Функции ожидания (Wait-функции)

Ожидание на одном дескрипторе.

```
DWORD WaitForSingleObject( HANDLE hObject, DWORD
dwMilliseconds);
```

Пример вызова *WaitForSingleObject* без ограничения времени ожидания (параметр времени ожидания – INFINITE):

```
DWORD dw;
dw = WaitForSingleObject( hProcess, INFINITE );
```

Пример вызова *WaitForSingleObject* с ограничением времени ожидания (предел времени ожидания - 5 секунд):

```
WaitForSlngleObject(hProcess, 5000);
switch (dw){
    case WAIT_OBJECT_0:{
        // процесс завершился
        ...
        }
    break;
    case WAIT_TIMEOUT:{
        // процесс не завершился за 5000 мс
        ...
        }
    break;
    case WAIT_FAILED:{
        // ошибка выполнения функции
        ...
        }
    break;
}
```

При передаче неверного параметра (например, недопустимого описателя) *WaitForSingleObject* возвращает WAIT_EAILED. Чтобы выяснить конкретную причину ошибки, вызовите *GetLastError( )*.

Ожидание на множестве дескрипторов

```
DWORD WaitForMultipleObjects( DWORD dwCount,
CONST HANDLE* phObjects, BOOL fWaitAll, DWORD
dwMilliseconds);
```

Если *fWaitAll* приравнен FALSE, она возвращает управление, как только освобождается любой из объектов. В этом случае возвращается значение от WAIT_OBJECT_0 до WAIT_OBJECT_0 + *dwCount* - 1.

Пример ожидания на множестве из трех дескрипторов.

```
HANDLE h[3];
h[0] = hProcess1;
h[1] = hProcess2;
h[2] = hProcess3;

DWORD dw;
dw = WaitForMultipleObjects(3, h, FALSE, 5000);
switch (dw)
    {
    case WAIT_FAILED:{
        // неправильный вызов функции
        ...
        }
    break;
    case WAIT_TIMEOUT:{
        // ни один из объектов не освободился
        // в течение 5000 мс
        ...
        }
    break;
    case WAIT_OBJECTJ)_0:{
        // завершился процесс, идентифицируемый
        // h[0], т.е. описателем (hProcess1)
        ...
        }
    break;
    case WAIT_OBJECTJ)_0 + 1:{
        // завершился процесс, идентифицируемый
```

```
    // h[1], т.е. описателем (hProcess2)
    ...
    }
break;

case WAIT_OBJECTJ)_0 + 2:{
    // завершился процесс, идентифицируемый
    // h[2], т.е. описателем (hProcess3)
    ...
    }
break;
}
```

Синхронизирующие объекты

Критическая секция

```
Объект
 CRITICAL_SECTION CriticalSection;
-------------------------------------------------
Функции:
void WINAPI InitializeCriticalSection(
  __out         LPCRITICAL_SECTION lpCriticalSection
);

BOOL WINAPI InitializeCriticalSectionEx(
  __out         LPCRITICAL_SECTION lpCriticalSection,
  __in          DWORD dwSpinCount,
  __in          DWORD Flags
);

BOOL WINAPI InitializeCriticalSectionAndSpinCount(
  __in_out      LPCRITICAL_SECTION lpCriticalSection,
  __in          DWORD dwSpinCount
);
DWORD WINAPI SetCriticalSectionSpinCount(
  __in_out      LPCRITICAL_SECTION lpCriticalSection,
  __in          DWORD dwSpinCount
);

void WINAPI EnterCriticalSection(
  __in_out      LPCRITICAL_SECTION lpCriticalSection
);
```

```
void WINAPI LeaveCriticalSection(
  __in_out      LPCRITICAL_SECTION lpCriticalSection
);
BOOL WINAPI TryEnterCriticalSection(
  __in_out      LPCRITICAL_SECTION lpCriticalSection
);
void WINAPI DeleteCriticalSection(
  __in_out      LPCRITICAL_SECTION lpCriticalSection
);
```

Схема использования

```
const int MAX_TIMES = 1000;
int g_nIndex = 0;
DWORD g_dwTimes[MAX_TIMES];
CRITICAL_SECTION g_cs;
...

InitializeCriticalSection(g_cs);
...

DWORD WINAPI FirstThread(PVOID pvParam)
{
for (BOOL fContinue = TRUE; fContinue; )
     {
     EnterCriticalSection(&g_cs);
     if (g_nIndex < MAX_TIMES)
     {
     g_dwTimes[g_nIndex] = GetTickCount();
     g_nIndex++;
     }
     else
     fContinue = FALSE;
     LeaveCriticalSection(&g_cs);
     }
return(0);
}
```

Критический участок кода

```
DWORD WINAPI SecondThread(PVOID pvParam)
{
for (BOOL fContinue = TRUE; fContinue; )
     {
     EnterCriticalSection(&g_cs);
     if (g_nIndex < MAX_TIMES)
     {
     g_nIndex++;
     g_dwTimes[g_nIndex - 1] = GetTickCount();
     }
     else
     fContinue = FALSE;
     LeaveCriticalSecLion(&g_cs);
     }
return(0);
}
```

Критический участок кода

Объект ядра «Мьютекс»

```
HANDLE CreateMutex( PSECURITY_ATTRIBUTES psa,
BOOL fIniLialOwner, PCTSTR pszName);

HANDLE OpenMutex( DWORD fdwAccess, 800L
bInheritHandle, PCTSTR pszName);

BOOL ReleaseMutex(HANDLE hMutex);
```

Принадлежность мьютекса потоку.

Покинутый мьютекс - WAIT_ABANDONED

Мьютексы и критические секции

Мьютексы и критические секции одинаковы в том, как они влияют на планирование ждущих потоков, но различны по некоторым другим характеристикам. Сравнение объектов представлено в следующей таблице.

Таблица. Сравнение объектов критическая секция и мьютех

Характеристика	Мьютекс	Критическая секция
Быстродействие	Хуже	Лучше
Возможность использования за границами процесса	Да	Нет
Объявление	*HANDLE hmtx;*	*CRITICAL_SECTION cs;*
Инициализация	*hmtx = CreateMutex (NULL, FALSE, NULL);*	*InitializeCriticalSection(&cs);*
Очистка	*CloseHandle(hmtx);*	*DeleteCriticalSection(&cs);*
Бесконечное ожидание	*WaitForSingleObject (hmtx, INFINITE);*	*EnterCrittcalSection(&cs);*
Проверка без ожидания (ожидание в течение о мс)	*WaitForSingleObject (hmtx, 0);*	*TryEnterCriticalSection (&cs);*
Ожидание в течение заданного периода времени	*WaitForSingleObject (hmtx, dwMilliseconds);*	Невозможно
Освобождение	*ReleaseMutex(hmtx);*	*LeaveCriticalSecliun(&cs);*
Возможность ожидания многих объектов ядра	Да (с помощью *WaitForMultipleObjects*)	Нет

Объект ядра «Семафор»

```
HANDLE CreateSemaphore( PSECURITY_ATTRIBUTE psa,
LONG lInitialCount, LONG lMaximumCount, PCTSTR
pszName)

HANDLE OpenSemaphore( DWORD fdwAccess, BOOL
bInhentHandle, PCTSTR pszName);

HANDLE hSem = CreateSemaphore(NULL, 0, 5, NULL);

BOOL ReleaseSemaphore( HANDLE hSem, LONG
lReleaseCount, PLONG p]PreviousCount);
```

Объект ядра "событие"

Создается функцией *CreateEvent:*

```
HANDLE CreateEvent(
PSECURITY_ATTRIBUTES psa, BOOL fManualReset,
BOOL fInitialState, PCTSTR pszName);
```

Функции для работы с событием

```
HANDLE OpenEvent( DWORD fdwAccess, BOOL fInhent,
PCTSTR pszName);

BOOL SetEventT(HANDLE hEvenet);

BOOL ResetEvent(HANDLE hEvent);

BOOL PulseEvent(HANDLE hEvent);
```

ПРИМЕР **Три потока должны ожидать пока четвертый поток подготовит данные.**

```
// Для синхронизации работы потоков
// создаем глобальный дескриптор объекта
// типа событие со сбросом вручную
// (в занятом состоянии)

HANDLE g_hEvent;

int WINAPI WinMain( )
{

// создаем объект "событие со сбросом вручную (в занятом состоянии)
g_hEvent = CreateEvent(NULL, TRUE, FALSE, NULL);

// порождаем три новых потока

HANDLE hThread[3];

DWORD dwThreadID;
```

```
hThread[0] = _beginthreadex(NULL, 0,
    WordCount, NULL, 0, &dwThreadID);
hThread[1] = _beginthreadex(NULL, 0,
    SpellCheck, NULL, 0, &dwThreadID);
hTbread[2] = _beginthreadex(NULL, 0,
    GrammarCheck, NULL, 0, &dwThreadID);


// Читаем данные для обработки в потоках

    OpenFileAndReadContentsIntoMemory( );

// Устанавливаем объект  «Событие» в
// сигнальное состояние и тем самым
// разрешаем всем трем потокам обращаться
// к подготовленным данным памяти
...
SetEvent(g__hEvent);

...

}

// Функции потоков

DWORD WINAPI WordCount(PVOID pvParam)
{
// ждем, когда в память будут загружены данные
из файла
WaitForSingleObject(g_hEvent, iNeiNITE);

// обращаемся к блоку памяти
// и выполняем задачу
return(0);
}

DWORD WINAPI SpellCheck(PVOID pvParam)
{
// ждем, когда в память будут загружены данные
из файла
WaitForSingleObject(g_hFvent, INFINITE);
```

```
// обращаемся к блоку памяти
// и выполняем задачу
return(0};
}

DWORD WINAPI GrammarCheck(PVOID pvParam)
{
// ждем, когда в память будут загружены данные
из файла
WaitForSingleObject(g_hFvent, INFINITE);

// обращаемся к блоку памяти
// и выполняем задачу
return(0);
}
```

Сводная таблица объектов, используемых для синхронизации потоков

В следующей таблице суммируются сведения о различных объектах ядра применительно к синхрониаации потоков.

Таблица. Использование объектов ядра для синхронизации потоков

Объект	**Находится в занятом состоянии, когда**	**Переходит в свободное состояние, когда**	**Побочный эффект успешного ожидания**
Процесс **Поток**	процесс еще активен поток еще активен	процесс завершается (*ExitProcess*, *TerminateProcess*) поток завершается (*ExitThread*, *TerminateThread*)	Нет Нет
Задание	время, выделенное заданию, еще не истекло	время, выделенное заданию, истекло	Нет
Файл	выдан запрос на ввод-вывод	завершено выполнение запроса на ввод-вывод	Нет
Консольный ВВОД	ввода нет	ввод есть	Нет
Уведомление об изменении файла	в файловой системе нет изменений	файловая система обнаруживает изменения	Сбрасывается в исходное состояние
Событие с автосбросом	вызывается *ResetEvent*, *PulseEvent* или ожидание успешно завершилось	вызывается *SetEvent* или *PulseEvent*	Сбрасывается в исходное состояние
Событие со сбросом вручную	вызывается *ResetEvent* или *PulseEvent*	вызывается *SetEvent* или *PulseEvent*	Нет
Ожидаемый таймер с автосбросом	вызывается *CancelWaitable - Timer* или ожидание	наступает время срабатывания (*SetWaitableTimer*)	Сбрасывается в исходное состояние

	успешно завершилось		
Ожидаемый таймер со сбросом вручную	вызывается *CancelWaitableTimer*	наступает время срабатывания (*SetWaitableTimef*)	Нет
Семафор	ожидание успешно завершилось	счетчик > 0 (*ReleaseSemaphore*)	Счетчик уменьшается на 1
Мьютекс	ожидание успешно завершилось	поток освобождает мьютекс (*ReleaseMutex*)	Передается потоку во владение
Критическая секция	ожидание успешно завершилось ((*Try*)*EnterCriticalSection*)	поток освобождает критическую секцию (LeaveCriticalSection)	