Теория по теме “Создание многопоточного приложения для ОС MS Windows используя WinAPI”

Создание многопоточного приложения

Основной поток, который присутствует в каждой программе, начинает свое исполнение с функции **int main(int arc, char** argv).**

В процессе работы любого потока им могут быть созданы дополнительные потоки, которым поручают исполнение некоторых задач. В **WinAPI** для этого предусмотрена функция **CreateThread**. Примеры использования данной функции будут приведены далее. Ее интерфейс:

```
HANDLE CreateThread(
  LPSECURITY_ATTRIBUTES lpThreadAttributes, // атрибут защиты
  DWORD dwStackSize, //размер стека потока в байтах
  LPTHREAD_START_ROUTINE lpStartAddress, //адрес исполняемой потоком функции
  LPVOID lpParameter, //адрес передаваемого параметра
  DWORD dwCreationFlags, //флаги создания потока
  LPDWORD lpThreadId);/ / уникальный идентификатор потока
```

При успешном завершении функция **CreateThread** возвращает дескриптор созданного потока и его идентификатор, который является уникальным в пределах системы, иначе возвращается значение NULL.

Описание параметров:

lpThreadAttributes – атрибуты защиты создаваемого потока. Передача значения NULL сообщает об установке таких же атрибутов защиты, как и у создавшего его процесса.

dwStackSize – размер стека в байтах, выделяемого потоку при запуске. Если передаваемое значение меньше 1Mb, то размер стека будет установлен в 1Mb, так как это наименьший допустимый размер. В противном случае будет выделен указанный размер стека, округленный до размера одной страницы памяти (обычно около 4Kb).

lpStartAddress – адрес исполняемой потоков функции. Она должна реализовывать следующий интерфейс:

```
DWORD WINAPI ThreadFunction(LPVOID lpParameters);
```

lpParameter – адрес передаваемого в поток параметра. Допустимо передавать лишь один параметр, следовательно, передать группу параметром можно создав некоторую структуру, содержащую в себе все необходимые параметры.

dwCreationFlags – задает начально состояние потока. Если значение параметра равно 0, то поток будет запущен сразу же после создания. Если же значение будет равно CREATE_SUSPENDED, то созданный поток запущен не будет. В дальнейшем его можно запустить, используя метод **ResumeThread**.

lpThreadId – идентификатор потока. Данный параметр является выходным, и его значение будет установлено операционной системой. Его значение будет уникально в пределах системы и может в дальнейшем использоваться для доступа к потоку.

Для ожидания завершения работы потока предусмотрена функция **WaitForSingleObject**. Ее интерфейс:

```
DWORD WaitForSingleObject(
   HANDLE hObject // дескриптор некоторого потока
   , DWORD  dwTimeout); //время ожидания в миллисекундах
```

Описание параметров:

hObject - дескриптор потока, до завершения работы которого основной поток будет приостановлен.
dwTimeout – время ожидания завершения работы потока в миллисекундах. Если значение будет равно INFINITE, то основной поток продолжит свое исполнение лишь после завершения текущего.

Каждый поток исполняется до тех пор, пока не произойдет одно из нижеследующих событий:

- Поток вызывает функцию **ExitThread**.
- Какой-либо поток процесса вызывает функцию **ExitProcess**.
- Функция возвращает значение потока.
- Какой-либо поток вызывает функцию **TerminateThread** с дескриптором потока.
- Какой-либо поток вызывает функцию **TerminateProcess** с дескриптором процесса.

Функция **CloseHandle** закрывает дескриптор открытого объекта. Она аннулирует заданный дескриптор объекта, уменьшает итоговое число дескрипторов объекта и выполняет проверку наличия объекта. После того, как последний дескриптор объект закрывается, объект удаляется из системы. Закрытие дескриптора потока не завершает работу связанного потока. Чтобы удалить объект потока, вы должны завершить работу потока, затем закрыть все дескрипторы потока. Ее интерфейс:

```
BOOL CloseHandle(
```

```
    HANDLE hObject   // дескриптор объекта
);
```

Полная информация о внутреннем устройстве потока описана в [1] во 2-й главе.
Для управления потоками и получения информации о них, используются следующие функции:

- GetCurrentThread
- GetCurrentThreadId
- GetExitCodeThread
- GetThreadPriority
- GetThreadTimes
- ResumeThread
- SetThreadPriority
- Sleep

Синхронизация потоков через мьютексы

В случае, когда в программе есть некоторый ресурс, владеть которым одновременно можно позволить только одному потоку, необходимо каким-либо образом блокировать область, где используется данный ресурс, от возможности одновременного входа в данную область более чем одного потока. Для решения данной задачи в многопроцессной среде принято использовать объект синхронизации, именуемый мьютексом.

В WinApi для работы с мьютексами предусмотрены функции **CreateMutex** и **ReleaseMutex**.
Тема выносится на самостоятельное изучение.

Планирование и приоритеты потоков и процессов.

Поток выполняет код и манипулирует данными в адресном пространстве своего процесса. Через определенный интервал времени (выделенный квант времени) операционная система сохранит значения регистров процессора в контексте потока и приостановит его выполнение. Далее система просмотрит остальные существующие потоки, ожидающие исполнения, выберет, с учетом приоритетов, один из них и загрузит его контекст в регистры процессора. В случае многоядерных систем число одновременно исполняемых потоков равно числу ядер. В случае поддержки процессором технологии Hyper-threading, это число возрастает в два раза.

Этот цикл операций (выбор потока - загрузка его контекста – выполнение - сохранение) начинается с момента запуска системы и продолжается до ее выключения. Таков вкратце механизм планирования работы множества потоков. Алгоритм планирования потоков существенно влияет на выполнение приложений.

В Windows используется принцип планирования, основанный на приоритетах – атрибутах, назначенных процессу. Абсолютный приоритет потока вычисляется как сумма приоритета процесса, которому принадлежит поток и относительного приоритета потока.

Допустимые значения приоритетов потоков (приведены по возрастанию приоритета):

- THREAD_PRIORITY_IDLE
- THREAD_PRIORITY_LOWEST
- THREAD_PRIORITY_BELOW_NORMAL
- THREAD_PRIORITY_NORMAL
- THREAD_PRIORITY_ABOVE_NORMAL
- THREAD_PRIORITY_HIGHEST
- THREAD_PRIORITY_TIME_CRITICAL

Потоки с более высоким приоритетом всегда вытесняют потоки с более низким приоритетом независимо от того, выполняются они или нет.

Допустимые значения приоритетов процессов (приведены по возрастанию приоритета):

- IDLE_PRIORITY_CLASS
- BELOW_NORMAL_PRIORITY_CLASS
- NORMAL_PRIORITY_CLASS
- ABOVE_NORMAL_PRIORITY_CLASS
- HIGH_PRIORITY_CLASS
- REALTIME_PRIORITY_CLASS

Для установки приоритета потока предусмотрена функция SetThreadPriority. Ее интерфейс:

```
BOOL SetThreadPriority(
  HANDLE hThread, // дескриптор потока
  int nPriority   // значение приоритета потока
);
```

Для установки приоритета процесса предусмотрена функция SetPriorityClass. Ее интерфейс:

```
BOOL SetPriorityClass(
  HANDLE hProcess,       // дескриптор процесса
  DWORD dwPriorityClass   // значение приоритета процесса
);
```

Для возврата (получения) описателя текущего процесса предусмотрена функция **GetCurrentProcess**. Ее интерфейс:

```
HANDLE GetCurrentProcess(VOID);
```

Теория по следующим функциям отводится на *самостоятельное изучение*:

- WaitForMultipleObjects
- CreateMutex
- ReleaseMutex
- ResumeThread
- ExitThread
- ExitProcess
- TerminateThread
- TerminateProcess
- GetCurrentThread
- GetCurrentThreadId
- GetExitCodeThread
- GetThreadPriority
- GetThreadTimes
- ResumeThread
- SetThreadPriority
- Sleep

Рекомендованная литература:

1. Рихтер Дж. «Windows для профессионалов: создание эффективных WIN32 приложений с учетом специфики 64-х разрядной версии Windows», «Питер», СПб, 2001 г.
2. Вильямс А. «Системное программирование в Windows 2000 для профессионалов», СПб, Питер, 2001 г
3. http://www.vsokovikov.narod.ru/New_MSDN_API/ref_api.htm - русскоязычный справочник по WinAPI.
4. http://www.cplusplus.com/
5. http://ru.wikipedia.org/wiki/Мьютекс
6. http://ru.wikipedia.org/wiki/Hyper-threading

Листинг 1:

```
#include <iostream>
#include <windows.h>

DWORD WINAPI Inc(LPVOID arg)
{
   *(static_cast<double*>(arg)) += 1;
   return 0;
}

int main()
{
   double value = 0.23;

   std::cout << "before: value = " << value << std::endl;

   HANDLE threadHandler;
   DWORD threadId;

   threadHandler = CreateThread(NULL, 0, Inc, static_cast<void*>(&value), 0, &threadId);
   if (threadHandler == NULL)
      return GetLastError();

   WaitForSingleObject(threadHandler, INFINITE);
   CloseHandle(threadHandler);

   std::cout << "after: value = " << value << std::endl;

   return 0;
}
```

Листинг 2:

```
#include <iostream>
#include <windows.h>
#include <string>

struct TStudent {
```

```
public:
   size_t Id_;
   std::string Name_;
   size_t Age_;
   double Points_;

public:
   TStudent(const size_t id, const std::string& name, const size_t age)
      : Id_(id)
      , Name_(name)
      , Age_(age)
      , Points_(0.0)
   {
   }
};

std::ostream& operator<<(std::ostream& out, const TStudent& student) {
   out << "id: " << student.Id_
      << "\nname: " << student.Name_
      << "\nage: " << student.Age_
      << "\npoints: " << student.Points_;
   return out;
}

struct TThreadArgs {
public:
   TStudent* StudentPtr_;
   double Points_;

public:
   TThreadArgs(TStudent* studentPtr, const double points)
      : StudentPtr_(studentPtr)
      , Points_(points)
   {
   }
};

DWORD WINAPI UpdateStatus(LPVOID arg)
```

```
{
    TThreadArgs* tArgs = static_cast<TThreadArgs*>(arg);
    tArgs->StudentPtr_->Points_ += tArgs->Points_;
    return 0;
}

int main() {
    TStudent Ivanov(123123, "Ivanov I.I.", 19);
    TThreadArgs threadArgs(&Ivanov, 3.4);

    std::cout << "before: " << std::endl
        << Ivanov << std::endl;

    HANDLE threadHandler;
    DWORD threadId;

    threadHandler = CreateThread(NULL, 0, UpdateStatus, static_cast<void*>(&threadArgs), 0, &threadId);

    if (threadHandler == NULL)
        return GetLastError();

    WaitForSingleObject(threadHandler, INFINITE);
    CloseHandle(threadHandler);

    std::cout << "after: " << std::endl
        << Ivanov << std::endl;

    return 0;
}
```