ЛАБОРАТОРНАЯ РАБОТА 3.
УПРАВЛЕНИЕ ПРОЦЕССАМИ В WINDOWS

Цель работы: Изучение механизмов управления процессами в операционной системе Windows, приобретение навыков программирования для создания, завершения, приостановки и возобновления процессов, а также для получения информации о процессах с использованием системных вызовов Windows API.

Основные теоретические сведения

Управление процессами в Windows API (WinAPI) включает в себя создание, управление и взаимодействие с процессами и потоками в операционной системе Windows. Ниже приведены основные функции и концепции, используемые для работы с процессами в WinAPI.

Создание процессов в операционной системе Windows с использованием Windows API (WinAPI) осуществляется с помощью функции `CreateProcess`. Эта функция позволяет запустить новый процесс и настроить различные атрибуты его выполнения. Пример использования функции `CreateProcess` на языке C++:

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

int main() {
    SetConsoleOutputCP(1251);
    // Имя исполняемого файла, который нужно запустить
    LPCTSTR applicationName = _T("C:\\Path\\To\\Your\\Program.exe");

    // Командная строка для передачи процессу
    LPTSTR commandLine = NULL;

    // Защитные атрибуты процесса и потока (обычно NULL)
    LPSECURITY_ATTRIBUTES processAttributes = NULL;
    LPSECURITY_ATTRIBUTES threadAttributes = NULL;

    // Флаги создания процесса
    BOOL inheritHandles = FALSE;
```

```
    DWORD creationFlags = 0;
    LPVOID environment = NULL;
    LPCTSTR currentDirectory = NULL;
    STARTUPINFO startupInfo;
    PROCESS_INFORMATION processInfo;

    // Заполнение структуры STARTUPINFO
    ZeroMemory(&startupInfo, sizeof(STARTUPINFO));
    startupInfo.cb = sizeof(STARTUPINFO);

    // Создание нового процесса
    BOOL result = CreateProcess(
        applicationName,     // Имя исполняемого файла
        commandLine,         // Командная строка
        processAttributes,   // Атрибуты процесса
        threadAttributes,    // Атрибуты потока
        inheritHandles,      // Флаг наследования дескрипторов
        creationFlags,       // Флаги создания процесса
        environment,         // Переменные окружения (обычно NULL)
        currentDirectory,    // Текущий рабочий каталог (обычно NULL)
        &startupInfo,        // Структура STARTUPINFO
        &processInfo         // Структура PROCESS_INFORMATION
    );

    if (result) {
        // Процесс успешно создан
        // Вы можете получить информацию о процессе, используя processInfo

        // Закрыть дескрипторы, чтобы избежать утечек ресурсов
        CloseHandle(processInfo.hProcess);
        CloseHandle(processInfo.hThread);
    } else {
        // Произошла ошибка при создании процесса
        DWORD error = GetLastError();
        // Обработка ошибки
    }

    return 0;
}
```

Функция `CreateProcess` возвращает `TRUE`, если процесс успешно создан, и `FALSE`, если произошла ошибка. В случае ошибки, можно использовать `GetLastError` для получения кода ошибки и дальнейшей обработки.

Функция `CreateProcess` имеет множество параметров для настройки создания нового процесса. Основные из них:

- **`lpApplicationName (LPCTSTR)`**: имя исполняемого файла, который нужно запустить. Этот параметр может быть `NULL`, если имя исполняемого файла включено в строку командной строки.
- **`lpCommandLine (LPTSTR)`**: командная строка, передаваемая запускаемому процессу. Этот параметр может содержать имя исполняемого

файла и его аргументы. Если `lpApplicationName` не является `NULL`, то `lpCommandLine` будет использоваться для передачи аргументов командной строки. В противном случае, `lpCommandLine` должен содержать полную команду.

- **`lpProcessAttributes (LPSECURITY_ATTRIBUTES)`**: атрибуты безопасности процесса. Обычно устанавливаются в `NULL` для использования атрибутов по умолчанию.
- **`lpThreadAttributes (LPSECURITY_ATTRIBUTES)`**: атрибуты безопасности потока. Обычно устанавливаются в `NULL` для использования атрибутов по умолчанию.
- **`bInheritHandles (BOOL)`**: флаг, указывающий, должны ли дескрипторы открытых файлов и другие ресурсы наследоваться процессом, созданным функцией `CreateProcess`.
- **`dwCreationFlags (DWORD)`**: флаги создания процесса, определяющие различные параметры и поведение процесса. Например, вы можете использовать `CREATE_NEW_CONSOLE`, чтобы создать новое окно консоли для процесса.
- **`lpEnvironment (LPVOID)`**: указатель на блок переменных окружения, которые будут использоваться в новом процессе. Обычно устанавливается в `NULL`, чтобы процесс унаследовал текущее окружение.
- **`lpCurrentDirectory (LPCTSTR)`**: текущий рабочий каталог для нового процесса. Обычно устанавливается в `NULL`, чтобы процесс использовал текущий рабочий каталог родительского процесса.
- **`lpStartupInfo (LPSTARTUPINFO)`**: указатель на структуру `STARTUPINFO`, которая содержит информацию о создаваемом процессе, такую как дескрипторы для ввода, вывода и ошибок.
- **`lpProcessInformation (LPPROCESS_INFORMATION)`**: указатель на структуру `PROCESS_INFORMATION`, в которой будут возвращены дескрипторы процесса и потока после успешного создания процесса.

Для получения информации о процессе в операционной системе Windows с помощью Windows API (WinAPI) можно использовать несколько функций и структур данных. Вот основные средства для получения информации о процессе:

- **`OpenProcess`**: позволяет открыть существующий процесс и получить дескриптор процесса (`HANDLE`), который можно использовать для выполнения различных операций над процессом. Эта функция принимает в качестве параметров идентификатор процесса (`PID`) и права доступа.

```
HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, processId);
```

- **`GetProcessId`**: используется для получения идентификатора процесса (PID) по его дескриптору.

```
DWORD processId = GetProcessId(hProcess);
```

- **`GetProcessTimes`**: позволяет получить информацию о времени выполнения процесса, включая время начала выполнения и использования ЦП.

```
FILETIME creationTime, exitTime, kernelTime, userTime;
if (GetProcessTimes(hProcess, &creationTime, &exitTime, &kernelTime,
&userTime)) {
    // Обработка информации о времени выполнения процесса
}
```

- **`GetProcessMemoryInfo`**: позволяет получить информацию о потреблении памяти процессом.

```
PROCESS_MEMORY_COUNTERS memInfo;
if (GetProcessMemoryInfo(hProcess, &memInfo, sizeof(memInfo))) {
    // Обработка информации о потреблении памяти процессом
}
```

- **`QueryFullProcessImageName`**: используется для получения полного пути к исполняемому файлу процесса.

```
TCHAR processPath[MAX_PATH];
DWORD pathSize = sizeof(processPath) / sizeof(TCHAR);
if (QueryFullProcessImageName(hProcess, 0, processPath, &pathSize)) {
    // Обработка пути к исполняемому файлу процесса
}
```

- **`CloseHandle`**: используется для освобождения ресурсов, связанных с дескриптором процесса.

```
CloseHandle(hProcess);
```

В приведенных примерах `hProcess` – это дескриптор открытого процесса, а `processId` – идентификатор процесса.

Управление приоритетом процессов в операционной системе Windows можно выполнять с помощью Windows API (WinAPI). Приоритет процесса определяет, как операционная система распределяет процессорное время между процессами. Основные функции и концепции, связанные с управлением приоритетом процессов:

- **Установка приоритета процесса**. Для установки приоритета процесса используется функция `SetPriorityClass`. Эта функция изменяет приоритет выполнения всего процесса.

```
BOOL success = SetPriorityClass(GetCurrentProcess(), priority);
```

Здесь priority может принимать одно из следующих значений:

  - `HIGH_PRIORITY_CLASS`: высокий приоритет.
  - `NORMAL_PRIORITY_CLASS`: нормальный приоритет (по умолчанию).
  - `IDLE_PRIORITY_CLASS`: низкий приоритет.
  - `REALTIME_PRIORITY_CLASS`: реальное время (осторожно при использовании, так как это может привести к зависанию системы).

- **Получение текущего приоритета процесса**. Для получения текущего приоритета процесса используется функция `GetPriorityClass`.

```
DWORD priority = GetPriorityClass(GetCurrentProcess());
```

Значение `priority` будет одним из перечисленных выше констант.

- **Установка приоритета потока**. Внутри процесса можно устанавливать приоритет для отдельных потоков с помощью функции `SetThreadPriority`. Это позволяет управлять приоритетами выполнения различных задач внутри одного процесса.

```
BOOL success = SetThreadPriority(hThread, priority);
```

Здесь `hThread` – дескриптор потока, а `priority` – желаемый приоритет потока.

- **Получение текущего приоритета потока**. Для получения текущего приоритета потока используется функция `GetThreadPriority`.

```
int priority = GetThreadPriority(hThread);
```

Значение `priority` будет числовым представлением приоритета потока.

Управление потоками внутри процесса в операционной системе Windows можно выполнять с использованием Windows API (WinAPI). Вот основные функции и концепции, связанные с управлением потоками:

- **Создание потока.** Для создания нового потока внутри процесса используется функция `CreateThread`. Эта функция позволяет запустить новый поток и выполнить в нем определенную функцию.

```
HANDLE hThread = CreateThread(
    NULL,                   // Атрибуты безопасности потока (обычно NULL)
    0,                      // Размер стека (0 = размер стека по умолчанию)
    ThreadFunction,         // Функция, которая будет выполнена в потоке
    lpParam,                // Дополнительные параметры для функции
    0,                      // Флаги создания потока (0 = запуск сразу после
создания)
    &dwThreadId             // Идентификатор потока
);
```

Здесь `ThreadFunction` – это указатель на функцию, которая будет выполняться в потоке, и `lpParam` – дополнительные параметры, которые могут быть переданы в функцию.

- **Завершение потока.** Для завершения выполнения потока используется функция ExitThread. Вызов этой функции приведет к завершению текущего потока.

```
ExitThread(0);
```

- **Ожидание завершения потока.** Для ожидания завершения выполнения потока используется функци `WaitForSingleObject` или `WaitForMultipleObjects`, в зависимости от количества потоков, которые нужно ожидать.

```
DWORD dwExitCode;
DWORD dwWaitResult = WaitForSingleObject(hThread, INFINITE);

if (dwWaitResult == WAIT_OBJECT_0) {
```

```
    // Поток завершил выполнение
    GetExitCodeThread(hThread, &dwExitCode);
    // Обработка результата выполнения потока (dwExitCode)
} else {
    // Обработка ошибки ожидания
}
```

- **Установка приоритета потока.** Установка приоритета выполнения потока внутри процесса выполняется с помощью функции `SetThreadPriority`.

```
BOOL success = SetThreadPriority(hThread, priority);
```

Здесь `hThread` – дескриптор потока, а `priority` – желаемый приоритет потока.

- **Получение текущего приоритета потока.** Для получения текущего приоритета потока используется функция `GetThreadPriority`.

```
int priority = GetThreadPriority(hThread);
```

- **Закрытие дескриптора потока.** После завершения работы с потоком, его дескриптор должен быть закрыт с помощью функции `CloseHandle`.

```
CloseHandle(hThread);
```

Основные механизмы синхронизации, доступные в Windows API:

- **Критические секции (Critical Sections):** предоставляют простой и легковесный способ синхронизации между потоками внутри одного процесса. Функции для работы с критическими секциями включают `InitializeCriticalSection`, `EnterCriticalSection`, `LeaveCriticalSection`, и `DeleteCriticalSection`.
- **Мьютексы (Mutexes)**: могут использоваться для синхронизации между разными процессами, а не только потоками внутри одного процесса. Функции для работы с мьютексами включают `CreateMutex`, `WaitForSingleObject`, и `ReleaseMutex`.
- **Семафоры (Semaphores)**: позволяют контролировать доступ к ресурсам, когда требуется счетчик. Функции для работы с семафорами включают `CreateSemaphore`, `WaitForSingleObject`, и `ReleaseSemaphore`.
- **События (Events)**: События используются для уведомления одного или нескольких потоков или процессов о возникновении события. Функции для работы с событиями включают `CreateEvent`, `SetEvent`, `WaitForSingleObject`, и другие.
- **Критические ресурсы и мьютексы файла (File Mapping and File Mutexes)**: позволяют синхронизировать доступ к разделяемым данным, которые находятся в памяти или на диске. Самые распространенные функции включают `CreateFileMapping`, `MapViewOfFile`, и `WaitForSingleObject`.
- **Readers-Writers Locks**: позволяют определять правила доступа для читающих и записывающих потоков к общим данным. Функции для работы с

Reader-Writer Locks включают `InitializeSRWLock`, `AcquireSRWLockExclusive`, и `AcquireSRWLockShared`.

- **События завершения (Completion Events)**: используются для уведомления о завершении выполнения асинхронных операций ввода – вывода. Функции для работы с событиями завершения включают `CreateIoCompletionPort`, `GetQueuedCompletionStatus`, и другие.

Выбор подходящего механизма синхронизации зависит от конкретных требований приложения и структуры данных. Важно правильно использовать синхронизацию, чтобы избежать проблем с гонками данных, блокировками и ожиданиями.

В Windows API (WinAPI) есть несколько способов завершения процессов, включая нормальное завершение и принудительное завершение. Вот основные функции и методы для завершения процессов:

- **ExitProcess** – используется для нормального завершения текущего процесса. Это завершает выполнение текущей программы и завершает процесс.

```
ExitProcess(exitCode);
```

Здесь `exitCode` – код завершения процесса, который будет возвращен операционной системой.

- **TerminateProcess** – используется для принудительного завершения другого процесса. Это позволяет завершить процесс, даже если он не отвечает или заблокирован.

```
BOOL success = TerminateProcess(hProcess, exitCode);
```

Здесь `hProcess` – дескриптор процесса, который нужно завершить, и `exitCode` – код завершения процесса.

- **WM_CLOSE и PostMessage**: если необходимо завершить приложение с графическим интерфейсом, вы можете отправить сообщение `WM_CLOSE` главному окну приложения с помощью функции `PostMessage`. Это позволит приложению выполнить закрытие как обычно.

```
PostMessage(hWnd, WM_CLOSE, 0, 0);
```

Здесь `hWnd` – дескриптор главного окна приложения.

Обратите внимание, что при использовании функции `TerminateProcess` процесс завершается немедленно и не выполняет никаких завершающих операций, что может привести к утечкам ресурсов и несохранению данных. По возможности рекомендуется использовать нормальное завершение процессов с помощью `ExitProcess` или закрытие окон с использованием `WM_CLOSE` и `PostMessage`, чтобы приложение могло выполнить необходимые действия перед завершением.

Для получения информации о процессах в операционной системе Windows можно использовать Windows API. Существует несколько функций и структур данных, которые позволяют получить информацию о текущих работающих процессах. Пример получения списка процессов и их атрибутов:

```
#include <windows.h>
#include <stdio.h>
#include <tlhelp32.h>
#include <tchar.h>

int main() {
    SetConsoleOutputCP(1251);
    // Создаем объект, представляющий снимок всех процессов
    HANDLE hProcessSnap = CreateToolhelp32Snapshot(TH32CS_SNAPPROCESS, 0);

    if (hProcessSnap == INVALID_HANDLE_VALUE) {
        // Обработка ошибки
        return 1;
    }

    // Структура, в которую будет сохранен атрибут процесса
    PROCESSENTRY32 pe32;
    pe32.dwSize = sizeof(PROCESSENTRY32);

    // Получаем информацию о первом процессе в снимке
    if (!Process32First(hProcessSnap, &pe32)) {
        CloseHandle(hProcessSnap);
        // Обработка ошибки
        return 1;
    }

    // Перебираем все процессы в снимке
    do {
        _tprintf(_T("Процесс ID: %d, Имя: %s\n"), pe32.th32ProcessID,
pe32.szExeFile);
        // Здесь можно получать и обрабатывать другие атрибуты процесса

    } while (Process32Next(hProcessSnap, &pe32));

    CloseHandle(hProcessSnap);

    return 0;
}
```

Для успешной компиляции и выполнения этого кода необходимо включить библиотеку `kernel32.lib` и указать верное символическое имя или путь к исполняемому файлу.

Для получения информации о модулях (библиотеках и исполняемых файлах) внутри процесса в операционной системе Windows, вы можете использовать Windows API. Основным инструментом для этой задачи является функция `EnumProcessModules`. Пример использования:

```
#include <windows.h>
#include <psapi.h>
#include <tchar.h>
#include <stdio.h>

int main() {
    SetConsoleOutputCP(1251);
```

```
    DWORD processId = GetCurrentProcessId(); // Идентификатор текущего процесса
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION | PROCESS_VM_READ,
FALSE, processId);

    if (hProcess == NULL) {
        // Обработка ошибки открытия процесса
        return 1;
    }

    HMODULE hModules[1024];
    DWORD cbNeeded;

    // Получаем список модулей внутри процесса
    if (EnumProcessModules(hProcess, hModules, sizeof(hModules), &cbNeeded)) {
        for (DWORD i = 0; i < (cbNeeded / sizeof(HMODULE)); i++) {
            TCHAR szModName[MAX_PATH];

            // Получаем имя модуля
            if (GetModuleFileNameEx(hProcess, hModules[i], szModName,
sizeof(szModName) / sizeof(TCHAR))) {
                _tprintf(_T("Модуль #%u: %s\n"), i, szModName);
            }
        }
    }

    CloseHandle(hProcess);
    return 0;
}
```

Не забудьте включить библиотеку `psapi.lib` при компиляции и убедитесь, что код выполняется с правами, позволяющими открывать процессы и читать их модули.

Функции `OpenProcessToken` и `GetTokenInformation` в Windows API используются для получения информации о безопасности и разрешениях, связанных с процессом. Рассмотрим их использование подробнее:

- **OpenProcessToken**: используется для открытия дескриптора безопасности (токена) процесса. Этот токен содержит информацию о безопасности, такую как SID (идентификатор безопасности) пользователя и группы, разрешения и другие атрибуты безопасности процесса. Вот как можно использовать `OpenProcessToken`:

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

int main() {
    SetConsoleOutputCP(1251);
    DWORD processId = ...; // Идентификатор процесса, для которого нужно
получить токен
    HANDLE hProcess = OpenProcess(PROCESS_QUERY_INFORMATION, FALSE, processId);

    if (hProcess == NULL) {
        // Обработка ошибки
        return 1;
```

```
    }

    HANDLE hToken;
    if (OpenProcessToken(hProcess, TOKEN_QUERY, &hToken)) {
        // Теперь у вас есть дескриптор токена (hToken) для процесса,
        // который вы можете использовать для получения информации о
безопасности.

        // Закрываем дескриптор токена после использования.
        CloseHandle(hToken);
    }

    CloseHandle(hProcess);
    return 0;
}
```

- **GetTokenInformation**: После открытия дескриптора токена с помощью `OpenProcessToken`, можно использовать функцию `GetTokenInformation`, чтобы получить различные атрибуты и информацию о безопасности. Необходимо предоставить буфер для хранения информации. Пример получения информации о SID (идентификаторе безопасности) пользователя из токена:

```
HANDLE hToken = ...; // Дескриптор токена
DWORD dwSize = 0;

// Получаем размер буфера, необходимый для информации о SID.
GetTokenInformation(hToken, TokenUser, NULL, 0, &dwSize);

// Выделяем буфер и получаем информацию о SID.
PTOKEN_USER pTokenUser = (PTOKEN_USER)malloc(dwSize);
if (GetTokenInformation(hToken, TokenUser, pTokenUser, dwSize, &dwSize)) {
    // Теперь у вас есть информация о SID пользователя.

    // Освобождаем выделенный буфер после использования.
    free(pTokenUser);
}

// Закрываем дескриптор токена.
CloseHandle(hToken);
```

Критические секции (Critical Sections) – это механизм синхронизации в Windows API, который позволяет защитить доступ к общим данным от одновременного доступа нескольких потоков внутри одного процесса. Они обычно используются для предотвращения гонок данных и обеспечения корректного доступа к разделяемым ресурсам. Пример использования критических секций:

```
#include <windows.h>

// Объявляем глобальную критическую секцию
```

```
CRITICAL_SECTION g_criticalSection;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Входим в критическую секцию
    EnterCriticalSection(&g_criticalSection);

    // Здесь можно выполнять операции с общими данными

    // Выходим из критической секции
    LeaveCriticalSection(&g_criticalSection);

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Инициализируем критическую секцию
    InitializeCriticalSection(&g_criticalSection);

    // Создаем потоки, которые будут использовать критическую секцию
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Уничтожаем критическую секцию
    DeleteCriticalSection(&g_criticalSection);

    return 0;
}
```

Использование критических секций обычно предпочтительнее внутри одного процесса, так как они более эффективны и проще в использовании по сравнению с другими механизмами синхронизации, такими как мьютексы и семафоры. Критические секции обеспечивают внутреннюю синхронизацию в пределах одного процесса и не подходят для синхронизации между разными процессами.

Мьютексы (Mutexes) – это механизм синхронизации в Windows API, который используется для управления доступом к разделяемым ресурсам, чтобы предотвратить гонки данных между несколькими потоками или процессами. Мьютексы обычно используются для синхронизации между потоками в разных процессах. Пример использования мьютексов для синхронизации между двумя потоками:

```c
#include <windows.h>
#include <stdio.h>

// Объявляем глобальный мьютекс
HANDLE g_mutex;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Попытка захвата мьютекса
    DWORD dwWaitResult = WaitForSingleObject(g_mutex, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Мьютекс успешно захвачен

        // Здесь можно выполнять операции с разделяемыми ресурсами

        // Освобождение мьютекса
        ReleaseMutex(g_mutex);
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем мьютекс
    g_mutex = CreateMutex(NULL, FALSE, NULL);

    if (g_mutex == NULL) {
        // Обработка ошибки создания мьютекса
        return 1;
    }

    // Создаем два потока
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Закрываем дескриптор мьютекса
    CloseHandle(g_mutex);

    return 0;
}
```

Это простой пример использования мьютексов для синхронизации между двумя потоками. Мьютексы также могут использоваться для синхронизации между разными процессами, если они используют один и тот же мьютекс с именем, доступным для обоих процессов.

Семафоры (Semaphores) – это ещё один механизм синхронизации в Windows API, который используется для контроля доступа к разделяемым ресурсам между несколькими потоками или процессами. Семафоры могут позволить нескольким потокам одновременно получить доступ к общему ресурсу в ограниченном количестве. Пример использования семафора для синхронизации между несколькими потоками:

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

// Объявляем глобальный семафор и устанавливаем начальное значение
HANDLE g_semaphore;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Попытка уменьшить счетчик семафора
    DWORD dwWaitResult = WaitForSingleObject(g_semaphore, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Семафор успешно уменьшен

        // Здесь можно выполнять операции с разделяемыми ресурсами

        // Увеличение счетчика семафора
        ReleaseSemaphore(g_semaphore, 1, NULL);
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем семафор с начальным счетчиком
    g_semaphore = CreateSemaphore(NULL, 2, 2, NULL); // В данном примере,
начальный счетчик равен 2

    if (g_semaphore == NULL) {
        // Обработка ошибки создания семафора
        return 1;
    }

    // Создаем два потока
    HANDLE hThread1 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);
    HANDLE hThread2 = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Ожидаем завершения потоков
```

```
    WaitForSingleObject(hThread1, INFINITE);
    WaitForSingleObject(hThread2, INFINITE);

    // Закрываем дескрипторы потоков
    CloseHandle(hThread1);
    CloseHandle(hThread2);

    // Закрываем дескриптор семафора
    CloseHandle(g_semaphore);

    return 0;
}
```

Семафоры предоставляют мощный механизм для управления доступом к разделяемым ресурсам в многопоточных и многопроцессных приложениях.

События (Events) – это механизм синхронизации в Windows API, который используется для уведомления одного или нескольких потоков о наступлении определенного события. События могут быть использованы для синхронизации между потоками или процессами, когда один поток ждет, пока другой поток или процесс оповестит его о наступлении события. Пример использования событий:

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

// Объявляем глобальное событие
HANDLE g_event;

// Функция, выполняемая в потоках
DWORD WINAPI ThreadFunction(LPVOID lpParam) {
    // Ожидание события
    DWORD dwWaitResult = WaitForSingleObject(g_event, INFINITE);

    if (dwWaitResult == WAIT_OBJECT_0) {
        // Событие успешно сработало

        // Здесь можно выполнять действия, связанные с событием

        printf("Событие сработало в потоке.\n");
    } else {
        // Обработка ошибки
    }

    return 0;
}

int main() {
    SetConsoleOutputCP(1251);
    // Создаем событие
    g_event = CreateEvent(NULL, FALSE, FALSE, NULL);

    if (g_event == NULL) {
        // Обработка ошибки создания события
```

```
        return 1;
    }

    // Создаем поток
    HANDLE hThread = CreateThread(NULL, 0, ThreadFunction, NULL, 0, NULL);

    // Задержка для демонстрации события
    Sleep(2000);

    // Устанавливаем событие
    SetEvent(g_event);

    // Ожидаем завершения потока
    WaitForSingleObject(hThread, INFINITE);

    // Закрываем дескриптор потока
    CloseHandle(hThread);

    // Закрываем дескриптор события
    CloseHandle(g_event);

    return 0;
}
```

События могут быть использованы для синхронизации и сигнализации между потоками или процессами, и они часто используются в многозадачных и многопроцессных приложениях для организации совместной работы потоков.

Порядок выполнения работы

1. Написать программу, которая создает новый процесс с помощью функции `CreateProcess`. Новый процесс должен выполнять простую консольную программу, которую также необходимо написать.
2. Расширить предыдущую программу, добавив возможность передачи аргументов командной строки новому процессу.
3. Написать программу, которая получает информацию о текущем процессе с помощью функции `GetCurrentProcess`.
4. Написать программу, которая получает идентификатор текущего процесса с помощью функции `GetCurrentProcessId` и выводит его на экран.
5. Написать программу, которая получает дескриптор текущего процесса и использует его для изменения приоритета процесса с помощью функции `SetPriorityClass`.
6. Написать программу, которая завершает работу другого процесса с помощью функции `TerminateProcess`. Для этого необходимо получить дескриптор процесса с помощью функции `OpenProcess`.
7. Написать программу, которая создает новый поток в текущем процессе с помощью функции `CreateThread`. Новый поток должен выполнять простую функцию, которую также необходимо написать.

8. Написать программу, которая синхронизирует работу нескольких потоков с помощью событий (функции `CreateEvent`, `SetEvent`, `WaitForSingleObject` или `WaitForMultipleObjects`).

9. Написать программу, которая использует функции `GetExitCodeProcess` и `GetExitCodeThread` для получения кодов завершения процесса и потока.

Контрольные вопросы

1. *Основные понятия:*
   - Что такое процесс в операционной системе Windows?
   - Какие атрибуты характеризуют процесс?
2. *Создание процессов:*
   - Какую функцию Windows API используют для создания нового процесса?
   - Какие параметры необходимо передать функции `CreateProcess` для успешного создания процесса?
   - Чем отличаются синхронное и асинхронное создание процесса?
3. *Получение информации о процессах:*
   - Какие функции Windows API используются для получения идентификатора текущего процесса и информации о других процессах?
   - Как получить информацию о времени работы процесса (начало, пользовательское время, системное время)?
4. *Завершение процессов:*
   - Какая функция используется для завершения процесса?
   - Какие параметры необходимы для завершения процесса с использованием функции `TerminateProcess`?
   - Как корректно завершить процесс, чтобы избежать утечек ресурсов?
5. *Приостановка и возобновление процессов:*
   - Какие функции Windows API используются для приостановки и возобновления процессов?
   - В каких случаях может потребоваться приостановка процесса?
   - Какой эффект может иметь приостановка основного потока процесса на его выполнение?
6. *Мониторинг процессов:*
   - Какие утилиты и инструменты в Windows можно использовать для мониторинга состояния процессов?
   - Как можно программно реализовать мониторинг активных процессов в системе?
   - Какие параметры процессов являются ключевыми для мониторинга?
7. *Безопасность и защита процессов:*
   - Какие методы защиты процессов от несанкционированного доступа существуют в Windows?
   - Как ограничить права доступа к процессу?

- Какие функции и параметры Windows API используются для реализации безопасности процессов?

8. *Практические аспекты:*
   - Как определить, что процесс успешно завершился?
   - Какие возможные ошибки могут возникнуть при управлении процессами, и как их диагностировать?
   - Как проверить, что процесс находится в ожидающем (приостановленном) состоянии?
9. *Отладка процессов:*
   - Какие средства и функции Windows API могут использоваться для отладки процессов?
   - Как программно осуществить пошаговое выполнение процесса?
   - Какие методы можно использовать для анализа причин сбоя процесса?