**

Под функцию, ю.в. также как и под переменную ю.в. выделяется память. соответственно можно создать указатель на ту область памяти, в которой находится эта функция.

По аналогии с массивом, имя функции является указателем на первую ячейку памяти, в которой она находится:

int summ(int a, int b)

{

return a + b;

}

int razn(int a, int b)

{

return a - b;

}

int main(void)

{

system("chcp 1251>nul");

int (\*f)(int, int);//указатель на функцию с двумя аргументами типа int и возвращающую int

f = razn;

printf("%d", f(2, 3));

return 0;

}

Передача указателя на функцию в качестве аргумента другой функции:

int kv(int a)

{

return a\*a;

}

int kub(int a)

{

return a\*a\*a;

}

int chto\_to(int (\*f)(int), int n)

{

return f(n);

}

int main(void)

{

system("chcp 1251>nul");

printf("1 - квадрат, 2 - куб ");

int (\*f)(int);

int i=0, n=0;

scanf("%d %d", &i, &n);

switch (i)

{

case 1:

f = kv;

break;

case 2:

f = kub;

break;

default:

f = kub;

break;

}

printf("\n%d", chto\_to(f,n));

return 0;

}

Указатель как аргумент функции:

void kv(int\* a)

{

\*a\*=\*a;

}

void kub(int\* a)

{

\*a\*=\*a\*\*a;

}

int main(void)

{

system("chcp 1251>nul");

int a = 2;

kub(&a);

printf("%d", a);

return 0;

}

Функция которая принимает и возвращает указатели:

char\* concat(char\* c1, char\* c2)//функция конкатенации строк

{

int l1 = 0, l2 = 0;

while (c1[l1]!='\0')//определяем длину первой строки

{

l1++;

}

while (c2[l2] != '\0')//определяем длину второй строки

{

l2++;

}

char\* c = malloc(l1+l2);//выделяем память под результирующую строку

for (int i = 0; i < l1; i++)//заносим посимвольно первую строку

{

c[i] = c1[i];

}

for (int i = 0; i < l2; i++)//заносим посимвольно вторую строку

{

c[i+l1] = c2[i];

}

c[l1 + l2] = '\0';//добавляем символ окончания строки

return c;

}

int main(void)

{

system("chcp 1251>nul");

char c1[] = "Hello ";

char c2[] = { 'w','o','r','l','d','\0' };

char\* c = concat(c1, c2);

printf("%s", c);

return 0;

}

**Указатели на структуры**

Структура также статический располагается в памяти, поэтому на неё тоже можно создать указатель.

Обращение к полю структуры через указатель осуществляется с помощью оператора Стрелка ->

struct MyStruct

{

int a;

char c;

};

int main(void)

{

system("chcp 1251>nul");

struct MyStruct m = { 10,'n' };

struct MyStruct\* p = &m;

printf("%d %c %d %c", m.a, m.c, (\*p).a, p->c);

return 0;

}

**Динамические структуры данных**

Большинство данных в компьютере Можно представить в виде списка из определенных структур.

Причём хранение осуществляется именно в виде списков, а не в виде массивов, поскольку массивы имеют ряд недостатков:

* Необходимо выделять большое количество последовательно идущих секторов памяти.
* Это большое количество операций переприсваивания при при работе с элементами в середине массива.

Для того чтобы избежать эти недостатки и используются динамические структуры данных

Линейный односвязный список

Это самая простая из существующих динамических структур.

По сути Каждый элемент списка является структурой, где часть полей отвечает за данные, а часть полей за адреса таких же элементов списка.

В односвязном списке поле с адресом только одно.

пример использования односвязных списков:

#include <stdio.h>

struct MyStruct

{

int a;

struct MyStruct\* next;

};

typedef struct MyStruct s;

s\* create(int);

void show(s\*);

void delete(s\*);

s\* insert(s\*, s, int);

int main(void)

{

system("chcp 1251>nul");

s\* list1 = create(10);

show(list1);

s item = { 25,NULL };

list1 = insert(list1, item, 4);

show(list1);

delete(list1);

return 0;

}

s\* create(int n)

{

s\* start = malloc(sizeof(s));//создаем первый элемент

start->a = 1;

start->next = NULL;

s\* p, \* q;//указатели на предыдущий и следующий элемент ЛОС

p = start;

for (size\_t i = 0; i < n-1; i++)//создаем в цикле все остальные элементы

{

q = malloc(sizeof(s));//инициализируем следующий элемент

q->a = p->a + 1;

p->next = q;//поле с указателем предыдущего элемента содержит адрес следующего

p = q;//предыдущий элемент стал следующим

}

p->next = NULL;

return start;

}

void show(s\* list)

{

while (list)//пока list != NULL

{

printf("%d ", list->a);

list = list->next;

}

printf("\n");

}

void delete(s\* list)

{

s\* p = list;

while (p)

{

p = list->next;//запомнили следующий

free(list);//удалили предыдущий

list = p;//следующий стал первым

}

}

s\* insert(s\* list, s item, int k)

{

s\* el = malloc(sizeof(s));//выделяем память под новый элемент списка

el->a = item.a;//помещаем в него поле из структуры

if (k == 1)//если мы меняем первый элемент

{

el->next = list;

list = el;

}

else // если меняем не первый элемент

{

s\* start = list;//запоминаем голову списка

for (size\_t i = 0; i < k - 1; i++)

{

list = list->next;//сдвигаем на k позиций

if (!(list->next))

{

printf("вы ввели индекс, превышающий размеры списка. МЫ вставим элемент в конец списка\n");

break;

}

}

void\* temp = list->next;//обмен адресами

list->next = el;

el->next = temp;

list = start;//возвращаем голову на место

}

return list;

}

**Многофайловые проекты**

Серьёзный программные продукты имеют достаточно большое количество строчек кода. Если весь программный код помещается в одном файле, то его восприятие становится затруднительным.

Поэтому логически завершенные фрагменты программного кода рекомендуется помещать в отдельные файлы.

В целом из фрагментов программы помещается целесообразно объявления глобальных переменных и отдельно взятой функции.

Для функций, описанных в другом файле, перед вызовом желательно указать их прототип (В том файле, откуда осуществляется вызов).

Для использования глобальной переменной из другого файла она также должна быть описана на этом файле, откуда осуществляется вызов. Для выстраивания связей между переменными в исходном файле Объявленная переменная должна быть помечена Зарезервирован словом extern.

Но в любом случае в каждом файле должны быть указаны прототипы функций, объявление структур и описание глобальных переменных. А также операторы переопределения типов.

При описании необходимо учитывать следующее:

* В глобальной области памяти допускается повторное определение, но не допускается повторная инициализация
* Внутри функций допускается повторная инициализация, но не допускается повторное определение

В связи с этим имеет смысл вынести все повторно используемые фрагменты кода да в отдельный файл и подключать его по мере необходимости.

Такие файлы называются файлами заголовков.

Файлы заголовков имеет расширение .h И добавляется в проекте стандартным способом (с помощью #include).

Пользовательские файлы заголовков указывается в двойных кавычках.

Файлы заголовков рекомендуется использовать для объявления объектов и не рекомендуется для их инициализации.

**Директивы препроцессора**

Препроцессор - это специальная программа, которая осуществляет алгоритмические действия перед компиляцией основного кода.

Команды для препроцессора называются директивами.

Директивы начинаются с символа #, В конце строки директивой “;” можно не ставить.

Список основных директив:

* #include. Вставляет содержимое из текстового файла в то место, где она написана
* #define. Имеет три основных применения:
  + Инициализация параметров (Задание флагов). Используется преимущественно для условной компиляции
  + Задание констант. Используется в качестве альтернативы глобальным переменным
  + Задание макроопределений (макросов)
* #undef. Отменяют задание параметра
* #ifdef - Условие компиляции, если определённый параметр задан
* #ifndef - Условие компиляции, если определённый параметр не задан
* #if - Инициализация условной компиляции. далее необходимо ввести условие, используя другие Директивы препроцессора.
* #elif - Директивы для создания вложенных условий.
* #else - Ветка при ложности всех условий. Её нельзя ставить выше чем #elif
* #error - Внесение искусственной ошибки для компиляции

**Разработка программ в системе Windows**

Операционная система Windows может предоставлять интерфейс для выполнения определённых системных задач сторонними программами. Такой интерфейс называется WinAPI.

WinAPI Представляет из себя набор функций, структур и различных параметров.

Для начала необходимо ознакомиться с типами данных, которые используется системой.

**Типы данных в windows**

* Тип BYTE обозначает 8-разрядное беззнаковое символьное значение.
* Тип WORD — 16-разрядное беззнаковое короткое целое.
* Тип DWORD — беззнаковое длинное целое.
* Тип UINT — беззнаковое 32-разрядное целое.
* Тип LONG эквивалентен типу long.
* Тип BOOL обозначает целое и используется, когда значение может быть либо истинным, либо ложным.
* Тип LPSTR определяет указатель на строку.
* Тип LPCSTR определяет константный (const) указатель на строку.
* Тип HANDLE обозначает 32-разрядное целое, используемое в качестве дескриптора.

Дескриптор выступает в качестве идентификатора определенного ресурса

**Функция запуска приложений Windows**

Функция называется WinMail. Принимает 4 параметра. они описаны ниже

int WINAPI WinMain(HINSTANCE hlnstance. // дескриптор, присваиваемый запущенному приложению

HINSTANCE hPrevInstance, // для совместимости с winl6. в Win32 не используется

LPSTR lpCmdLine. // указатель на командною строку, если приложение так запущено

int nCmdShow); // значение, которое может быть передано в функцию Show Window ()

Перед запуском приложений Windows необходимо особым образом настроить проект

Также при работе с API системы Windows необходимо подключать заголовочный файл windows.h

В данном случае создается системный процесс, который не имеет интерфейса.

Для ввода информации обычно используются файлы или другие способы (буфер обмена, именованный канал и т.п.)

Вывод информации также осуществляется в файл или другой источник данных.

Также вывод может осуществляться в диалоговое окно вывода.

За это отвечает функция MessageBox. Она принимает 4 параметра:

1. Дескриптор окна, который вызывает данный messagebox. Поскольку мы не работаем с оконными приложениями здесь ставим NULL
2. Непосредственно текст сообщения. Если для вывода используется текст в формате юникода, да то перед строкой ставятся префикс L
3. Заголовок окна messagebox . Это тоже может быть строка в формате ascii или Unicode
4. Набор параметров. Набор кнопок. Например кнопки OK отмена И другие. Также в качестве параметра можно вставить иконку

**Файловый ввод и вывод информации в языке си**

Файловый ввод-вывод Находится также в заголовочном файле <stdio.h>

Типовыми функциями для работы с файлами являются следующие:

* Создание указателя на файл (Открытие файла)
* закрытие файла
* вывод информации из файла (чтение)
* Запись информации в файл

Для открытия файла в системе C необходимо объявить переменную типа файл (Точнее указатель на файловый поток)

Зам открытии файла отвечает функция fopen.

Данная функция Принимает два параметра:

1. Имя файла (путь к нему)
2. Режим доступа к файлу (имеется в виду чтение или запись)

Список режимов:

| r | Чтение. Файл должен существовать. |
| --- | --- |
| w | Запись нового файла. Если файл с таким именем уже существует, то его содержимое будет потеряно. |
| a | Запись в конец файла. Операции позиционирования (fseek, fsetpos, frewind) игнорируются. Файл создаётся, если не существовал. |
| r+ | Чтение и обновление. Можно как читать, так и писать. Файл должен существовать. |
| w+ | Запись и обновление. Создаётся новый файл. Если файл с таким именем уже существует, то его содержимое будет потеряно. Можно как писать, так и читать. |
| a+ | Запись в конец и обновление. Операции позиционирования работают только для чтения, для записи игнорируются. Если файл не существовал, то будет создан новый. |

Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например “rb”, “wb”, “ab”, или, для смешанного режима “ab+”, “wb+”, “ab+”. Вместо b можно добавлять букву t, тогда файл будет открываться в текстовом режиме. Это зависит от реализации. В новом стандарте си (2011) буква x означает, что функция fopen должна завершиться с ошибкой, если файл уже существует. Дополним нашу старую программу: заново откроем файл и считаем, что мы туда записали.

**Работа с файлами через WinAPI**

За создание дескриптора файла отвечает функция Createfile

HANDLE CreateFile(

LPCTSTR lpFileName, // Указатель на имя файла (устройства)

DWORD dwDesiredAccess, //Параметры доступа

DWORD dwShareMode, //Разделяемый доступ

LPSECURITY\_ATTRIBUTES lpSecurityAttributes, //безопасность

DWORD dwCreationDistribution,// Описание

DWORD dwFlagsAndAttributes, // Атрибуты файла

HANDLE hTemplateFile // Файл шаблона

);

конкретный пример открытия файла:

//создаем текстовый файл

HANDLE hFile = CreateFile(PATH,//путь к файлу

GENERIC\_READ | GENERIC\_WRITE,//флаги на открытие как на чтение, так и на запись

FILE\_SHARE\_READ,//совместный доступ только на чтение

NULL,//структура безопасности по умолчанию

OPEN\_ALWAYS,//режим создания файла (открыть, перезаписать и т.п.)

FILE\_ATTRIBUTE\_NORMAL,//атрибуты файла по умолчанию

NULL);//шаблон файла отсутствует

Запись данных в файл:

## BOOL WriteFile(HANDLE hFile, //собственно указатель на файл

## LPVOID lpBuffer, // указатель на буфер - откуда записываем данные в файл

## DWORD nNumberOfBytesToWrite, //объем записываемых данных

## LPDWORD lpNumberOfBytesWrite, //фактический размер записанных данных

## LPOVERLAPPED lpOverlapped // флаг режима доступа к файлу: асинхронный(FILE\_FLAG\_OVERLAPPED)

## //или синхронный(NULL)

конкретный пример записи в файл:

LPCSTR MyString = "Hello world";//буфер для записи (что записываем)

DWORD d = 0;

//функция записи в файл

WriteFile(hFile,//дескриптор открытого файла

MyString,//указываем буфер для записи

strlen(MyString), //указываем, сколько байт мы хотим записать

&d,//передаем указатель на DWORD

NULL//синхронный режим записи

);

Функция для чтения данных из файла:

За это отвечает функция ReadFile. Аргументы у неё такие же как его функции WriteFile,

Но при чтении есть некоторые особенности:

* Надо указывать заведомо большой буфер для чтения (Так как мы не знаем Сколько информации находится в файле)
* После прочтения строку нужно закрыть (Дописать к ней символ \0)

пример кода:

DWORD d = 0;//сколько фактически байт было прочитано

DWORD sizeBuffer = 521;//объем буфера

LPSTR str = malloc(sizeBuffer+1);//куда считывать

ReadFile(hFile, str, sizeBuffer, &d, NULL);

str[d] = '\0';

Пример со структурой Overlapped:

OVERLAPPED olf = { 0 }; //Структура, в которой задана позиция в файле

DWORD sizeBuffer = 512;//объем буфера

LPSTR str = malloc(sizeBuffer+1);//куда считывать

ReadFile(hFile, str, sizeBuffer, &d, &olf);

free(str);

olf.Offset = 0;//задаем смещение (позицию в файле)

LPSTR str1 = malloc(d + 1);

ReadFile(hFile, str1, d, &d1, &olf);

str1[d1] = '\0';

**Использование динамических загружаемых библиотек**

По мере усложнения проекта количество используемых функций возрастает.

И не всегда использование Многофайловых проектов решает эту проблему.

Помимо всего прочего использование множества функций в одном проекте влечет за собой следующие проблемы:

* Не все функции, описанные в проекте единовременно могут быть использованы, но они занимают пространство оперативной памяти и это сказывается на объёме исполняемого файла.
* Если одни и те же функции необходимо использовать в разных проектах, то код приходится дублировать. Причём не всегда копирование кода решает эту проблему.

Данные проблемы Как раз-таки решается за счет подключения динамических библиотек.

Есть и статические библиотеки (lib), но на сегодняшний момент их стараются не использовать.

Динамические библиотеки подключаются к проекту и отключаются в тот момент времени, когда это надо. Таким образом достигается экономия оперативной памяти.

*Создание библиотеки DLL в winapi*

Для создания библиотеки DLL нужно добавить соответствующий проект в решение И настроить его во на запуск как динамическую библиотеку.

Проект DLL имеет некоторые особенности.

Он настраивается также, как и проект winapi, Ну помимо этого ещё необходимо указать В настройках проекта на вкладке общие Тип конфигурации динамическая библиотека.

Точка входа у библиотеки DLL тоже своя особенная:

BOOL WINAPI DllMain(HINSTANCE hlnstDll, DWORD dwReason, LPVOID IpReserved)

{

BOOL bAllWentWell = TRUE;

switch (dwReason)

{

case DLL\_PROCESS\_ATTACH:

break;

case DLL\_THREAD\_ATTACH:

break;

case DLL\_THREAD\_DETACH:

break;

case DLL\_PROCESS\_DETACH:

break;

}

if (bAllWentWell)

return TRUE;

else

return FALSE;

}

Всем экспортируемые из DLL функции. Должны иметь специальные соглашения о вызовах \_cdecl. (Дело в том что приложение winapi по умолчанию имеет соглашения вызовах \_\_stdcall).

Также же функции необходимо пометить с помощью специального оператора:

\_\_declspec(dllimport) - для Импортируемых функций

\_\_declspec(dllexport) - для экспортируемых функций.

И это всё описывается в портативе функции. Без прототипа мы её импортировать или экспортировать не сможем (Но просто писать функции в dll мы можем).

Пример экспортируемой функции:

\_\_declspec(dllexport) int Hello(LPWSTR str);

int Hello(LPWSTR str)

{

MessageBox(NULL, str, L"Проверка связи", MB\_OK);

return 0;

}

*Импорт функции из DLL в основную программу*

Для этого используются три основных шага:

1. Подключается DLL. (Создается дескриптор данной библиотеки)
2. Импортируется функция из этой библиотеки
3. Освобождается память под дескриптор (Отключается библиотека)

Для подключения библиотеки используется функция LoadLibrary. В качестве аргумента этой функции передается путь к DLL. Возвращает она дескриптор HINSTANCE

Однако если эту библиотеку загрузить не удалось, то функция возвращает NULL

За Отключение библиотеки отвечает функция FreeLibrary. В качестве аргумента она получает дескриптор библиотеки.

Для импорта функции необходимо сначала создать указатель на функцию с сигнатурой, Которую мы хотим вызвать.

Также мы должны указать соглашение о вызовах.

typedef int(\_\_cdecl\* MyFunction)(LPWSTR);

Оператор typedef в данном случае объявляет пользовательский тип, который позволяет создать указатель на эту функцию.

За инициализацию функции Из DLL Отвечает функция GetProcAddress. Она принимает два параметра. это дескриптор DLL и строку, содержащую имя импортируемой функции. Возвращает она естественно указатель на эту функцию.

пример вызова функции из DLL:

#include <windows.h>

#define PATH L"CodeDll.dll"

typedef int(\_cdecl\* MyFunction)(LPWSTR);

int WINAPI WinMain(HINSTANCE hInstance, HINSTANCE hPrevInstance,

LPSTR lpCmdLine, int nCmdShow)

{

HINSTANCE MyDLL;

if (!(MyDLL = LoadLibrary(PATH))) return 1;//подключение DLL

MyFunction MyF1; //создали переменную типа указатель на вызываемую функцию

MyF1 = (MyFunction)GetProcAddress(MyDLL, "Hello");//инициализация указателя на функцию

MyF1(L"Привет, мир");

FreeLibrary(MyDLL);//отключение DLL

return 0;

}

Не забывайте, что при любом изменении в исходном коде DLL, проект нужно пересобрать (иначе сама библиотека не перекомпилируется);

**Использование DLL в языке си#**

Языках высокого уровня также используется DLL.

*DLL библиотеки Также можно разрабатывать Средствами языка си#*.

Алгоритм примерно тот же самый:

* В уже существующий проект на си# Надо добавить ещё один проект dll
* Поставить ссылку на проект с DLL
* Там где, мы хотим использовать функционал из dll Подключить соответствующее пространство имён.
* Теперь можно использовать классы и методы из dll

Это работает Как для статических классов, также для не статических классов и для вложенных пространств имён.

пример кода:

код DLL:

namespace ClassLibrary1

{

public class Class1

{

public int summ(int a,int b)

{ return a + b; }

}

namespace SubClassLibrary1

{

public static class SubClass1

{

public static int razn(int a, int b)

{ return a - b; }

}

}

}

Код основной программы:

using ClassLibrary1;

using ClassLibrary1.SubClassLibrary1;

namespace ConsoleApp4

{

internal class Program

{

static void Main(string[] args)

{

Class1 class1 = new Class1();//нестатический класс из DLL

Console.WriteLine(class1.summ(3, 6));

теперь Console.WriteLine(SubClass1.razn(3, 6));//статический класс из DLL

Console.ReadKey();

}

}

}

*Подключение к проекту c# Библиотеки от winapi*

Иногда выгоднее подключать низкоуровневые функции к работе вашей программы.

Преимущества этого решения - скорость выполнения самой программы.

минус данного решения В том, что кот становится неуправляемым. То есть встроенные средства языка си# Не смогут проанализировать этот код Ну и например предотвратить утечку памяти, если она допускается функциями из библиотеки.

Алгоритм подключения системной библиотеки примерно такой же, Как и на языке си, но только используя синтаксис си#.

* Объявить прототип функции. Для этого нужно:
  + подключить пространство имён: using System.Runtime.InteropServices;
  + Использовать атрибут [DllImport]. У этого атрибута есть несколько параметров, Они передаются внутри скобок ( как у метода)
  + Нужно Правильно указать соглашение о вызовах в атрибуте [DllImport] импорт. Дело в том что по умолчанию считается \_\_stdcall, А у нас в библиотеке \_\_cdecl
  + Далее описать сигнатуру вызываемой функции dll. Только нужно заменить типы данных ( те, те которые используются в Windows, на те, как которые Используются в языке си #)

пример кода.

для DLL (WinAPI):

#include <Windows.h>

BOOL WINAPI DllMain(HINSTANCE hlnstDll, DWORD dwReason, LPVOID IpReserved)

{

BOOL bAllWentWell = TRUE;

switch (dwReason)

{

case DLL\_PROCESS\_ATTACH:

break;

case DLL\_THREAD\_ATTACH:

break;

case DLL\_THREAD\_DETACH:

break;

case DLL\_PROCESS\_DETACH:

break;

}

if (bAllWentWell)

return TRUE;

else

return FALSE;

}

\_\_declspec(dllexport) int MyFunc(LPWSTR str);

int MyFunc(LPWSTR str)

{

MessageBox(NULL, str, L"Я сделяль", MB\_OK);

return 0;

}

\_\_declspec(dllexport) int Summ(int a, int b);

int Summ(int a, int b)

{

return a + b;

}

для c#:

using System.Runtime.InteropServices;//пространство имен для импорта функций из DLL

namespace ConsoleApp4

{

internal class Program

{

[DllImport(@"D:\VSProject\UsingDLLWinAPI\Debug\DLLCode.dll",CallingConvention=CallingConvention.Cdecl)]//CallingConvention - это соглашение о вызовах

public static extern int MyFunc(byte[] str);//подключил функцию messagebox из DLL

[DllImport(@"D:\VSProject\UsingDLLWinAPI\Debug\DLLCode.dll", CallingConvention = CallingConvention.Cdecl)]

public static extern int Summ(int a, int b);//подключил функцию суммы из DLL

static void Main(string[] args)

{

byte[] b = Encoding.Unicode.GetBytes("Мой текст");

MyFunc(b);

int s = Summ(2, 6);

Console.ReadKey();

}

}

}

**Работа с многозадачностью**

В основе парадигмы программирования лежит понятие процесс.

Процесс можно понимать как некий объект для исполняемого файла. один и тот же исполняемый файл может быть запущен несколько раз (при этом создаётся несколько процессов).

Вся информация о процессе хранится в оперативной памяти.

Процесс характеризуется неким набором информации. За работу процессов отвечают потоки. Поток рассматривается как последовательный набор инструкций, Который выполняется на процессоре.Каждый процесс имеет хотя бы один поток. Данный поток называется основным

Процессы могут находиться в одном из следующих состояний:

* Выполняется. Основному потоку выделено процессорное время. Количество выполняемых процессов одновременно и не может быть больше, чем Потоков, который поддерживает процессор
* Состояние готовности к выполнению. Процессу Предоставлены все ресурсы, кроме процессорного времени.
* Состояние ожидания. Процессу предоставлены не все ресурсы (Например идёт ввод или вывод данных).

За время своего существования один и тот же процесс может многократно менять свои состояния.

Место процесса в очереди определяется его приоритетом.

Приоритет процесса может поменять приоритет (в том числе и создаться) в одном из следующих случаев:

* По команде пользователя.
* При выборе из очереди планировщиком операционной системы
* По таймеру системному
* По инициативе другого процесса

При создании процесса в него могут быть переданы аргументы.

Для этого необходимо правильным образом описать функцию запуска

int main(int argc, char\* argv[])

{

setlocale(LC\_ALL, "RUS");

for (size\_t i = 0; i < argc; i++)

{

std::cout << argv[i] << "\n";

}

}

Аргументы представляет из себя массив строковых значений. В примере показано, как их вывести на экран.

Первым аргументом ( с индексом 0) является имя исполняемого файла.

При создании процесса создаётся также его дескриптор, который является структурой, содержащий всю необходимую информацию о процессе.

Основной поток в процессе может также порождать и вспомогательные потоки.

Они нужны для параллельного выполнения операций. Желательно, чтобы данные, которые используют различные потоки не взаимодействовали между собой.

**Для создания потоков в winapi используется функция**

CreateThread (  
NULL, //указатель на структуру безопасности (NULL по умолчанию)

0, // размер стека (0 по умолчанию)

func, //указатель на функцию, которая будет выполняться в потоке

NULL, // указатель на аргумент функции потока (NULL - пустой указатель, без аргументов)

0, // флаги создания потока. 0 - по умолчанию.

0)//ID потока (0 - автоматически).

Функция потока должна иметь следующую сигнатуру:

DWORD WINAPI func(LPVOID param);

Она должна возвращать значение dword, Иметь в качестве аргумента указатель общего типа.

**Примеры задач с использованием потоков**

**Задача:** Посчитать в одном потоке факториал числа, а в другом последовательность Фибоначчи.

В основном потоке мы создаем 2 вспомогательных С использованием функции CreateThread.

пример:

HANDLE hF[2];

hF[0] = CreateThread(NULL, 0, TreadFactr, NULL, 0, 0);

hF[1] = CreateThread(NULL, 0, TreadFib, NULL, 0, 0);

Данная функция возвращает дескриптор. мы используем Один массив для всех дескрипторов создаваемых потоков (Это нужно для применения функции WaitForMultipleObjects, Который в качестве аргумента принимают указатель на дескрипторы).

Структура функции WaitForMultipleObjects

(count, //Количество потоков, завершения которых необходимо ждать

hF,//Указатель на дескрипторы (Массив дескрипторов)

TRUE, //Флаг ожидания. Истина - будет ждать завершения всех потоков из count, Ложь - ожидание Завершения одного любого потока.

INFINITE)//Время ожидания завершения потока.

коды функций потока:

DWORD WINAPI TreadFactr(LPVOID param)

{

int f = 0;

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

{

if (i == 0)

{

f = 1;

}

else

{

f \*= i;

}

printf("factorial %d raven %d \n", i, f);

}

ExitThread(0);

}

DWORD WINAPI TreadFib(LPVOID param)

{

int f = 0;

int f1 = 1;

int f2 = 1;

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

{

if (i > 1)

{

f = f1;

f1 = f2;

f2 += f;

}

printf("%d element Fibonachi raven %d \n", i, f2);

}

ExitThread(0);

}

Задача про управление одним потоком из другого потока:

HANDLE hF[2];

int work = 0;

VOID Upravlenie(VOID)

{

system("chcp 1251");

hF[0] = CreateThread(NULL, 0, TreadWorker, NULL, 0, 0);

hF[1] = CreateThread(NULL, 0, TreadManager, NULL, 0, 0);

WaitForMultipleObjects(2, hF, TRUE, INFINITE);

}

DWORD WINAPI TreadWorker(LPVOID param)

{//просто увеличивает значение счетчика

while (TRUE)

{

Sleep(100);

work++;

}

}

DWORD WINAPI TreadManager(LPVOID param)

{

int i;

while (TRUE)

{

printf("Выберите действие: \n 1-посмотреть значение счетчика \n 2-поставить рабочий поток на паузу \n 3-снять рабочий поток с паузы \n 4 - завершить все потоки \n ");

scanf("%d", &i);

switch (i)

{

case 1:

printf("Значение счетчика равно %d\n", work);

break;

case 2:

printf("Рабочий поток поставлен на паузу\n");

SuspendThread(hF[0]);

break;

case 3:

printf("Рабочий поток снят с паузы\n");

ResumeThread(hF[0]);

break;

case 4:

printf("Все потоки завершили работу\n");

TerminateThread(hF[0], 0);

ExitThread(0);

break;

default:

printf("Ничего не изменилось\n");

break;

}

}

}

Поток имеет свой счётчик прерываний (То есть если его поставить на паузу два раза, То есть снять с паузы надо тоже два раза). Пример:

int i = 0;

HANDLE hF[2];

VOID FactrSinh(VOID)

{

hF[0] = CreateThread(NULL, 0, ThreadF, NULL, 0, 0);

hF[1] = CreateThread(NULL, 0, ThreadVivod, NULL, 0, 0);

WaitForMultipleObjects(2, hF, TRUE, INFINITE);

}

DWORD WINAPI ThreadF(LPVOID param)

{

for (i = 0; i < 50; i++)

{

Sleep(500);

if (i >10 && i<=20) { SuspendThread(hF[1]); }

if (i>20) { ResumeThread(hF[1]); }

if (i == 49) { TerminateThread(hF[1],0); }

}

ResumeThread(hF[1]);

ExitThread(0);

}

DWORD WINAPI ThreadVivod(LPVOID param)

{

while (TRUE)

{

Sleep(100);

printf("index raven %d \n", i);

}

ExitThread(0);

}

**Синхронизация потоков**

В случае когда два или более потоков пытаются получить одновременный доступ к какому-либо общему ресурсу (Например К данным в оперативной памяти), Поведение программа может быть неверным. Значение может быть записано раньше времени или прочитано раньше времени.

Существует несколько способов синхронизации потоков, однако все они в той или иной мере способствуют запрещению параллельного доступа к общему ресурсу.

Рассмотрим принцип действия критической секции.

Критическая секция - это участок кода, да который фиксируется для выполнения только одним потоком (Набор инструкций, который может выполняться только одним потоком одновременно).

Критическая секция объявляется следующим образом.

* Создаётся Переменная соответствующего типа

CRITICAL\_SECTION section = { 0 };

Создавать её желательно в глобальной области видимости.

* До начала выполнения потока необходимо инициализировать критическую секцию с помощью соответствующей функции:

InitializeCriticalSection(&section);

* Участок кода, да который контролируется критической секцией должен находиться между инструкциями EnterCriticalSection(&section); и LeaveCriticalSection(&section);
* После окончание работы потоков можно освободить критическую секцию с помощью функции DeleteCriticalSection(&section);

Пример кода:

#define \_CRT\_SECURE\_NO\_WARNINGS

#include <Windows.h>

#include <stdio.h>

#include <malloc.h>

#define I\_WILL\_WAIT 10

CRITICAL\_SECTION section = { 0 }; //Критическая секция

VOID Crit(VOID);

DWORD WINAPI Thread1(DWORD param);

DWORD WINAPI Thread2(DWORD param);

int count = 0;

HANDLE h[3];

VOID Crit(VOID)

{

DWORD tmp1 = 1;

DWORD tmp2 = 2;

DWORD tmp3 = 3;

InitializeCriticalSection(&section);

h[0] = CreateThread(NULL, 0, Thread2, tmp1, 0, 0);

h[1] = CreateThread(NULL, 0, Thread2, tmp2, 0, 0);

h[2] = CreateThread(NULL, 0, Thread2, tmp3, 0, 0);

WaitForMultipleObjects(3, h, TRUE, INFINITE);

DeleteCriticalSection(&section);

printf("count = %d\n", count);

}

DWORD WINAPI Thread1(DWORD param)

{

for (int i=0;i<10;i++)

{

Sleep(I\_WILL\_WAIT);

count++;

printf("count = %d, potok = %d\n", count, param);

}

ExitThread(0);

}

DWORD WINAPI Thread2(DWORD param)

{

EnterCriticalSection(&section);

for (int i = 0; i < 10; i++)

{

Sleep(I\_WILL\_WAIT);

count++;

printf("count = %d, potok = %d\n", count, param);

}

LeaveCriticalSection(&section);

ExitThread(0);

}

**Технологии обмена данными между процессами**

На данный момент нам известны следующие способы:

* Использовать аргументы командной строки. Однако этот способ работает только для создания новых процессов. в уже запущенную программу таким способом данные мы передать не сможем
* Использовать файлы. Однако этот способ также не лишён недостатков. основным недостатком является скорость работы ( из-за долгого обращения к файловой системе)

В качестве решения проблемы можно рассматривать файл, который создан в оперативной памяти ( другими словами выделенный объем оперативной памяти, который может использоваться совместно разными процессами).

**Каналы передачи данных**

Канал (англ. Pipe) - Область виртуального пространства, которое может быть использовано для совместного доступа различными процессами.

Однако в силу этого канал не может храниться ze1 дельно от какого-то процесса (По факту дескриптор канала является глобальным указателем). Поэтому информация о нём будет очищена если завершить процесс, который его создал.

Будем называть процесс, Который создаёт канал с сервером, а процессы, которые подключаются к каналу - клиентами.

Каналы имеют несколько разновидностей:

* Симплексные или дуплексные
  + Симплексный это однонаправленные. Например сервер только записывает данные, а Клиенты только читают их. Или клиенты только пишут, а сервер только читает.
  + Дуплексные это когда и клиент и сервер может и читать и писать
* Бинарные или текстовые. По аналогии с бинарными или текстовыми файлами
* С общим или разделяемым доступом к содержимому
* Именованные или анонимные

Анонимные каналы.

В качестве их идентификатора используется дескрипторы на чтение или на запись.

Пример функции для создания анонимного канала

BOOL CreatePipe (PHANDLE pliRead. // переменная для дескриптора чтения (входной канал)

PHANDLE phWrite. // переменная для дескриптора записи (выходной канал)

LPSECURITY\_ATTRIBUTES Ipsa. // привилегии доступа

DWORD dwPipeSize); // размер буфера канала (0 по умолчанию)

Из недостатков такого способа следует отметить, что дескриптор канала сервер должен передать клиенту по какому-то другому пути.

Использование именованных каналов

Именованные каналы поем дескриптора имеют также имя в виде строки. Причём Если дескриптор каждый раз разный, то имя является константой.

Имя является сетевым.

Сервер создаёт именованный канал с помощью функции CreateNamedPipe():

Она имеет следующую сигнатуру:

HANDLE CreateNamedPipe (

LPTSTR IpszPipeName, // строка с именем нового канала (сетевое)

DWORD fdwOpenMode, // доступ (симплексный или дуплексный)

DWORD fdwPipeMode, // тип, режимы чтения и ожидания

DWORD dwMaxInstances, // максимальное число клиентов

DWORD dwOutBuf, // размер выходного буфера, байты

DWORD dwInBuf, / размер входного буфера/байты

DWORD dwTimeout, // время паузы, миллисекунды (ожидание подключения)

LPSECURITY ATTRIBUTES Ipsa); // структура безопасности

Проверить статус подключения клиента к серверу можно с помощью функции ConnectNamedPipe();

В качестве аргументов она принимает дескриптор канала и Структуру OVERLAPPED (для асинхронного доступа). Для синхронного доступа можно поставить NULL

Возвращает она логическое значение.

Для того чтобы установить подключение, на стороне клиента Должна быть вызвана функция SetNamedPipeHandleState();

Она имеет следующие аргументы:

* Дескриптор канала
* режим подключения
* максимальное количество пользователей
* Время ожидания

Пример:

BOOL isSuccess = SetNamedPipeHandleState(hNamedPipe,&dwMode,NULL,NULL);

Если клиент подключился к серверу, toobi функция возвращает значение True;

Если Клиент не подключен, то функция ConnectNamedPipe возвращает false;

если сервер не отвечает, то функция SetNamedPipeHandleState возвращает FALSE.

Далее вся логика функционирования имя нового канала настраивается исходя из контекста задачи.

В приведённом ниже случае клиент пишет сообщение серверу первым, затем ждёт ответного сообщения от сервера. и после получения этот процесс повторяется.

Полный код программы выглядят следующим образом:

для сервера:

int main()

{

system("chcp 1251");

HANDLE hNamedPipe;//объявление дескриптора калала

LPWSTR lpszPipeName = "\\\\.\\pipe\\MyPipe";//переменная, содержащая имя канала

DWORD size\_buffer = SIZE\_BUFFER;//размер буфера для чтения

LPSTR buffer = (CHAR\*)calloc(size\_buffer, sizeof(CHAR));//строковая переменная, в которую будут считаны данные

char message[SIZE\_BUFFER];

BOOL Connected;

DWORD actual\_readen; //сколько на самом деле было прочитано байт

BOOL SuccessRead;

DWORD d = 0;//переменная, в которой будет храниться значение числа, передаваемого от клиента

while (TRUE)

{

hNamedPipe = CreateNamedPipeA( //создание канала

lpszPipeName, //имя канала

PIPE\_ACCESS\_DUPLEX, //режим доступа к каналу (односторонний/двусторонний)

PIPE\_TYPE\_MESSAGE | PIPE\_READMODE\_MESSAGE | PIPE\_WAIT, //режимы работы канала: передавать сообщения|читать сообщения|ждать

PIPE\_UNLIMITED\_INSTANCES, //количество водящих соединений к каналу. в данном случае неограничено

SIZE\_BUFFER, // объем буфера на чтение (байт)

SIZE\_BUFFER, // объем буфера на запись (байт)

INFINITE, // максимальное время ожидания сообщения

NULL); //указатель на структуру безопасности

Connected = ConnectNamedPipe(hNamedPipe, NULL); //установка соединения клиента с каналом

if (Connected) //если клиент подключился

{

// printf("\nКлиент успешно подключился \n");

SuccessRead = ReadFile(hNamedPipe, buffer, size\_buffer, &actual\_readen, NULL);

if (SuccessRead)

{

printf("\nКлиент пишет: ");

printf(buffer);

printf("\n");

//отвечаем клиенту

printf("\nвведите сообщение для клиента:\n");

gets(message);

buffer = &message;//строковая переменная, значение которой записывается в канал

WriteFile(hNamedPipe, buffer, size\_buffer, &actual\_readen, NULL);

}

}

else

{

printf("\nКлиент отключился от сервера\n");

}

CloseHandle(hNamedPipe);//закрываем канал

}

}

для клиента:

int main()

{

system("chcp 1251");

LPSTR lpszPipeName = "\\\\.\\pipe\\MyPipe";//имя канала (такое же, как и на сервере)

BOOL flag\_otvet = TRUE;

char message[SIZE\_BUFFER];

DWORD size\_buffer = SIZE\_BUFFER;//размер буфера для записи

DWORD actual\_written; //сколько на самом деле было записано байт

LPSTR buffer;

DWORD actual\_readen;

BOOL SuccessRead;

while (TRUE)

{

char message[SIZE\_BUFFER];

HANDLE hNamedPipe = CreateFileA(//открываем канал. по аналогии с открытием файла

lpszPipeName, GENERIC\_READ | GENERIC\_WRITE,

0, NULL, OPEN\_EXISTING, 0, NULL);

DWORD dwMode = PIPE\_READMODE\_MESSAGE;

BOOL isSuccess = SetNamedPipeHandleState(hNamedPipe,&dwMode,NULL,NULL);

if (!isSuccess)

{

printf("сервер не отвечает\n");

flag\_otvet = TRUE;

}

else

{

// printf("соединение установлено\n");

if (flag\_otvet)

{

printf("введите сообщение для сервера:\n");

gets(message);

buffer = &message;//строковая переменная, значение которой записывается в канал

WriteFile(hNamedPipe, buffer, size\_buffer, &actual\_written, NULL);

flag\_otvet = FALSE;

}

buffer = (CHAR\*)calloc(size\_buffer, sizeof(CHAR));

SuccessRead = ReadFile(hNamedPipe, buffer, SIZE\_BUFFER, &actual\_readen, NULL);

if (SuccessRead)

{

printf("\nСервер пишет: ");

printf(buffer);

printf("\n");

flag\_otvet = TRUE;

}

}

Sleep(100);

CloseHandle(hNamedPipe);//закрываем подключение к каналу

}

}

**Использование системного буфера обмена**

Несмотря на все удобства, именованные каналы имеют ряд недостатков:

* Канал не может функционировать без сервера (Если программа сервер завершится, то канал закроется);
* Доступ к каналу нужно специально организовывать из программного кода.

Одним из решений данной проблемы является использование системного буфер обмена.

По сути буфер обмена можно рассматривать как канал или глобальный указатель, к которому имеют доступ все прикладные программы.

Однако при работе с буфером следует учитывать и его недостатки:

* Буфер обмена только один в системе. и любое приложение может его перезаписать. Поэтому нет гарантии целостности получения данных
* Информация не может быть приватной (Во сколько буфер обмена доступен всем)

Буфер обмена на позволяет хранить данные различных типов. Однако при работе с ним программы учитывают этот контекст.

В зависимости от формата данных в буфере с ним могут взаимодействовать те или иные приложения.

Работа с буфером обмена

Буфер обмена нужно скорее воспринимать как указатель, а не как переменную.

У буфера нет своей заранее выделенной области. Поэтому память выделяется в самой программе.

Примеры функций для работы с буфером:

int ClipboardInputText(LPWSTR buffer)//записать строку в системный буфер

{

DWORD len;//длина сообщения

HANDLE hMem;//дескриптор глобальной области памяти

len = wcslen(buffer) + 1; // определение длины строки в формате юникода

hMem = GlobalAlloc(GMEM\_MOVEABLE, len \* sizeof(LPWSTR)); //выделение памяти в глобальной области видимости

memcpy(GlobalLock(hMem), buffer, len \* sizeof(LPWSTR));// копирование области памяти из buffer в hMem

GlobalUnlock(hMem); //разблокировать содержимое этой памяти (сделать доступным для других программ)

OpenClipboard(0);//открыть буфер обмена

EmptyClipboard();//очистить буфер обмена

SetClipboardData(CF\_UNICODETEXT, hMem);//записать в буфер обмена данные соответствующего типа

CloseClipboard();//закрыть буфер обмена, сделать его доступным для других приложений

return 0;

}

int ClipboardOutputText()//считать информацию из системного буфера

{

OpenClipboard(NULL);//открыть буфер обмена

LPWSTR Mess = (LPWSTR)GetClipboardData(CF\_UNICODETEXT);//Считать из глобального участка памяти, привести это все к стороке

CloseClipboard();//закрыть буфер обмена, сделать его доступным для других приложений

MessageBox(NULL, Mess, L"Содержимое буффера обмена", MB\_OK);

return 0;

}

**Работа с системным реестром Windows**

Формально реестр представляет из себя базу данных.

Элементами этой базы данных являются параметры. Организация параметров представляет из себя иерархию.

Чаще всего реестр используется для хранения параметров операционной системы или прикладных программ.

Реестр задумывался как ниткой общее хранилище настроек. До этого программы хранили свои настройки в конфигурационных файлах (или файлах инициализации).

Использование конфигурационных файлов имела ряд недостатков:

* Не было централизованного хранилища этих файлов.
* Проблема защиты файлов:
  + Текстовые файлы можно было удалить
  + Можно было изменить содержимое текстовых файлов по-своему ведому

Для работы с реестром в системе Windows предусмотрена специальная утилита RegEdit.EXE

Сам реестр имеет 5 глобальных веток:

* HKEY CLASSES ROOT— хранится информация о зарегистрированных классах, расширениях документов;
* HKEY CURRENT USER — хранится информация о текущей пользовательской конфигурации, внешнем виде рабочего стола, сетевых настройках;
* HKEY LOCAL MACHINE — хранится информация о системной и аппаратной конфигурации;
* HKEY USERS — хранится информация обо всех зарегистрированных пользователях;
* HKEY\_CURRENT\_CONFIG — текущая аппаратная конфигурация.

Остальные параметры имеют более длинные пути, которые начинаются в одной из глобальных веток.

Крупные ветки называются ульями.

Более мелкие ветки называются ключами реестра.

Каждый ключ может содержать внутри себя другие ключи или параметры.

Параметры также имеют свои определенные типы:

* Строковый параметр. Содержит последовательность символов в определенной кодировке. Его идентификатор в системе REG\_SZ
* Двоичный параметр. По сути содержит массив байт. Его идентификатор в системе REG\_BINARY
* Машинное слово для 32-битных систем. Его идентификатор в системе REG\_DWORD
* Машинное слово для 64 разрядной системы. Его идентификатор в системе REG\_QWORD
* Мультистроковый параметр. другими словами массив строк. Его идентификатор в системе REG\_MULTI\_SZ
* Расширяемый строковый параметр. По сути представляет из себя строку переменной длины. REG\_EXPAND\_SZ

API функции для работы с реестром

Дескрипторов ключа реестра является переменная типа HKEY.

HKEY Является указателем на соответствующую структуру.

Для открытия ключа реестра используется функция RegOpenKey(HKEY\_CURRENT\_USER, NULL, &hKey);

Она возвращает числовое значение (которое интерпретируется как код). Код успешного завершения этой функции определяется макросом ERROR\_SUCCESS

В качестве аргументов этой функции выступают три параметра:

* Название глобальной ветки реестра (одной из 5). Передаётся в качестве макроса
* Пути к конкретному ключу от глобальной ветки (Весь остальной путь). Если нужно создать ключ в самой ветки, то в качестве второго параметра пишется NULL
* Ссылка на дескриптор HKEY.

Примеры использования этой функции в программном коде:

HKEY hKey = NULL;//дескриттор ключа реестра (он является структурой)

if (RegOpenKey(HKEY\_CURRENT\_USER, NULL, &hKey) != ERROR\_SUCCESS) //открываем раздел HKEY\_CURRENT\_USER

return 1;

Функция RegCreateKey открывает ключ, если он есть или создаёт его. Имеет такие же параметры и тип возвращаемого значения, как и RegOpenKey

Для того, чтобы создать параметр используется функция RegSetValue.

Возвращаемое значение у неё такое же, как и у предыдущих функций, Аргументы следующие:

* Дескриптор ключа реестра
* Название параметра, который мы будем создавать или открывать
* Тип параметра реестра
* Значение параметра
* Объем выделяемой памяти под значения параметра

Пример использования данной функции:

if (RegSetValueW(hKey, L"Mykey", REG\_SZ, L"Значение по умолчанию", 22 \* sizeof(WCHAR)) == ERROR\_SUCCESS)

{

MessageBox(NULL, L"Ключ успешно создан и ему присвоено значение по умолчанию", L"Информация", MB\_OK);

}

Для того чтобы получить значение параметра Используется функция RegGetValue

Возвращаемое значение у неё то же, аргументы следующие:

* Дескриптор ключа реестра
* Путь к подразделу реестра
* Типы допустимых значений
* Указатель на переменную, которая хранит тип данных для ключа реестра. То есть функция возвращает код того параметра, который считала из реестра.
* Указатель на переменную, в которую запишется значение из реестра

Пример использования данной функции:

DWORD DWValue = 0;

if (RegGetValueW(hKey, L"Mykey",L"MyDwordParam" , RRF\_RT\_ANY, &DataType, &DWValue, &DataLen) == ERROR\_SUCCESS)

{

LPWSTR OutputString = malloc(512);

swprintf(OutputString, 512, TEXT("%d"), DWValue\*2);

MessageBox(NULL, OutputString, L"Значение параметра", MB\_OK);

}

else

{

MessageBox(NULL, L"Что-то пошло не так", L"Информация", MB\_OK);

}

Для удаления параметра используется функция RegDeleteValue

Возвращает тот же код, вот а в качестве аргументов указываются Дескриптор ключа реестра и имя параметра.

if (RegDeleteValue(hKey, L"MyDwordParam") == ERROR\_SUCCESS)

{

MessageBox(NULL, L"Параметр успешно удален", L"Информация", MB\_OK);

}

Для удаления ключа используется функция RegDeleteKey.

Её использование по аналогии с удалением параметра

if (RegDeleteKey(hKey, L"MyKey") == ERROR\_SUCCESS)

{

MessageBox(NULL, L"Ключ успешно удален", L"Информация", MB\_OK);

}

Для освобождения памяти под дескрипторы ключа используется функция RegCloseKey(hKey);

В качестве аргумента ей необходимо передать дескриптор ключа реестра

**Перехватчики системных событий (Windows HOOK)**

Хуками называются перехватчики системных событий. Они позволяют программно подменять обработчик этих событий.

Мы будем рассматривать работу хуков на примере перехватчика сигналов нажатия клавиатуры.

За установку хука отвечает функция SetWindowsHookEx Она возвращает Указатель на структуру HHOOK.

Она имеет следующие аргументы:

* Флаг, отвечающий за тип события
* Указатель на функцию, которая будет являться обработчиком события
* Дескриптор в библиотеке DLL, связанный с с обработчиком события. Если обработчик находится не в dll, то ставим NULL
* Идентификатор потока, на которой распространяется хук. Если указать 0, то он будет работать со всеми потоками

Сама работа huka направлена на обработку системных событий. Поэтому также необходимо обрабатывать список входящих системных событий.

за это отвечает функция GetMessageW. Она возвращает логическое значение, аргументы у неё следующие:

* Указатель на структуру MSG
* Дескриптор окна. мы ставим NULL
* Порог самого низкоуровневого сообщения. NULL - Для всех входящих сообщений
* Порог самого высокоуровневого сообщения. NULL - Для всех входящих сообщений

Для того чтобы интерпретировать события нажатия виртуальные клавиши используется функция TranslateMessage.

Она возвращает логическое значение и принимает в качестве аргумента указатель на структуру MSG.

Для отправки сообщения в в обработчик хука используется функция DispatchMessage.

Она возвращает числовой код, а в качестве аргумента также принимает указатель на структуру MSG.

Функция UnhookWindowsHookEx(hHook); Освобождает дескриптор хука.

Также для правильного интерпретации нажатые клавиши важно понятию регистр.

Символ должен быть напечатан в Верхнем регистре в двух раздельных случаях:

* Если зажата клавиша Shift
* если нажата клавиша caps lock

Для проверки этих ситуаций используются функция

GetKeyState

Она принимает в качестве аргумента код виртуальной клавиши и возвращает значение типа SHORT.

И для того чтобы это значение перевести в логический тип используется операция конъюнкция с определённой константой

Пример функции:

BOOL IsCaps(void)//функция, которая проверяет регистр букв. TRUE - если в верхнем регистре

{

//GetKeyState используется в основном для определения состояния нажатия системной кнопки

//VK - Virtual Key

// ^ - это XOR (потому что Shift во время нажатого CapsLock опять делает букву в нижнем регистре)

if ((GetKeyState(VK\_CAPITAL) & 0x0001) != 0 ^ (GetKeyState(VK\_SHIFT) & 0x8000) != 0)

return TRUE;

return FALSE;

}

Также для работы программы необходимо вспомогательная функция записи в файл.

Здесь используется стандартный подход языка си. единственное ограничение на то что записываются символы юникода.

VOID WriteToFile(LPWSTR wstr)//функция записи в файл

{

FILE\* f = NULL;

if (!\_wfopen\_s(&f, PATH, L"ab"))

{

fwrite(wstr, sizeof(WCHAR), wcslen(wstr), f);

fclose(f);

}

}

Теперь рассмотрим более детально функцию обработки хука

LRESULT CALLBACK LogKey(int iCode, WPARAM wParam, LPARAM lParam)//функция обработчика системного сообщения

В качестве аргументов данной функции передаются параметры из структуры MSG

wParam Содержит код системного события.

Для нас нужно событие нажатой клавиши. оно будет обрабатываться следующим образом:

if (wParam == WM\_KEYDOWN)

В lParam Содержится код действия, которое повлекло событий ( например, Какая именно кнопка нажата).

Но для этого его предварительно необходимо распарсить в структуру Хука

PKBDLLHOOKSTRUCT pHook = (PKBDLLHOOKSTRUCT)lParam;

Далее для работы Нам необходимо получить раскладку клавиатуры

Это можно сделать путем использования комбинации функций

WORD KeyLayout = LOWORD(GetKeyboardLayout(GetWindowThreadProcessId(GetForegroundWindow(), 0)));

//GetForegroundWindow возвращает дескриптор активного в данный момент окна

//GetWindowThreadProcessId возвращает ID данного процесса

//GetKeyboardLayout возвращает раскладку клавиатуры, используемую в этом процессе

LOWORD Преобразует в формат Word результат, возвращаемое функцией GetKeyboardLayout

Код виртуальные клавиши получается следующим образом:

DWORD iKey = MapVirtualKeyW(pHook->vkCode, NULL) << 16;

Первые 32 символа является непечатными.

И чтобы их правильно интерпретировать нужно выполнить следующую операцию

if (!(pHook->vkCode <= 1 << 5)) // 32 (т.к. первые 32 символа являются не печатными)

iKey |= 0x1 << 24; //Задаём истину для 24 бита (|= - побитовое присваивание a|= b эквивалентно a = a|b)

Занята репутацию кода клавиши отвечает функция GetKeyNameText

в качестве аргументов она принимает следующее:

* Код нажатой клавиши в формате Long
* Строка, в которую запишется название клавиши
* Длина строки

Далее надо обработать результат функции GetKeyNameText

Если длина строки больше 1, то это означает, что нажата не символьная клавиша

Её название будет записано в текстовый файл в квадратных скобках:

if (wcslen(KeyString) > 1) //Если нажата не текстовая клавиша

{

WriteToFile(L"[");

WriteToFile(KeyString);

WriteToFile(L"]");

}

В другом случае мы записываем название символьные клавиши как есть (Только перед этим необходимо проверить её регистр)

if (!IsCaps()) KeyString[0] = tolower(KeyString[0]);//переводим в нижний регистр.

Функция GetKeyNameText по умолчанию возвращает название клавиши в верхнем регистре И только на латинице.

Для того, чтобы записать название клавиши на кириллице необходимо также учесть раскладку клавиатуры

if (KeyLayout == ENG)//если английская раскладка

{

//сюда будем писать код, если есть какие-то спецсимволы на английской раскладке

}

if (KeyLayout == RUS)//если русская раскладка

{

KeyString[0] = EnToRus(KeyString[0]);

}

Для перевода английской клавиша в русскую можно воспользоваться самостоятельно написанной функцией через switch.

Пример такой функции:

WCHAR EnToRus(WCHAR c)

{

switch (c)

{

case L'q':

return L'й';

case L'w':

return L'ц';

//и так далее

default:

return L' ';

}

}

Также надо учесть, что в конце обработчика хука необходимо вернуть следующее:

return CallNextHookEx(NULL, iCode, wParam, lParam);