Novell Developer Kit

www.novell.com

NLM™ THREADS MANAGEMENT

October 11, 2006

Novell®

Legal Notices

Novell, Inc. makes no representations or warranties with respect to the contents or use of this documentation, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to revise this publication and to make changes to its content, at any time, without obligation to notify any person or entity of such revisions or changes.

Further, Novell, Inc. makes no representations or warranties with respect to any software, and specifically disclaims any express or implied warranties of merchantability or fitness for any particular purpose. Further, Novell, Inc. reserves the right to make changes to any and all parts of Novell software, at any time, without any obligation to notify any person or entity of such changes.

Any products or technical information provided under this Agreement may be subject to U.S. export controls and the trade laws of other countries. You agree to comply with all export control regulations and to obtain any required licenses or classification to export, re-export, or import deliverables. You agree not to export or re-export to entities on the current U.S. export exclusion lists or to any embargoed or terrorist countries as specified in the U.S. export laws. You agree to not use deliverables for prohibited nuclear, missile, or chemical biological weaponry end uses. Please refer to www.novell.com/info/exports/ for more information on exporting Novell software. Novell assumes no responsibility for your failure to obtain any necessary export approvals.

Novell, Inc. has intellectual property rights relating to technology embodied in the product that is described in this document. In particular, and without limitation, these intellectual property rights may include one or more of the U.S. patents listed at http://www.novell.com/company/legal/patents/ and one or more additional patents or pending patent applications in the U.S. and in other countries.

Novell, Inc.
404 Wyman Street, Suite 500
Waltham, MA 02451
U.S.A.
www.novell.com

Online Documentation: To access the online documentation for this and other Novell developer products, and to get updates, see developer.novell.com/ndk. To access online documentation for Novell products, see www.novell.com/documentation.

Novell Trademarks

AppNotes is a registered trademark of Novell, Inc.
AppTester is a registered trademark of Novell, Inc. in the United States.
ASM is a trademark of Novell, Inc.
Beagle is a trademark of Novell, Inc.
BorderManager is a registered trademark of Novell, Inc.
BrainShare is a registered service mark of Novell, Inc. in the United States and other countries.
C3PO is a trademark of Novell, Inc.
Certified Novell Engineer is a service mark of Novell, Inc.
Client32 is a trademark of Novell, Inc.
CNE is a registered service mark of Novell, Inc.
ConsoleOne is a registered trademark of Novell, Inc.
Controlled Access Printer is a trademark of Novell, Inc.
Custom 3rd-Party Object is a trademark of Novell, Inc.
DeveloperNet is a registered trademark of Novell, Inc., in the United States and other countries.
DirXML is a registered trademark of Novell, Inc.
eDirectory is a trademark of Novell, Inc.
Excelerator is a trademark of Novell, Inc.
exteNd is a trademark of Novell, Inc.
exteNd Director is a trademark of Novell, Inc.
exteNd Workbench is a trademark of Novell, Inc.
FAN-OUT FAILOVER is a trademark of Novell, Inc.
GroupWise is a registered trademark of Novell, Inc., in the United States and other countries.
Hardware Specific Module is a trademark of Novell, Inc.
Hot Fix is a trademark of Novell, Inc.
Hula is a trademark of Novell, Inc.
iChain is a registered trademark of Novell, Inc.
Internetwork Packet Exchange is a trademark of Novell, Inc.
IPX is a trademark of Novell, Inc.
IPX/SPX is a trademark of Novell, Inc.
jBroker is a trademark of Novell, Inc.
Link Support Layer is a trademark of Novell, Inc.
LSL is a trademark of Novell, Inc.
ManageWise is a registered trademark of Novell, Inc., in the United States and other countries.
Mirrored Server Link is a trademark of Novell, Inc.
Mono is a registered trademark of Novell, Inc.
MSL is a trademark of Novell, Inc.
My World is a registered trademark of Novell, Inc., in the United States.
NCP is a trademark of Novell, Inc.
NDPS is a registered trademark of Novell, Inc.
NDS is a registered trademark of Novell, Inc., in the United States and other countries.
NDS Manager is a trademark of Novell, Inc.
NE2000 is a trademark of Novell, Inc.
NetMail is a registered trademark of Novell, Inc.
NetWare is a registered trademark of Novell, Inc., in the United States and other countries.
NetWare/IP is a trademark of Novell, Inc.

NetWare Core Protocol is a trademark of Novell, Inc.
NetWare Loadable Module is a trademark of Novell, Inc.
NetWare Management Portal is a trademark of Novell, Inc.
NetWare Name Service is a trademark of Novell, Inc.
NetWare Peripheral Architecture is a trademark of Novell, Inc.
NetWare Requester is a trademark of Novell, Inc.
NetWare SFT and NetWare SFT III are trademarks of Novell, Inc.
NetWare SQL is a trademark of Novell, Inc.
NetWire is a registered service mark of Novell, Inc., in the United States and other countries.
NLM is a trademark of Novell, Inc.
NMAS is a trademark of Novell, Inc.
NMS is a trademark of Novell, Inc.
Novell is a registered trademark of Novell, Inc., in the United States and other countries.
Novell Application Launcher is a trademark of Novell, Inc.
Novell Authorized Service Center is a service mark of Novell, Inc.
Novell Certificate Server is a trademark of Novell, Inc.
Novell Client is a trademark of Novell, Inc.
Novell Cluster Services is a trademark of Novell, Inc.
Novell Directory Services is a registered trademark of Novell, Inc.
Novell Distributed Print Services is a trademark of Novell, Inc.
Novell iFolder is a registered trademark of Novell, Inc.
Novell Labs is a trademark of Novell, Inc.
Novell SecretStore is a registered trademark of Novell, Inc.
Novell Security Attributes is a trademark of Novell, Inc.
Novell Storage Services is a trademark of Novell, Inc.
Novell, Yes, Tested & Approved logo is a trademark of Novell, Inc.
Nsure is a registered trademark of Novell, Inc.
Nterprise is a registered trademark of Novell, Inc., in the United States.
Nterprise Branch Office is a trademark of Novell, Inc.
ODI is a trademark of Novell, Inc.
Open Data-Link Interface is a trademark of Novell, Inc.
Packet Burst is a trademark of Novell, Inc.
PartnerNet is a registered service mark of Novell, Inc., in the United States and other countries.
Printer Agent is a trademark of Novell, Inc.
QuickFinder is a trademark of Novell, Inc.
Red Box is a trademark of Novell, Inc.
Red Carpet is a registered trademark of Novell, Inc., in the United States and other countries.
Sequenced Packet Exchange is a trademark of Novell, Inc.
SFT and SFT III are trademarks of Novell, Inc.
SPX is a trademark of Novell, Inc.
Storage Management Services is a trademark of Novell, Inc.
SUSE is a registered trademark of Novell, Inc., in the United States and other countries.
System V is a trademark of Novell, Inc.
Topology Specific Module is a trademark of Novell, Inc.
Transaction Tracking System is a trademark of Novell, Inc.
TSM is a trademark of Novell, Inc.

TTS is a trademark of Novell, Inc.

Universal Component System is a registered trademark of Novell, Inc.

Virtual Loadable Module is a trademark of Novell, Inc.

VLM is a trademark of Novell, Inc.

Yes Certified is a trademark of Novell, Inc.

ZENworks is a registered trademark of Novell, Inc., in the United States and other countries.

Third-Party Materials

All third-party trademarks are the property of their respective owners.

Contents

About This Guide

This documentation explains functions used for developing applications that make use of classical NetWare® threads.

IMPORTANT: Access to functionality in the multi-processor kernel of the NetWare 5.x and 6.x OS is provided through the Libraries for C (LiBC) (http://developer.novell.com/ndk/libc.htm). LibC provides a full set of mutex, reader-writer lock, condition variable, and semaphore API functions. It also provides functions for management of virtual machines (VMs), context and threads, general file and directory, and both physical and virtual memory.

This guide contains the following sections:

Feedback

We want to hear your comments and suggestions about this manual and the other documentation included with this product. Please use the User Comments feature at the bottom of each page of the online documentation.

Documentation Updates

For the most recent version of this guide, see NLM and NetWare Libraries for C (including CLIB and XPlat) (http://developer.novell.com/ndk/clib.htm).

Additional Information

For information about other CLib and XPlat interfaces, see the following guides:

- *NDK: NLM Development Concepts, Tools, and Functions*
- *NDK: Program Management*
- *NDK: Connection, Message, and NCP Extensions*
- *NDK: Multiple and Inter-File Services*
- *NDK: Single and Intra-File Services*
- *NDK: Volume Management*
- *NDK: Client Management*
- *NDK: Network Management*
- *NDK: Server Management*
- *NDK: Internationalization*
- *NDK: Unicode*
- *NDK: Sample Code*
- *NDK: Getting Started with NetWare Cross-Platform Libraries for C*
- *NDK: Bindery Management*

For CLib source code projects, visit Forge (http://forge.novell.com).

For help with CLib and XPlat problems or questions, visit the NLM and NetWare Libraries for C (including CLIB and XPlat) Developer Support Forums (http://developer.novell.com/ndk/devforums.htm). There are two for NLM development (XPlat and CLib) and one for Windows XPlat development.

Documentation Conventions

In this documentation, a greater-than symbol (>) is used to separate actions within a step and items within a cross-reference path.

A trademark symbol (®, ™, etc.) denotes a Novell trademark. An asterisk (*) denotes a third-party trademark.

Threads Concepts

1

This documentation describes the NetWare® Threads API, its functions, and features.

1.1 Threads

A *thread* is a stream of control that can execute its instructions independently. In the past, threads have sometimes inaccurately been referred to as NetWare processes. Process is more properly a concept used in UNIX* or Windows* NT. For operating systems in which the kernel does not support threads, a process is the unit of execution. In contrast, for a multithreaded system in which the kernel supports threads, a thread-rather than a process-is the unit of execution.

The NetWare OS allows NLM™ applications to establish multiple threads, each representing a single path of execution. An NLM usually contains at least one thread to accommodate the main (page 83) function. (This is not true if the NLM is a library, such as CLIB.NLM.)

Two or more threads can be running concurrently-simultaneously in the midst of code execution-although only one thread can have control of the CPU at any given time. Concurrent threads can be executing the same code or different code. In a multiprocessing evironment, two or more threads can also be running parallel-that is, simultaneously running on different processors. Parallel threads are of course also running concurrently.

Historically, the NetWare OS has been a nonpreemptive ("good guy") scheduling environment. (NLMs written for NetWare 5.x and 6.x can be marked preemtable, as explained below.) When a thread gains control of the CPU, the thread remains in control until it has run to the end of its execution or until it calls a function that 'blocks'-that is, relinquishes control of the CPU. (Blocking functions are identified in the function reference manuals.)

No other thread can interfere with an active thread, regardless of priority. Only a hardware interrupt can temporarily interrupt a currently running thread.

1.1.1 Thread Context

In the NetWare OS, many of the threads run in groups that are organized such that all threads in a group share data that is global to the group. The characteristics of that global data is collective called context. Context is one of the more important concepts to understand in programming to NetWare. For more information on context, see Section 1.4, "Context and Thread Groups," on page 16 and Section 1.5, "NetWare Global Data," on page 18 below.

For developers of driver, stack, and other lower-level code, please refer to Section 1.9, "Context and Development of Drivers, Stacks, etc.," on page 38

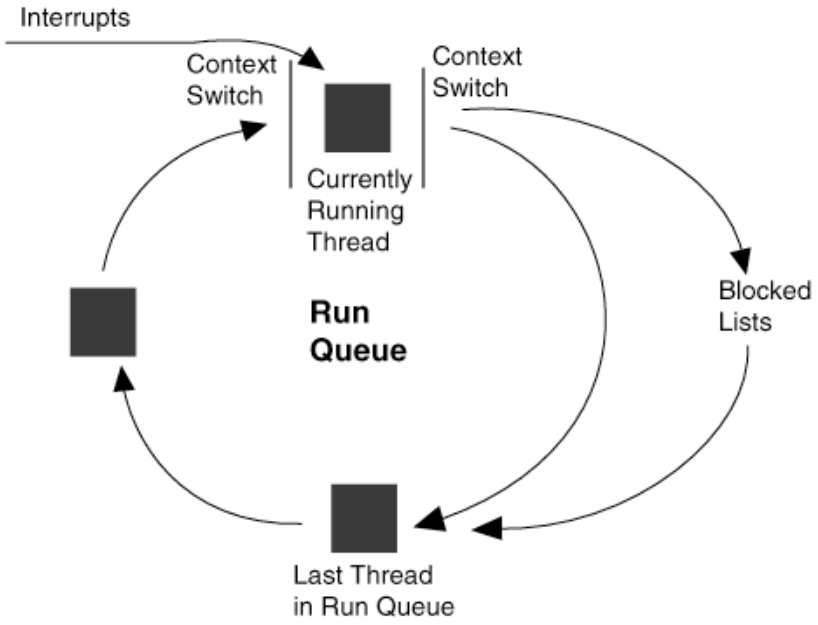
1.2 Thread Management

Although the NetWare OS has used threads extensively since version 3.x, the model for handling threads has changed considerably over time.

1.2.1 Thread Management in NetWare 3.x

The thread management for the NetWare 3.x OS is different from that of the NetWare 4.x, 5.x, and 6.x OS. The following figure illustrates thread management in the NetWare 3.x OS.

Figure 1-1 *NetWare 3.x Thread Management*



In the NetWare 3.x OS, threads waiting to be executed are placed in a Run Queue, which is serviced on a FIFO (first in, first out) order.

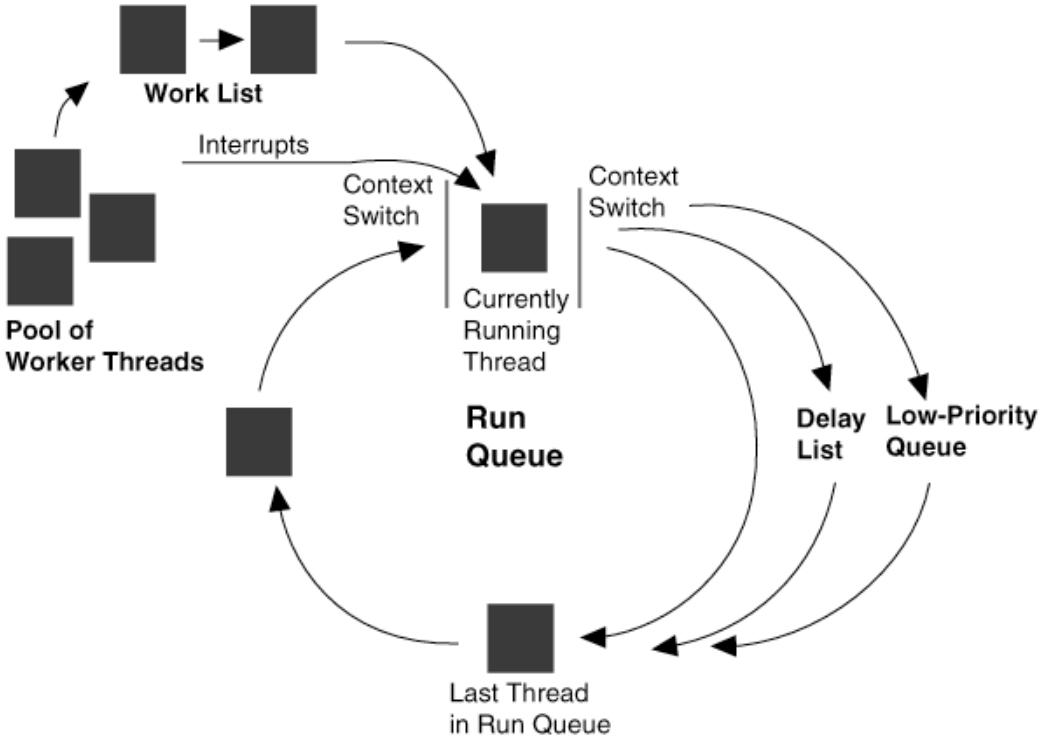
Threads waiting for resources to become available are placed in a blocked list, yielding the CPU to other threads in the Run Queue. When the needed resources become available, the thread moves from a blocked list to the end of the Run Queue.

Each time a thread becomes the current (executing) thread, or changes from the current thread to another state, a context switch occurs.

1.2.2 Thread Management in NetWare 4.x-6.x

The NetWare 4.0 OS introduced new features that add flexibility to thread management. The following figure illustrates thread management in the NetWare 4.x, 5.x, and 6.x OS.

Figure 1-2 *NetWare 4.x-6.x Thread Management*



The NetWare 4.0 OS introduced the concept of *work*. Each work unit has a routine and data associated with it, but the work unit is not a thread. To handle these work units, the OS reserves a pool of worker threads that are dedicated to running work.

When a work unit is scheduled, it is placed on the Work To Do List, and is serviced immediately after the current thread relinquishes the CPU. If the current thread is already a worker thread, the worker thread does not relinquish control; instead, it executes the next work unit, thereby avoiding a context switch.

Because worker threads avoid unnecessary context switching, single threads running many separate work units provide higher performance.

NOTE: Novell® recommends that work units be short, discrete routines that can complete quickly. If the work code calls a function that relinquishes control of the CPU, the worker thread is transformed into a regular thread.

Worker threads wait in their own pool. Regular (not worker) threads wait in the Run Queue, the Low-Priority Queue, or the Delay List, queues that are serviced on a FIFO order by the CPU.

The Work To Do List has the highest priority, followed by the Run Queue and the Low-Priority Queue. The Delay List has a variable priority status, and is usually serviced after the Low-Priority Queue has gained control of the CPU.

The Low-Priority Queue does not gain control of the CPU unless there is nothing else for the CPU to do. If a thread does "busy waiting" (looping while waiting for a resource to become available, for example), continually rescheduling itself on the Run Queue, the Low-Priority Queue cannot gain control of the CPU.

Following is an example of busy waiting:

```
while (!finished)
   ThreadSwitchWithDelay();
```

To allow the Low-Priority Queue to be serviced by the CPU, threads that do busy waiting should be rescheduled on the Delay List. After a thread has been scheduled on the Delay List, it waits for a number of context switches (50 is the default), then is placed at the end of the Run Queue. While threads wait in the Delay List, the Low-Priority Queue has a chance to be serviced by the CPU.

As in NetWare 3.x, each time a thread becomes the current thread, or changes from the current thread to another state, a context switch occurs.

For more information about programming with threads, see Section 1.7, “Multithreaded Programming,” on page 25.

1.3 Routine Scheduling by Thread Type

Threads can be scheduled either as normal threads or as work. To decide which type is appropriate for a given thread, consider the following sections.

NOTE: Work is unique to the 4.x, 5.x, and 6.x OS. If your NLM is going to run on 3.x servers, you cannot schedule threads as work.

1.3.1 When to Schedule a Routine as a Thread

The following conditions serve as guidelines for when to schedule a routine as a thread:

- The routine is a long-term process
- It needs a very large stack
- It needs to deliberately handicap itself temporarily to avoid *spin-waiting* (being rescheduled while waiting for something needed to complete execution).

If a routine is a long-term process, little benefit results from scheduling it as work because work that yields cannot be rescheduled as work. Instead, it is rescheduled on the Run Queue at the same priority as a normal thread.

All work is given a single stack size; but you can specify a stack size for threads. If you need to specify the stack size, you must schedule your routine as a thread.

If your routine is a polling process or one that does spin-waiting, you should schedule it as a thread.

In general, if your NLM already uses the NetWare 3.x process-scheduling scheme-which can still be carried out in the NetWare 4.x, 5.x, and 6.x kernel-and its routines are mostly long-term, continue to schedule the routines as threads. But if any of the routines are short-term, you can reschedule them as work.

NOTE: If your NLM is going to run in the NetWare 3.x environment as well as in the NetWare 4.x, 5.x, and 6.x environment, you cannot schedule any threads as work, since work does not exist in the NetWare 3.x environment. Schedule all threads then as normal threads.

Changing Thread Priority: It is possible to change the priority of a thread in one of ways described in the following sections.

Permanently Handicapping Threads

(NetWare 4.x, 5.x, and 6.x) If a particular NLM does not yield often enough, the OS places a handicap in the NLM thread's process control block (PCB), which prevents the thread from being rescheduled immediately. For example, it the OS places a handicap of 100 on a thread, 100 other pieces of work or threads run and yield before the handicapped thread is rescheduled in the RunList Queue.

A thread can also handicap itself by calling SetThreadHandicap.

Temporarily Handicapping Threads

(NetWare 3.x, 4.x, 5.x, and 6.x) If a thread needs a resource that will not be ready for a moment, but you do not want it to assume the overhead of sleeping on a semaphore or doing busy waiting, you can have the thread reschedule itself with a temporary handicap using ThreadSwitchWithDelay.

An example of busy waiting is the following:

```
while (!finished)
   ThreadSwitchWithDelay();
```

Temporarily handicapped threads are not placed in the Run Queue until their handicap has expired. Upon expiration, they are rescheduled at the end of the Run Queue. Letting threads temporarily handicap themselves prevents needless rescheduling overhead caused by a busy-waiting condition.

Temporarily handicapping threads is an issue for NetWare 4.x, 5.x, adn 6.x OS since the Low-Priority Queue does not gain control of the CPU unless there is nothing else for the CPU to do. If a thread does "busy waiting", continually rescheduling itself on the Run Queue (by using ThreadSwitch), the Low-Priority Queue cannot gain control of the CPU.

Low Priority Threads

(NetWare 4.x, 5.x, and 6.x) Low priority threads run when there is nothing to run except hardware polling routines and temporarily handicapped threads. Programs that might be candidates for low priority threads are file compression utilities, once-a-week backup, and cleanup utilities.

A thread can reschedule itself as a low priority thread by calling ThreadSwitchLowPriority.

To maintain a thread as low priority when you relinquish control, you must use ThreadSwitchLowPriority. If you use ThreadSwitch, the thread becomes a regular thread.

1.3.2 When to Schedule a Routine as Work

The following criteria can help determine when you should schedule a routine as work:

- The routine has a high priority.

- It needs to gain control of the processor quickly, that is, get in and run with as little scheduling overhead as possible.

Examples

A database request routine needs to gain access to the processor quickly and does not need to yield before it is completed. The performance of the database would be enhanced by scheduling requests as *work*, so that the work is serviced quickly and the CPU is relinquished to the next thread.

Similarly, scheduling end-of-routine cleanup as work enhances the operation of all threads in the kernel. A task such as freeing up the stack needs to be executed immediately after a routine ends, and is quick.

Other candidates for work are service routines, which check and update an item regularly. A service routine that updates object information for network management, for example, could be scheduled as work.

In general, NLM applications benefit when lower-level services are scheduled as work. For example, repetitive services such as disk reads could be scheduled as work. NetWare typically does a lot of disk reading. The entire operation usually completes without yielding because the data is found in cache memory.

1.4 Context and Thread Groups

The abstractions of context and thread groups are unique to NetWare. These abstractions allow for various kinds of efficiency in the way threads perform, but they also require understanding of certain key concepts to avoid the pitfalls associated with that efficiency. Many if not most of the threads in an NLM require access to important global data. That global data can belong to the thread itself, to the thread group to which the thread belongs, or to an entire NLM. Of particular importance are the data associated with a thread itself and the data associated with its thread group. This thread group data is also called CLib context, and access to that data is essential to most of the threads created by an NLM in the execution of an NLM developer's code. For more detail, refer to Section 1.5, “NetWare Global Data,” on page 18 and Section 1.8, “Context,” on page 32 below.

Each NLM can have more than one thread group, and each thread group may consist of one or more threads, as defined by the programmer. When an NLM is started, it has one thread group that includes the thread that executes the user-supplied main function.

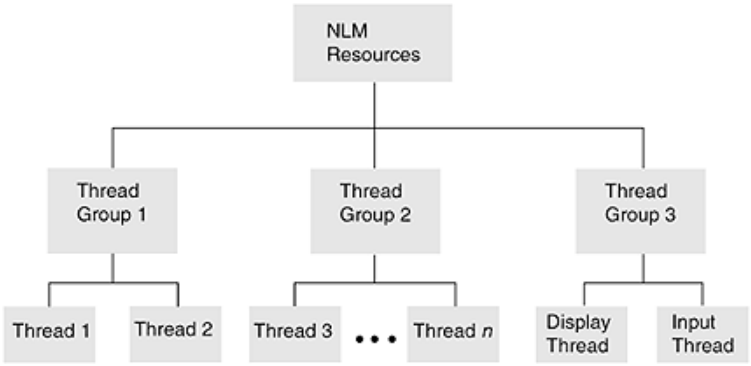
Threads are created by the NetWare® API in four ways, which determine the thread group:

- By default, a thread is started at the function main (page 83). This thread belongs to a default thread group.
- BeginThread (page 47) is called, creating a new thread that belongs to the current thread group.
- BeginThreadGroup (page 49) is called, creating a new thread group with one new thread belonging to it.
- ScheduleWorkToDo (page 98) is called, creating a new thread that belongs to the current thread group. ScheduleWorkToDo is works on the NetWare 4.x, 5.x, and 6.x OS.

The following figure shows a sample multithreaded NLM configuration. Threads 1 and 2 belong to the same group, Thread Group 1. All other numbered threads belong to Thread Group 2. This means threads 1 and 2 share the same thread group level context information (where their CWD could be \MYDIR1) and threads 3 through `n` share a different thread group level context (where their CWD could be \MYDIR2). Developers must understand that when there is more than one thread in a

thread group, changing the context (such as CWD) for one thread changes the context for all of the threads in the group. For example, if thread 3 changes its CWD, it also changes the CWD of threads 4 through n.

Figure 1-3 *Multithreaded NLM*



Because the display and input threads work together to handle server commands, the two threads have been assigned to the same thread group. This allows them to share the current working directory and current screen, among other resources.

1.4.1 Creating and Terminating Threads

The BeginThread function creates a thread. A thread can terminate itself using the ExitThread function as follows:

```
ExitThread(EXIT_THREAD, ...)
ExitThread(TSR_THREAD, ...)
```

A return statement from the original function (the function that was started by BeginThread) also terminates the thread.

1.4.2 Creating and Terminating Thread Groups

A single thread or multiple threads can be grouped to have a unique context.

The BeginThreadGroup function creates a thread group. A thread group can be terminated using the *ExitThread* function as follows:

```
ExitThread(EXIT_THREAD, ...)
in the last thread in the group
ExitThread(TSR_THREAD, ...)
in the last thread in the group,
```

A return statement from the original function (the function that was started by BeginThread) in the last thread in the group also terminates the thread group.

1.4.3 Interprocess Synchronization

Use local semaphores to control finite resources, to synchronize execution among threads, or to queue threads that need to use critical code sections. A semaphore has an associated signed 32-bit value.

Local semaphores can be used only by NLM applications running on a particular server (as opposed to network semaphores, which can be used by all NLM applications executing on, and all workstations attached to, the server).

The following Execution Thread functions deal with local semaphores:

- CloseLocalSemaphore
- ExamineLocalSemaphore
- OpenLocalSemaphore
- SignalLocalSemaphore
- WaitOnLocalSemaphore

The OpenLocalSemaphore function allocates a semaphore and gives the NLM access (a handle) to it.

A thread can use the WaitOnLocalSemaphore call to gain access to the associated resource or to wait for the resource to become available. WaitOnLocalSemaphore can also be used to cause one thread to wait for another thread to signal it to continue. WaitOnLocalSemaphore decrements the semaphore's associated value.

When a thread is finished using a semaphore's resource, it typically calls SignalLocalSemaphore to increment the semaphore's value. A thread can also use this function to cause a thread that is waiting on a semaphore to resume execution. SignalLocalSemaphore increments the semaphore's associated value.

The ExamineLocalSemaphore call allows a thread to retrieve a semaphore's value. The semaphore value can be positive or negative (from -2^{31} through $2^{31}-1$). A negative value means that one or more threads are waiting on the semaphore.

1.5 NetWare Global Data

Because NLM applications can vary widely in their design and purpose, the NetWare API maintains a variety of global data items for NLM applications. These data items are divided into the following categories described in the sections below:

- Thread Global Data
- Thread Group Global (CLib) Data
- NLM Global Data

These global data items can be set and queried by various functions in the NetWare API. Each category of global data items represents a different level of scope or context.

1.5.1 Thread Global Data

Each thread has its own set of data items. The data items are global only within that thread. That is, they have separate values for each thread. The data items of one thread cannot be referenced by another thread.

A thread is the lowest level within an NLM, and its context can consist of the following data items:

Data Item	Description
asctime, asctime_r char String Pointer	Only allocated if asctime is called. The asctime function returns a char *.
Critical Section Count	Contains the number of outstanding EnterCritSec (page 58) calls against a thread.
ctime, ctime_r, gmtime, gmtime_r, and localtime, localtime_r functions, tm structure	The ctime, ctime_r, gmtime, gmtime_r, and localtime, localetime_r functions return a pointer to a tm structure. Each thread has its own tm structure. The tm structure is allocated only if one of these three functions is called.
errno	Some functions set the `errno` return code to the last error code that was detected.
Last Value from the rand Function	Each thread has its own **seed** value (to start or continue a sequence of random numbers).
NetWareErrno	A NetWare specific error code. Some functions set both `NetWareErrno` and `errno`
stack	This points to the block of memory that BeginThread (page 47) allocated for the thread's stack.
strtok Pointer	The strtok function maintains a pointer into the string being parsed.
t_errno	Used with Transport Level Interface (TLI) functions. chapter).
Thread Custom Data Area Pointer and Size	The `threadCustomDataPtr` points to space that the NetWare API allocates to be associated with an individual thread. The `threadCustomDataPtrSize` variable specifies the size (in bytes) of this data.
Suspend Count	This count contains the number of outstanding SuspendThread (page 119) calls against a thread.

1.5.2 Thread Group Global Data

One instance of the following data items exists for each thread group. This collection of data effectively makes up what is caled CLib context. Any change that one thread makes to the value of a thread group global data item affects all the threads in the group. All threads within a thread group share the same thread group context, and require this context data to for proper execution. This fact makes it important not to tear down the data structures assoicated with a thread group until all threads in the group have cleaned up the resources individually allocated to them and until thread group resoureces are cleaned up. This issue becomes centrally important at the time an NLM unloads, especially on the ULOAD command. For information and instructions on cleaning up threads and thread groups successfully for an UNLOAD termination, see Terminating an NLM in the Advanced NLM Tasks chapter of NDK: NLM Development Concepts, Tools, and Functions.

Data Item	Description
Current Connection	The current connection number is described in Connection Number and Task Management Concepts (*NDK: Connection, Message, and NCP Extensions*).

Data Item	Description
Current Screen	The current screen is the target of screen I/O functions (see Screen Handling).
Current Task	The current task number is described in Connection Number and Task Management Concepts (Connection, Message, and NCP Extensions).
CWD	Current working directory (see File System Concepts in Multiple and Inter-File Services).
Current User	The "current user" is the user context used in NDS™ functions.
signal Settings	Most signal handler functions are set by the signal function. (See signal (page 111) and raise (page 90).)
stdin, stdout, stderr	These data items are the second-level standard I/O handles (see Stream I/O Concepts (Single and Intra-File Services)).
Thread Group Custom Data Area Pointer and Size	The `threadGroupCustomDataPtr` points to space that the NetWare API allocates to be associated with a thread group. The `threadGroupCustomDataPtrSize` variable specifies the size (in bytes) of this data.
umask Flags	These flags are set by the umask function (see File System Concepts in Multiple and Inter-File Services).

1.5.3 NLM Global Data

These data items have only one value for the entire NLM. The data items are global to all the thread groups and threads in the NLM. Any changes made to the values of NLM global data items affect all the thread groups and threads in the NLM.

Data Item	Description
active "Advertisers"	Each NLM may have a set of active "advertisers" (started by AdvertiseService (NLM)(unsupported)).
argv Array	This is the `argv` array passed to main.
atexit, AtUnload	Registers functions that are to be called when the NLM exits normally or is unloaded.
Libraries' Work Areas Pointers	Pointers to the data areas of any NLM libraries that the NLM has called (see Library Concepts (NDK: Program Management) for more information on library work areas).
locale Settings	Used by the locale functions.
Open Directories	The set of directories (opened by opendir) that the NLM has opened.
Open IPX/SPX Sockets	The set of IPX/SPX™ sockets that the NLM has opened.
Open Files	The set of files the NLM has opened. First-level open files include those opened with open, sopen, creat; second-level files include those opened with fopen, fdopen, freopen.

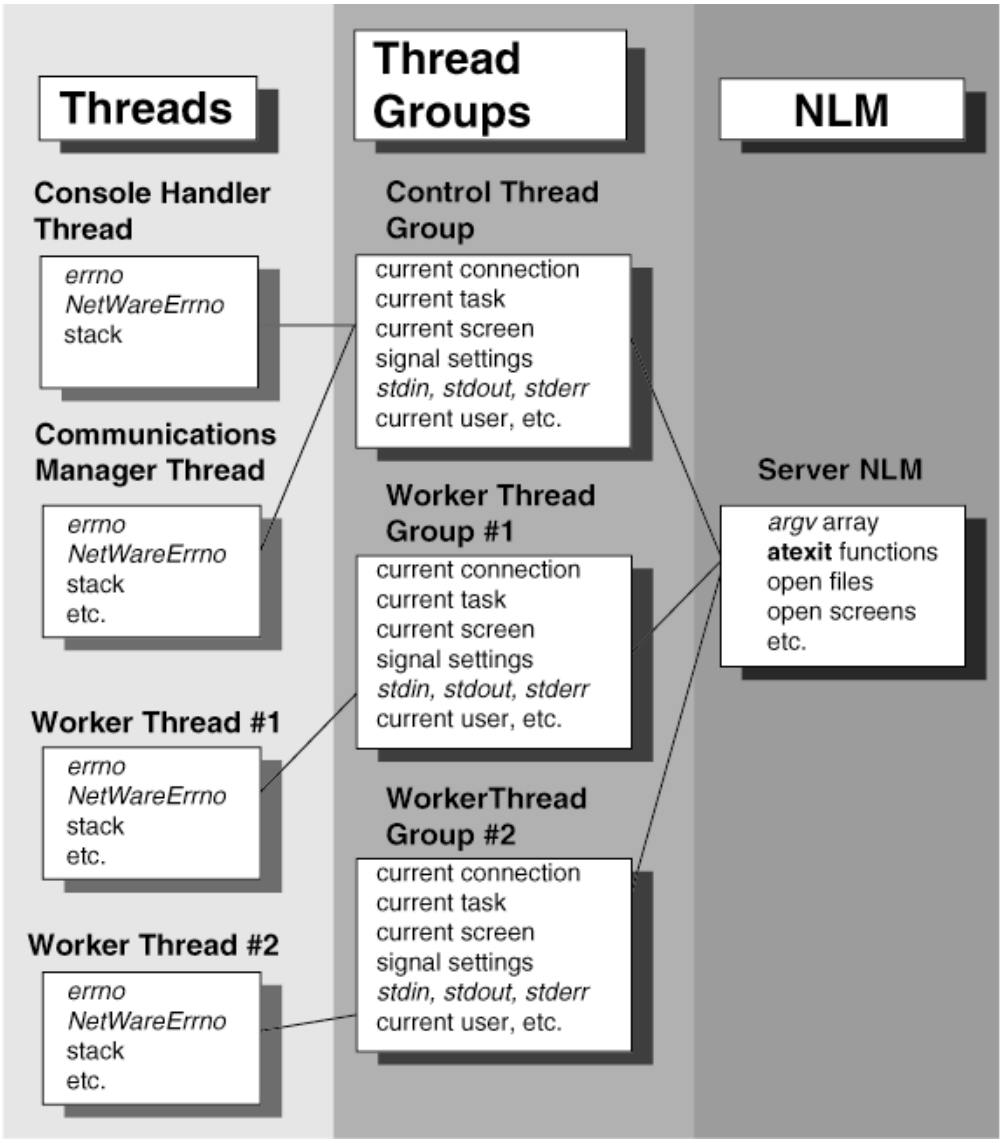
Data Item	Description
Open Network Semaphores	The set of network semaphores (opened by OpenSemaphore) the NLM has opened.
Open Screens	The set of screens the NLM has opened.
Original Command Line	A copy of the original command line that was entered when the NLM was started is saved (used by getcmd (page 68)).
Resource Tag	A tag used whenever the NLM allocates memory.
signal Settings	The SIGTERM constant that triggers a call to the signal handler in NLM termination.
Thread Name	Pattern used for naming new threads (used by BeginThread (page 47) and BeginThreadGroup (page 49)).

1.5.4 Hierarchy of Global Data

The different scope levels of the global data items in the NetWare API form a three-tier hierarchy. The thread global data items comprise the bottom level of the hierarchy. At the middle level are the thread group global data items. The top level of the hierarchy includes NLM global data items.

For example, consider the organization of a hypothetical NLM that services requests from multiple clients, as shown in the following figure.

Figure 1-4 *Hierarchy of Global Data Items*



Bottom Level-Thread Global Data Items: Each of the four threads has its own set of global data items. The global data items in one thread cannot be referenced by any of the other threads.

Middle Level-Thread Group Global Data Items: The global data items for the Control Thread Group are common to two threads: the Console Handler Thread and the Communications Manager Thread. The global items for the worker Thread Group #1 can be referenced only by worker Thread #1. The global items for the worker Thread Group #2 can be referenced only by Worker Thread #2.

Top Level-NLM Global Data Items: The global data items for the Server NLM are common to all the thread groups and threads.

1.5.5 NetWare 4.x-6.x Global Data

The NetWare API for NetWare 4.x, 5.x, and 6.x has added custom data areas to the thread and thread group contexts. These custom data areas can be referenced with the following variables:

Variable	Description
`threadCustomDataPtr`	A void pointer that points to a data area always associated with the current thread. You can use this area to associate data with a thread.
`threadCustomDataSize`	The size (in bytes) of the data area pointed to by `threadCustomDataPtr`. This variable is a LONG.
`threadGroupCustomDataPtr`	A void pointer that points to a data area always associated with the current thread group. You can use this area to associate data with a thread group.
`threadGroupCustomDataSize`	The size (in bytes) of the data area pointed to by `threadGroupCustomDataPtr`. This variable is a LONG.

When a thread is running it can use the custom data associated with its thread and thread group. You can use these data areas to store in memory information associated with a thread or a thread group.

`threadCustomDataPtr` and `threadGroupCustomDataPtr` point to areas in memory that the NetWare API has set aside. Before using the data areas these pointers point to, you should check `threadCustomDataSize` and `threadGroupCustomDataSize` to see if the available space is sufficient. (These data areas may shrink with future versions of the OS.)

NOTE: You should not change these pointers to point to data that you have allocated. However, you can use the data areas to hold the addresses to data that you have allocated.

1.6 Thread Function List

Function	Purpose
abort	Terminates an NLM abnormally.
atexit	Creates a list of functions that are executed on a last-in, first-out basis when the NLM exits normally or is unloaded.
AtUnload	Registers a function that is called if the NLM is unloaded with the UNLOAD console command.
BeginThread	Initiates a new thread within the current thread group.
BeginThreadGroup	Establishes a new thread within a new thread group.
Breakpoint	Suspends NLM execution and causes a break into the NetWare Internal Debugger.
ClearNLMDontUnloadFlag	Sets a flag in the NLM header to allow the NLM to be unloaded with the UNLOAD console command.
delay	Suspends execution for an interval (milliseconds).

Function	Purpose
EnterCritSec	Prevents all other threads in the NLM from being scheduled.
exit	Causes the NLM to terminate normally.
exit	Terminates the NLM without executing atexit functions or flushing buffers.
ExitCritSec	Allows other threads in the NLM to run.
ExitThread	Terminates either the current thread or the NLM.
FindNLMHandle	Returns the handle of a loaded NLM.
getcmd	Returns the command line it its original format (unparsed).
getenv	Searches the environment area for the environment variable and returns its value (presently environment variables are not supported).
GetNLMHandle	Returns the handle of the current NLM.
GetNLMID	Returns the ID of the current NLM.
GetNLMNameFromNLMID	Returns the name of an NLM.
GetThreadContextSpecifier	Returns the CLIB context used by callback routines scheduled by the specified thread.
GetThreadGroupID	Returns the ID of the current thread group.
GetThreadHandicap	Gets the number of context switches a thread is delayed before being rescheduled.
GetThreadID	Returns the thread ID of the current thread.
GetThreadName	Returns the name of a thread.
longjmp	Restores a saved environment.
main	A developer-supplied function where NLM execution begins.
MapNLMIDToHandle	Returns the handle associated with the NLM ID.
raise	Sends a signal to the executing program.
RenameThread	Renames a thread.
ResumeThread	Allows a previously suspended thread to run.
RetunNLMVersionInfoFromFile	Returns version information for a loaded NLM that corresponds to the specified file.
ReturnNLMVersionInformation	Returns version information for a loaded NLM that corresponds to a specified NLM handle.
ScheduleWorkToDo	Schedules a routine as work, which puts it on the highest priority queue (NetWare 4.x, 5.x, and 6.x).
setjmp	Saves its calling environment in its jmp_buf for subsequent use by longjmp.
SetNLMDontUnloadFlag	Sets a flag in the NLM header to prevent the NLM from being unloaded with the UNLOAD console command.

Function	Purpose
SetNLMID	Changes the current NLM.
SetThreadContextSpecifier	Determines the CLIB context that is used by all callback routines scheduled by the specified thread (NetWare 4.x, 5.x, and 6.x).
SetThreadGroupID	Changes the current thread group.
SetThreadHandicap	Sets the number of context switches a thread is permanently handicapped (delayed) before being rescheduled (NetWare 4.x, 5.x, and 6.x).
signal	Specifies an action to take place when certain conditions are detected (signalled).
spawnlp, spawnvp	Creates and executes a new child process.
SuspendThread	Prevents a thread from being scheduled.
system	Used to execute OS commands.
ThreadSwitch	Allows other threads a chance to run, where no natural break in the running thread would normally occur.
ThreadSwitchLowPriority	Reschedules a thread onto the low-priority queue (NetWare 4.x, 5.x, and 6.x).
ThreadSwitchWithDelay	Reschedules a thread to be place on the RunList after a specified number of switches have taken place (NetWare 4.x, 5.x, and 6.x).
Local Semaphore Functions:	
CloseLocalSemaphore	Closes a local semaphore.
ExamineLocalSemaphore	Returns the current value of a semaphore.
OpenLocalSemaphore	Allocates a local semaphore and gives the NLM access to it.
SignalLocalSemaphore	Increments a semaphore’s value.
TimedWaitOnLocalSemaphore	Waits on a local semaphore until it is signalled or the specified timeout elapses.
WaitOnLocalSemaphore	Decrements a semaphore’s value.
WaitOnLocalSemaphore	Decrements a semaphore’s value

1.7 Multithreaded Programming

Multithreading is common in multiclient distributed applications. Typically, a client-server NLM uses a different thread group for each client to which it provides service. This allows the NLM to service multiple clients concurrently. In addition, NLM applications often establish specialized threads, such as display, input, and communication threads. These threads can be used, for example, to accept commands from the server console, receive incoming requests, and send outgoing replies.

An efficient way to handle multiple clients is for the NLM to create a new thread for each client it services. As the NLM receives client requests, it creates a new thread to process each request. Then, after the request is serviced, the thread runs to the end of its initial procedure and is terminated.

There are cases where the method mentioned above would not be efficient. For example, if you are servicing 250 users and have 250 threads with 8 KB stacks, then just the stacks of these threads take up 2,000 KB of memory. In this case you might want to establish a pool of threads to handle multiple clients. As the NLM receives client requests, it selects a free thread to process the request. After the thread processes the request, it returns to the pool of free threads.

Using multiple threads has many advantages. It allows you to:

Simplify code through modularization

By separating processes into threads, programs become more easily read, maintained, and updated. Multithreading relieves the developer of having to use task switching logic.

Increase throughput

By dividing the NLM into multiple threads, you can reduce the amount of time the CPU remains idle. Instead of blocking during I/O requests, the OS switches control to another thread and more fully utilizes the CPU.

Enhance response time

Because the server is able to switch between threads (thus preventing a single thread from monopolizing the CPU), clients receive faster replies to their requests. A lengthy I/O-intensive request from one workstation does not preclude the completion of a smaller request from another workstation.

Develop multiple contexts through thread grouping

A thread group consists of one or more threads, as defined by the programmer. Threads in the same thread group share the same context, such as the CWD and current connection. This provides the programmer with shortcuts, such as the ability to use the CWD instead of specifying the full pathname.

The advantages of a multithreaded application are increased performance and efficiency. In a multithreaded application, processes get more equal time to use system resources. Additionally, any thread can process separately from other threads. For example, a process can update files in the background while a foreground process produces data that needs to be written to those files. Similarly, one thread can be used to interact with a client process while performing complex, time-consuming computations in the background.

The following is a simple example showing the creation of multiple threads:

Creating Multiple Threads

```
#include <stdio.h>
#include <stdlib.h>
#include <process.h>


int   getOut = FALSE;
int   twoOut = FALSE;
int   threeOut = FALSE;
int   fourOut = FALSE;


void ThreadTwo();
void ThreadThree();
```

```
void ThreadFour(void *data);


main()
{
   BeginThreadGroup(ThreadThree, NULL, NULL, NULL);
   ThreadSwitch();

   BeginThread(ThreadTwo, NULL, NULL, NULL);
   ThreadSwitch();

   while (!kbhit())
      printf("Thread One.\n");
   getOut = TRUE;

   // allow all threads to clean up before NLM exits
   while (!(twoOut && threeOut && fourOut))
      ThreadSwitch();
}


void ThreadTwo()
{
   while (!getOut)
   {
      printf("           Thread Two\n");
      ThreadSwitch();
   }
   twoOut = TRUE;
}


void ThreadThree()
{
   BeginThread(ThreadFour, NULL, NULL, "THREAD FOUR");
   while (!getOut)
   {
      printf("In Thread Three\n");
      ThreadSwitch();
   }
   threeOut = TRUE;
}


void ThreadFour(void *data)
{
   while (!getOut)
   {
      printf("            %s\n",(char *) data);
      ThreadSwitch();
   }
   fourOut = TRUE;
}
```

See Threads Concepts for more information about threads, thread groups, and the context that the NetWare API maintains. The Section 1.8, “Context,” on page 32 also discusses context issues.

1.7.1 Shared Memory

For true reentrant NLM programming functionality, see the Libraries for C (LibC) (http://developer.novell.com/ndk/libc.htm). CLib does not truly support this functionality—not without considerable resource leaks. For example, before NetWare 4.11 (when CLib started automatically cleaning up allocated semaphores), an abend occurs if you fail to manually deallocate semaphores. Plus, whenever CLib is unloaded, it almost always sends messages about unfreed memory resources.

However, LibC was designed to support reentrant NLM programming and the use of the REENTRANT flag. LibC's support of this functionality is mainly due to its context architecture and its ability to react to all penetrations of the library.

Shared memory allows multiple threads to communicate. To share memory among threads in the same NLM, use a global or static pointer to a single block of memory. The following example uses a global pointer to share memory among thread groups in the same NLM:

Using a Global Pointer to Share Memory among Thread Groups

```
int SharedMemoryFlag = 0;
char *SharedMemory;

void ThreadGroup2()
{
   while (!SharedMemoryFlag)
      ThreadSwitch();
   strcpy (SharedMemory,
      "ThreadGroup2 has accessed shared memory.");
   SharedMemoryFlag = 0;
}

main()
{
   /* Start the second thread group */
   if (BeginThreadGroup (ThreadGroup2,0,0,0) == EFAILURE)
   {
      printf ("BeginThreadGroup failed.\n");
      exit(0);
   }

/* Allocate the memory to be shared. Note that SharedMemory
 * could have been defined as an array, if desired. */
   SharedMemory = malloc (100);
   if (SharedMemory == NULL)
   {
      printf ("Could not allocate memory.\n");
      exit(0);
   }

/* Store a string in the allocated memory and print it */
```

```
strcpy (SharedMemory, "Main ThreadGroup has accessed shared memory.");
printf ("%s\n",SharedMemory);

/* Let ThreadGroup2 know it is OK to access the memory */
SharedMemoryFlag = 1;

/* Wait for ThreadGroup2 to access the memory */
while (SharedMemoryFlag)
   ThreadSwitch();

/* Print the message stored by ThreadGroup2 */
printf ("%s\n",SharedMemory);
free (SharedMemory);
}
```

If you want to use shared memory with multiple NLM applications, write a function that passes the memory address pointer among the modules. The following example shows the first NLM setting up values to share with the second NLM:

Setting up Values to Share Memory with Another NLM

```
/*  ── FIRST NLM ──  *
 * This NLM must be loaded first. Its .LNK file    *
 * exports the two shared values, SharedMemoryFlag *
 * and SharedMemory.                               *
 *  ──────- */
int SharedMemoryFlag = 0;
char *SharedMemory;

main()
{
   /* Allocate the memory to be shared. Note that
    * SharedMemory could have been defined as an array,
    * if desired. */
   SharedMemory = malloc (100);
   if (SharedMemory == NULL)
   {
      printf ("Could not allocate memory.\n");
      exit(0);
   }

   /* Store a string in the allocated memory and print it */
   strcpy (SharedMemory,
      "The main NLM has accessed shared memory.");
   printf ("%s\n",SharedMemory);

   /* Let the other NLM know it is OK to access the
      memory */
   SharedMemoryFlag = 1;

   /* Wait for the other NLM to access the memory */
   while (SharedMemoryFlag)
   ThreadSwitch();
```

```
   /* Print the message stored by the other NLM */
   printf ("%s\n",SharedMemory);
   free (SharedMemory);
}
```

The following directive file should be used to link the first NLM:

```
form novell nlm 'Example of Shared Memory between NLM applications'
name     nlm1
file     prelude,nlm1
import   @clib.imp
export   SharedMemoryFlag, SharedMemory
```

The following example shows the second NLM using the shared values set up by the first NLM:

A Second NLM Sharing Memory with the First

```
/*  ── SECOND NLM ──  *
 * This NLM must be loaded after the first. Its     *
 * .LNK file imports the two shared values.         *
 *  ──────-  */
extern int SharedMemoryFlag = 0;
extern char *SharedMemory;

main()
{
   while (!SharedMemoryFlag)
      ThreadSwitch();
      strcpy (SharedMemory, "The second NLM has accessed shared
            memory.");
    SharedMemoryFlag = 0;
}
```

The following directive file should be used to link the second NLM:

```
form novell nlm 'Example of Shared Memory between NLM applications'
name     nlm2
file     prelude,nlm2
import   @clib.imp, SharedMemoryFlag, SharedMemory
```

NOTE: For the NetWare® 4.x, 5.x, and 6.x OS, NLM applications can only share memory with modules that are loaded in the same domain (address space for NetWare 5.x and 6.x). For example NLMs loaded into OS address space can share memory among themselves, as can NLMs loaded into a protected address space. However, NLMs in the OS address space cannot share memory with NLMs loaded into a protected address space.

1.7.2 Thread Termination

Programming successfully for the thread termination process, especially CLIB threads, is discussed in Advanced NLM Tasks within the NDK: NLM Development Concepts, Tools, and Functions documentation.

1.7.3 Relinquishing Control

The NetWare 5.x and 6.x OS offers optional preemption for applications written to be preemptable and marked preemptive with MPKXDC.EXE. However, with previous versions of the NetWare OS that do not timeslice or preempt thread execution, the responsibility of relinquishing control falls to the thread itself. To relinquish control of the processor in nonpreempting NetWare versions, a thread can do one of the following:

Call a function that can relinquish control

For example, if a thread calls printf, it can relinquish control because printf writes to a device. However, this method should not be used in a program that must be guaranteed that control is relinquished.

Functions that might block are identified in the function descriptions.

Call ThreadSwitch (page 122)

ThreadSwitch passes control of the CPU to the OS, which then passes control to the next thread in the run queue. The calling thread is placed at the end of the run queue.

Call delay (page 56) or ThreadSwitchWithDelay (page 124)

These functions suspend thread execution for a specified time (in milliseconds). ThreadSwitchWithDelay is can be used with the NetWare 4.x, 5.x, and 6.x OS, but it has been added to the 3.11 version of CLIB.

IMPORTANT: Threads that do busy waiting in NetWare 4.x, 5.x, and 6.x need to allow low priority threads to run. For this reason, these threads should call ThreadSwitchWithDelay, instead of ThreadSwitch. Low priority threads can only run when there are no threads waiting on the RunList, and ThreadSwitch places the threads that call it on the RunLIst. ThreadSwitchWithDelay places threads that call it on the DelayedList.

Call SuspendThread (page 119)

The SuspendThread function puts a thread to sleep until it is awakened.

NOTE: A sleeping thread can be awakened only by calling ResumeThread (page 92) from another thread.

Call ThreadSwitchLowPriority (page 123)

The ThreadSwitchLowPriority function suspends thread execution and places the thread in the Low-Priority Queue. This function can be used with the NetWare 4.x, 5.x, and 6.x OS, but it has not been added to the 3.11 version of CLIB.

Wait on an event

The OS automatically puts to sleep any threads waiting on events. For example, if a thread waits in a semaphore queue, it relinquishes control.

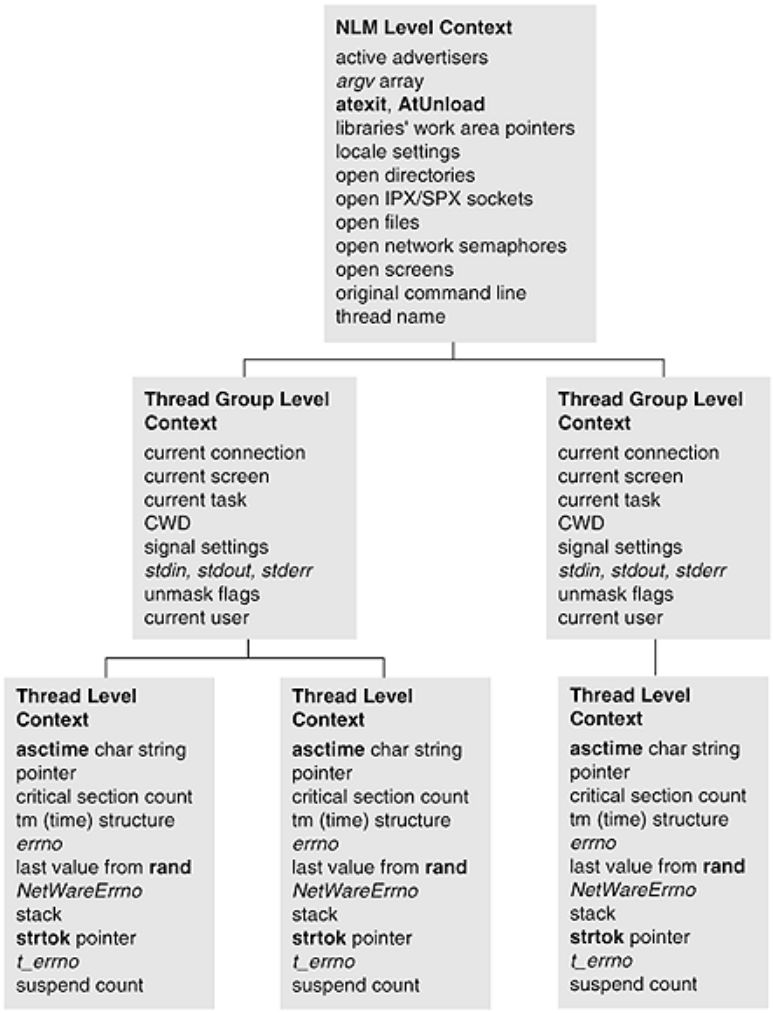
IMPORTANT: Do not use this method when waiting to read a file from a disk. If the file is stored in cache memory, the thread does not have to wait and does not relinquish control.

One side effect of failing to relinquish control is that incoming client requests are still received by the server, but the packets cannot be processed. Thus, without acknowledgment of the request, the client connection eventually times out.

1.8 Context

The NetWare API maintains a context for each NLM that is running. The context is divided into three levels of scope: thread level context, thread group level context, and NLM level context. (The following figure illustrates the three levels of context.) Because this context is created using the functions found in CLIB.NLM, it is commonly known as CLIB context.

Figure 1-5 *Levels of NLM Context*



NOTE: An understanding of context is critical to NLM development. Many errors in NLM applications are caused by developers not understanding context and how it can change.

Threads created in one of the four ways described in Section 1.4, “Context and Thread Groups,” on page 16 have the CLIB thread level, group level, and NLM level context. These context levels contain different information that is changed by the NetWare API. The context information cannot be changed directly by the programmer.

NOTE: A fifth way to create threads is for the OS to create threads. These threads do not have CLIB context, and must be given CLIB context. This issue is discussed after the following discussion about the context levels.

1.8.1 Thread Level Context

Thread level context is the most private level of context information within an NLM; the context of each thread is available only to each thread. These values are separate for each thread. The data items of one thread cannot be referenced by another thread.

Threads maintain the following context:

asctime char string pointer

This character string is only allocated if the asctime, asctime_r function is called. The asctime, asctime_r function returns a char *. (*NDK: Program Management*)

critical section count

This count contains the number of outstanding EnterCritSec calls against a thread.

ctime, ctime_r, gmtime, gmtime_r, and localtime, localtime_r tm structure pointer (*NDK: Program Management*)

The ctime, ctime_r, gmtime, gmtime_r, and localtime, localtime_r functions return a pointer to a tm structure. Each thread has its own tm structure. The tm structure is allocated only if one of these three functions is called.

`errno`

Some functions set the `errno` return code to the last error code that was detected.

Last value from the function rand

Each thread has its own seed value (to start or continue a sequence of random numbers). (See rand, rand_r (*NDK: Program Management*).)

`NetWareErrno`

This error code is a NetWare specific error code. Some functions set both `NetWareErrno` and `errno`.

Stack

This points to the block of memory BeginThread (page 47) allocated for the thread's stack.

strtok pointer

The strtok function maintains a pointer into the string being parsed (*NDK: Program Management*).

`t_errno`

This error code is used by TLI functions.

Suspend count

This count contains the number of outstanding SuspendThread calls against a thread.

1.8.2 Thread Group Level Context

All of the threads in a thread group share the same thread group level context. Any change that one thread makes to the value of a thread group data item affects all the threads in the group. The context of a thread group, however, is not shared with other thread groups, so changes within one thread group do not affect another group.

Thread groups maintain the following context:

Current connection

The current connection number is described in Connection Number and Task Management Concepts (*NDK: Connection, Message, and NCP Extensions*).

Current screen

The current screen is the target of screen I/O functions, as described in Screen Handling Concepts (*NDK: NLM Development Concepts, Tools, and Functions*).

Current task

The current task number is described in Connection Number and Task Management Concepts (*NDK: Connection, Message, and NCP Extensions*).

CWD

Current Working Directory (*NDK: NLM Development Concepts, Tools, and Functions*, as described in "OS Related Issues" in this documentation.

Signal settings

Signal handler functions are set by signal. (The signal and raise functions are discussed in Signal Handling in NDK: NLM Development Concepts, Tools, and Functions.)

`stdin`, `stdout`, `stderr`

These data items are the second-level standard I/O handles, as mentioned in Stream I/O Concepts (Single and Intra-File Services).

umask flags

These flags are set by the umask (Multiple and Inter-File Services) function.

Current user

The "current user" is the user context used by NDS™ functions (NetWare® 4.x, 5.x, and 6.x).

1.8.3 NLM Level Context

The NLM™ level context is shared by all thread groups and threads in the NLM, and these data items have only one value for the entire NLM. The data items are global to all the thread groups and threads in the NLM. Any changes made to the values of NLM global data items affect all the thread groups and threads in the NLM.

NLM applications maintain the following context on a per NLM basis:

"Active advertisers"

Each NLM can have a set of "active advertisers" (started by AdvertiseService (NLM) (unsupported)).

`argv` array

This is the `argv` array passed to main (page 83).

atexit (page 43), AtUnload (page 45) (which signal the SIGTERM handler)

These functions register functions that are to be called when the NLM exits normally or is unloaded.

Libraries' work areas pointers

These are pointers to the data areas of any NLM libraries that the NLM has called, as described in Library Concepts (*NDK: Program Management*).

Locale settings

These settings are used by the locale functions.

Open directories

The set of directories (opened by opendir) the NLM has open.

Open IPX/SPX/SPX II sockets

The set of IPX/SPX/SPX II sockets the NLM has open.

Open files

The set of files the NLM has open. First-level open files include those opened with open, sopen, and creat. Second-level files include those opened with fopen, fdopen, and freopen (Single and Intra-File Services).

Open network semaphores

The set of network semaphores (opened by NWOpenSemaphore (Single and Intra-File Services)) the NLM has open.

open screens

The set of screens the NLM has open.

original command line

A copy of the original command line when the NLM was started is saved (used by the getcmd function).

thread name

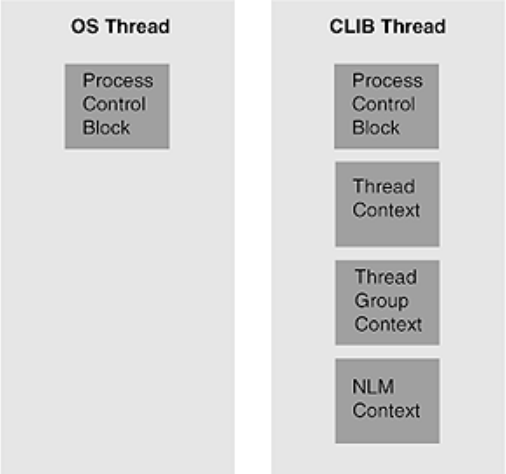
This pattern is used for naming new threads (used by BeginThread and BeginThreadGroup).

1.8.4 Context Problems with OS Threads

There are two types of threads running in the NetWare® OS: OS threads (such as callbacks) and CLIB threads (those created by the CLIB.NLM functions). The OS threads are created by the OS in instances such as when the LOAD and the UNLOAD commands are used. CLIB threads are created by calling the NetWare API functions BeginThread, ScheduleWorkToDo, and BeginThreadGroup,

and by a default thread starting at the main function. The following figure shows that OS threads are missing the context that CLIB threads have.

Figure 1-6 *Context of OS and CLIB Threads*



The problem here is that many-but not all-NetWare API functions need to have a context in order to work correctly. For example: printf writes to the calling thread's current screen. The current screen is kept in the thread's thread group context. OS threads do not have any CLIB context, so their calls to printf do not produce output anywhere. In more extreme cases, OS threads calling the NetWare API functions that need CLIB context can cause the server to abend.

NOTE: The solution to this problem is to give the OS threads context, thereby turning them into CLIB threads. The method for doing this is presented in the following section.

Developers must be aware of all the situations where NLMs will be running with OS threads instead of CLIB threads, and adjust their code accordingly to give the OS threads CLIB context. The following is a list of conditions where the NLM runs with OS threads:

- In the optional startup function that is specified with the "OPTION START" directive, when using the WLINK linker, or with the "START" directive for NLMLINK
- In the check function that is specified with the "OPTION CHECK" directive when using the WLINK linker, or with the "CHECK" directive for NLMLINK
- In the function registered with FERegisterNSPathParser (might not have the correct context)
- In the library cleanup function set by RegisterLibrary

In the following conditions, threads might or might not have CLIB context, depending on the context specifier:

- In functions registered with RegisterForEvent
- In functions registered with ScheduleSleepAESProcessEvent
- In functions registered with ScheduleNoSleepAESProcessEvent
- In the function registered with RegisterConsoleCommand (see Adding Console Commands: Example.)

1.8.5 Context Solutions for OS Threads

Two solutions to these context problems are as follows:

NetWare 3.11, 4.x, 5.x, and 6.x solution: read group ID of one of the groups, such as for the default thread group created for the main (page 83) function. (You might also want to create a global variable for each of the thread groups that are created.) The thread group ID of the current thread group can be obtained with GetThreadGroupID (page 78), as shown in the following example:

```
#include <process.h>
   int globalThreadGroupID;

   main()
   {
      globalThreadGroupID = GetThreadGroupID();
      ...
   }
```

Then, when you have an OS thread running, you give the OS thread context using SetThreadGroupID as follows:

```
oldTGID = SetThreadGroupID(globalThreadGroupID); /* do work */
...
SetThreadGroupID(oldTGID); /* always set back the thread group ID */
```

At this point, the NetWare® API takes the OS thread and gives it context, just as if it had been a CLIB thread. This lets you use the NetWare API functions that need context.

IMPORTANT: You must be careful when using the thread group ID that other threads are using. Changes to the context affect all threads in that group.

NetWare 4.x, 5.x, and 6.x solution: CLIB threads in the NetWare 4.x, 5.x, and 6.x OS have been given a context specifier that gives these threads the ability to automatically give context to callbacks (OS threads that are registered to be called to run when specific conditions occur) that they register. The context that is given to the callbacks when they are registered is determined by the setting of the registering thread's context specifier. The context specifier can be set to one of the following settings:

NO_CONTEXT

Use this when you don't want callbacks to be automatically registered with a CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context, the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

Call SetThreadGroupID (page 108) and pass in a valid thread group ID. Use this once inside your callback to give your callback thread CLIB context.

USE_CURRENT_CONTEXT

Use this to register callbacks to have the thread group context of the registering thread.

A valid thread group ID

Use this when you want the callbacks to have a different thread group context than the thread that schedules them.

You can determine the existing setting of the registering thread's context specifier by calling GetThreadContextSpecifier. Call SetThreadContextSpecifier to set a thread's context specifier.

When a new thread is started with BeginThread, BeginThreadGroup, or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

Using this solution, if you want the registered thread to have the thread group context of the registering thread, you would set the registering thread's context specifier to USE_CURRENT_CONTEXT (if it has been changed from the default) and then register the function that will run as a callback.

NOTE: The drawback to using this solution is that the context specifier is specific to the NetWare 4.x, 5.x, and 6.x OS. If you use this solution, your NLM will not run on the NetWare 3.11 OS.

1.9 Context and Development of Drivers, Stacks, etc.

Developing code for drivers, stacks, and similar low-lever funcionality requires you to use only functions that have no dependency on library context. If your code has this need, use only klib.imp to link into NDK interfaces. Functions of symbols thus imported are guaranteed to work in a classic NLM/CLib environment, in an NKS/LibC environment, or an a low-level environment that has no library context at all.

Threads Functions

2

This documentation alphabetically lists the threads functions and describes their purpose, syntax, parameters, and return values.

abort

Terminates an NLM™ application abnormally

Local Servers: blocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <assert.h>
#include <nwthread.h>
#include <stdlib.h>

void abort  (void);
```

Return Values

None

Remarks

This function causes the NLM to be terminated abnormally. It writes the following termination message to the System Console Screen:

```
ABNORMAL NLM TERMINATION in: NLMname
```

The abort function then raises SIGABRT and calls _exit (3).

The following sequence of events occurs when the NLM is terminated abnormally:

- All threads in the NLM are destroyed.
- Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library Concepts (*NDK: Program Management*).)
- All screens are closed.
- All first-level files (opened with open, sopen, create), including UNIX STREAMS, and also including files opened as a result of second-level I/O (opened with fopen, fdopen, freopen), are closed; however, the buffers of these are not flushed.
- All open directories are closed.
- All service advertising (started by AdvertiseService) is terminated.
- All memory allocated by the NLM is freed.
- The NLM is unloaded.

See Also

exit (page 61), _exit (page 62), ExitThread (page 65), raise (page 90)

Example

```
#include <assert.h>
#include <nwthread.h>
#include <stdlib.h>
#include <nwconio.h>

main()
{
   printf("this should print\r\n");
   getch();
   abort();
   printf("this should not print\r\n");
   getch();
}
```

atexit

(Designed for drivers) Creates a list of functions that are executed on a "last-in, first-out" basis when the NLM exits normally or is unloaded

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <nwthread.h>
#include <stdlib.h>

int  atexit  (
   void   (* func) (void));
```

Parameters

`func`

(IN) Points to the function to be registered as an exit function.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Registration was successful.
-1		EFAILURE	Registration failed (32 functions are already registered).

Remarks

WARNING: The atexit function is designed to be used by drivers. Do *not* use it as an NLM cleanup routine. A server abend may result for this reason:

CLIB context does not exist for a thread that is running an atexit or AtUnload routine, since all NLM thread groups have been destroyed by the time these functions are called. Thus any saved thread group ID is invalid, and neither atexit nor AtUnload routines can use SetThreadGroupID to establish CLIB context for the thread. They therefore also cannot use NetWare API functions that require thread group or thread level context.

It is wise programing practice to have one exit point for a program, which could be the cleanup routine for the NLM. The "Remarks" section of AtUnload (page 45) gives suggestions for developing a cleanup routine to be executed when an NLM is manually unloaded. The same routine can be used when exit is called from your program.

The atexit function is called when an NLM is terminated normally.

Successive calls to atexit create a list of functions that are executed on a "last-in, first-out" basis when:

- The NLM calls exit.
- The NLM calls ExitThread and it causes the NLM to be terminated.
- The last thread in the NLM returns from its original function.
- The NLM is unloaded with the UNLOAD command.

No more than 32 functions can be registered with atexit. The functions have no parameters and do not return values. Such functions can use only NLM (OS) level context.

See Using atexit() functions: Example (*NDK: Sample Code*).

See Also

AtUnload (page 45), exit (page 61), _exit (page 62), ExitThread (page 65)

AtUnload

(Designed for drivers) Registers a function that is called if the NLM is unloaded with the UNLOAD command

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int AtUnload  (
   void   (*func) (void));
```

Parameters

`func`

(IN) Points to the function to be registered.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Registration was successful.
–1		EFAILURE	Function was already registered.

Remarks

WARNING: AtUnload was designed for use with drivers when an NLM is unloaded with the UNLOAD command. Do *not* use it as a cleanup routine for NLM applications. CLIB context does not exist for a thread that is running an atexit or AtUnload routine, and a server abend may result.

The cleanup routine to be run at NLM unload time should be registered as a signal handler using signal with the condition SIGTERM. This signal handler can use the thread group ID from the main NLM to switch to a CLIB context, allowing it to call any CLIB function. Ensure that the signal handler routine always restores the original thread group ID before exiting. Also take care that the main NLM does not exit before the signal handler exits. You can do this with a global variable modified by the signal handler, which is monitored by the main NLM before exiting (see GetThreadGroupID (page 78) and SetThreadGroupID (page 108)).

The AtUnload function is passed the address of a function to be called when the NLM is unloaded. Such functions can use only NLM (OS) level context.

Only one function can be registered with AtUnload.

See Using AtUnload() functions: Example (*NDK: Sample Code*).

See Also

atexit (page 43), exit (page 61), _exit (page 62), ExitThread (page 65)

BeginThread

Initiates a new thread of execution within the current thread group

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int BeginThread  (
   void       (*func) (void *),
   void       *stack,
   unsigned    stackSize,
   void       *arg);
```

Parameters

`func`

(IN) Points to the function to execute as the new thread.

`stack`

(IN) Points to a block of memory to use for the new thread's stack.

`stackSize`

(IN) Specifies the size (in bytes) of the stack.

`arg`

(IN) Points to an argument to be passed to the new thread.

Return Values

This function returns the new thread's ID if successful. It returns EFAILURE if an error occurs.

If an error occurs, `errno` is set to:

Value	Name	Description
5	ENOMEM	Not enough memory.
9	EINVAL	Invalid argument was passed in.

Remarks

The new thread begins execution at the specified function (`func`). The function `func` receives `arg` as a parameter. The `stack` parameter is a pointer to a block of memory that the new thread uses as its stack.

- If `stack` is NULL, a block of memory (as specified by the `stackSize` parameter) is allocated.
- If `stack` is NULL and the specified stack size is too small, the size for the new thread stack is increased automatically.
- If `stack` is not NULL and the specified stack size is too small, the function fails and `errno` is set to EINVAL. The minimum stack size for the 3.x OS is 2,064 bytes and 16,384 bytes for the NetWare 4.x, 5.x, and 6.x OS.
- If `stackSize` is zero and stack is NULL, then the default stack size (8,192 bytes for 3.x or 16,384 bytes for 4.x, 5.x, and 6.x) is used.

The `arg` parameter is any 32-bit quantity, although typically some sort of pointer is passed, or NULL is passed if the specified function does not take any arguments.

If the newly created thread returns from the function `func`, it is be equivalent to its having executed the ExitThread function with an action code of EXIT_THREAD.

To begin a thread in a new thread group, call BeginThreadGroup.

See Also

BeginThreadGroup (page 49), ExitThread (page 65)

Example

```
#include <nwthread.h>

void newThreadFunc (char *funcArg);
int      completionCode;
.
.
.
completionCode = BeginThread (newThreadFunc, NULL, 8192, /A/Q
     "input.fil");
.
.
.
void newThreadFunc (char *arg)
{
   printf ("in new thread\n");
}
```

BeginThreadGroup

Establishes a new thread within a new thread group

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int BeginThreadGroup  (
   void       (*func) (void*),
   void       *stack,
   unsigned    stackSize,
   void       *arg);
```

Parameters

`func`

(IN) Points to the function to execute as the new thread.

`stack`

(IN) Points to a block of memory to use for the new thread's stack.

`stackSize`

(IN) Specifies the size (in bytes) of the stack.

`arg`

(IN) Points to an argument to be passed to the new thread.

Return Values

This function returns the new thread group's ID if successful. It returns EFAILURE if an error occurs.

If an error occurs, `errno` is set to:

Value	Name	Description
5	ENOMEM	Not enough memory.
9	EINVAL	Invalid argument was passed in.

Remarks

The BeginThreadGroup function creates a new thread group which contains one thread defined by `func`. Other than putting the new thread in its own thread group, this function is identical to BeginThread. A thread group can consist of one or more threads as defined by the programmer, and an NLM can have more than one thread group.

The new thread group is not the current thread group. To create more threads within the new thread group, you must make the new thread group current by calling SetThreadGroupID with the thread group ID returned by BeginThreadGroup. You can then create more threads within the new thread group by calling BeginThread for each additional thread for the new thread group.

The new thread begins execution at the specified function (`func`). The function `func` receives `arg` as a parameter. The `stack` parameter is a pointer to a block of memory that the new thread uses as its stack.

- If `stack` is NULL, a block of memory (as specified by the `stackSize` parameter) is allocated.
- If `stack` is NULL and the specified stack size is too small, the size for the new thread stack is increased automatically.
- If `stack` is not NULL and the specified stack size is too small, the function fails and `errno` is set to EINVAL. The minimum stack size for the 3.x OS is 2,064 bytes and 8,192 bytes for the 4.x, 5.x, and 6.x OS.
- If `stackSize` is zero and stack is NULL, then the default stack size (8,192 bytes for 3.x and for 4.x, 5.x, and 6.x) is used.

The `arg` parameter is any 32-bit quantity, although typically some sort of pointer is passed, or NULL is passed if the specified function does not take any arguments.

If the newly created thread returns from the function `func`, it is be equivalent to its having executed the ExitThread function with an action code of EXIT_THREAD.

See Also

BeginThread (page 47)

Example

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

void newThreadFunc (char *funcArg);
int   completionCode;
.
.
.
completionCode = BeginThreadGroup (newThreadFunc, NULL, 8192, "/A/Q
    input.fil");
.
.
.
void newThreadFunc(char *arg)
```

```
{
   printf ("in new thread group\n");
}
```

Breakpoint

Suspends the execution of an NLM and causes a break into the NetWare Internal Debugger

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void Breakpoint  (
   int   breakFlag);
```

Parameters

`breakFlag`

(IN) Specifies whether or not to take a breakpoint.

Return Values

None

Remarks

This function causes a breakpoint in the program if `breakFlag` is nonzero. The breakpoint occurs at the instruction following the call to Breakpoint. The `breakFlag` parameter can be any nonzero value to cause a breakpoint. The `breakFlag` value is loaded into the EDI register.

NOTE: If your application relies on the EDI register, you should not call Breakpoint with a nonzero value. Rather, call EnterDebugger, which merely enters the system debugger.

ClearNLMDontUnloadFlag

Sets a flag in the header of an NLM to allow it to be unloaded with the UNLOAD command at the system console

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ClearNLMDontUnloadFlag  (
   int   NLMID);
```

Parameters

NLMID

(IN) Specifies the ID of the NLM that is to be made so it can be unloaded from the system console. This ID can be obtained from the GetNLMID function.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
–1		EFAILURE	NLMID was an invalid NLM ID.
0	(0x00)	ESUCCESS	The don't unload flag has been set.

Remarks

This function reverses the effects of the SetNLMDontUnloadFlag function.

If SetNLMDontUnloadFlag is called, the NLM cannot be unloaded until ClearNLMDontUnloadFlag is called.

For more information unloading NLM applications, see CHECK Function (*NDK: NLM Development Concepts, Tools, and Functions*).

See Also

SetNLMDontUnloadFlag (page 102), GetNLMID (page 72)

Example

See example for SetNLMDontUnloadFlag (page 102).

CloseLocalSemaphore

Closes a local semaphore

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

int CloseLocalSemaphore  (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle

(IN) Specifies the semaphore handle of an open semaphore.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name
0	(0x00)	ESUCCESS

WARNING: A bad semaphore handle causes the server to abend.

Remarks

This function closes an open semaphore and makes any threads waiting on the semaphore runnable. After this function is called, the semaphore handle is no longer valid and should not be used again.

See Also

ExamineLocalSemaphore (page 60), OpenLocalSemaphore (page 89), SignalLocalSemaphore (page 114), TimedWaitOnLocalSemaphore (page 125), WaitOnLocalSemaphore (page 126)

delay

Suspends execution for an interval (milliseconds)

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void delay  (
   unsigned   milliseconds);
```

Parameters

`milliseconds`

(IN) Specifies the number of milliseconds the calling thread is to be delayed.

Return Values

This function returns no value. If an error occurs, `errno` is set to:

Value	Name	Description
–1	ENOMEM	Not enough memory.

Remarks

The delay function puts the calling thread to sleep for the number of milliseconds specified by the `milliseconds` parameter, rounded up to the next system clock tick.

NOTE: In practical application, the thread is delayed until it regains control of the processor, which might be considerably longer than the specified number of milliseconds.

A thread blocked on delay can be restarted (that is, delay can be cancelled) by calling ResumeThread.

See Also

EnterCritSec (page 58), SuspendThread (page 119), ThreadSwitch (page 122)

Example

```
#include <nwthread.h>
#include <stdio.h>
#include <nwconio.h>

main()
{
   printf("start");
   delay(10000);
   printf("end");
   getch();
}
```

EnterCritSec

Prevents all other threads in the NLM from being scheduled

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int EnterCritSec  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Threads were suspended in a critical section.

Remarks

The EnterCritSec function suspends all the other threads in the NLM.

After executing EnterCritSec, the current thread runs exclusively until it calls ExitCritSec. The ExitCritSec function makes the other threads in the NLM runnable again.

EnterCritSec and ExitCritSec can be nested. EnterCritSec maintains a count of the number of outstanding EnterCritSec requests. The count is increased with each call to EnterCritSec and decreased with each call to ExitCritSec. To restore normal thread dispatching ExitCritSec must be called once for each call to EnterCritSec.

The maximum number of concurrent critical sections is 4 billion.

Since NetWare 3.x, 4.x, 5.x, and 6.x are nonpreemptive operating systems, NLM applications very rarely need to use this function. The only time it is needed is when a critical section of code calls a function which might relinquish control.

Additionally, use of the EnterCritSec function should be avoided in favor of using locks or semaphores.

NOTE: If a new thread is started while the NLM is in a critical section, the new thread is also in the critical section. If the new thread is started after a call to EnterCritSec but before a corresponding call to ExitCritSec, the new thread is immdeately suspended.

See Also

ExitCritSec (page 64)

ExamineLocalSemaphore

Returns the current value of a local semaphore

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

int ExamineLocalSemaphore  (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle

(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns the current value of a semaphore.

WARNING: If a bad semaphore handle is specified, the server abends.

Remarks

This function returns the current value of a semaphore. The semaphore value is decremented for each WaitOnLocalSemaphore and incremented for each SignalLocalSemaphore. A positive semaphore value indicates that the thread can access the associated resource. If the semaphore value is zero or negative, the thread must either enter a waiting queue by calling the function WaitOnLocalSemaphore, or temporarily abandon its attempt to access the resource.

A semaphore handle is obtained by calling OpenLocalSemaphore.

See Also

CloseLocalSemaphore (page 55), OpenLocalSemaphore (page 89), SignalLocalSemaphore (page 114), TimedWaitOnLocalSemaphore (page 125), WaitOnLocalSemaphore (page 126)

exit

Causes the NLM to terminate normally

Local Servers: blocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <nwthread.h>
#include <stdlib.h>
#include <unistd.h>

void exit  (
   int   status);
```

Parameters

status

(IN) Specifies the NLM's return code. (Currently, the status is ignored.)

Return Values

None

Remarks

The exit function causes a normal termination consisting of the following sequence of events:

- All threads in the NLM are destroyed.
- This function calls the atexit functions, which are executed in "last-in, first-out" order.
- Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library Concepts (*NDK: Program Management*).)
- I/O buffers are flushed, and all second-level files (opened with fopen, fdopen, freopen) are closed. Any files created by tmpfile are deleted and purged.
- All screens are closed.
- All remote sessions are terminated.
- All other resources allocated by the NLM are freed.
- The NLM is unloaded.

See Also

abort (page 41), atexit (page 43), _exit (page 62), ExitThread (page 65), RegisterLibrary (*NDK: Program Management*)

_exit

Terminates the NLM without executing atexit functions or flushing buffers

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Thread

Syntax

```
#include <nwthread.h>

void _exit  (
   int   status);
```

Parameters

`status`

(IN) Specifies the NLM's return code (currently, the status is ignored.)

Return Values

None.

Remarks

The _exit function causes a normal termination consisting of the following sequence of events:

- All threads in the NLM are destroyed.
- Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library Concepts (*NDK: Program Management*).)
- All second-level files (opened with fopen, fdopen, freopen) are closed; however, the buffers of these are not flushed. Any files created by tmpfile are deleted and purged.
- All screens are closed.
- All remote sessions are terminated.
- All other resources allocated by the NLM are freed.
- The NLM is unloaded.

See Also

abort (page 41), exit (page 61), ExitThread (page 65), RegisterLibrary (*NDK: Program Management*)

Example

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

int main (int argc, char **argv)
{
   FILE *fp;
   atexit (myFunction);          /* myFunction declared elsewhere */
   fp = fopen (argv[1], "r");
   if (fp == NULL)
   {
      fprintf (stderr, Unable to open '%s'\n, argv[1]);
      _exit (1);
   }
   fclose (fp);
   exit (0);
}
```

ExitCritSec

Allows other threads in the NLM to run

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ExitCritSec  (void);
```

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Threads suspended by EnterCritSec are released to resume.
19		EWRNGKND	One or more threads in the NLM were not in a critical section.

Remarks

The ExitCritSec function reverses the effect of the EnterCritSec function.

NOTE: If a thread is created (with BeginThread or BeginThreadGroup) while the NLM is in a critical section, ExitCritSec returns EWRNGKND, but still releases threads suspended by EnterCritSec.

See Also

EnterCritSec (page 58), ResumeThread (page 92)

ExitThread

Terminates either the current thread or the NLM

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void ExitThread  (
   int   action_code,
   int   status);
```

Parameters

`action_code`

(IN) Specifies action code:

TSR_THREAD (-1)-Terminate only the current thread.

EXIT_THREAD (0)-Terminate the current thread; if the current thread is also the last thread, terminate the NLM.

EXIT_NLM (1)-Terminate the entire NLM.

`status`

(IN) Specifies the return code of the NLM (currently, the status is ignored).

Return Values

None

Remarks

The action code determines whether to destroy the current thread or the NLM:

- Action code TSR_THREAD terminates only the current thread and should only be used in NLM applications that are libraries (that is, they export symbols).
- Action code EXIT_THREAD is used to terminate the current thread. If the current thread is also the only thread of the NLM, the NLM itself is terminated.
- Action code EXIT_NLM is equivalent to the exit function.

The ExitThread function causes a normal NLM termination consisting of the following sequence of events:

- All threads in the NLM are destroyed.

- This function calls the atexit functions, which are executed in "last-in, first-out" order.
- Cleanup routines are called for any libraries that have registered cleanup routines and that the NLM has called. (For more information, see Library Concepts (*NDK: Program Management*).)
- I/O buffers are flushed, and all second-level files (opened with fopen, fdopen, freopen) are closed. Any files created by tmpfile are deleted and purged.
- All screens are closed.
- All remote sessions are terminated.
- All other resources allocated by the NLM are freed.
- The NLM is unloaded.

Executing the following statement

```
return (completionCode);
```

from the function (including main) where a thread began is equivalent to the following:

```
ExitThread (EXIT_THREAD, completionCode);
```

See Also

abort (page 41), exit (page 61), _exit (page 62), RegisterLibrary (*NDK: Program Management*)

FindNLMHandle

Returns the handle of a loaded NLM

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

unsigned int FindNLMHandle  (
   char   *moduleName);
```

Parameters

moduleName

(IN) Points to the full filename (including the extension) of the NLM whose handle is desired.

Return Values

If the NLM is loaded, its handle is returned. Otherwise, NULL is returned.

Remarks

This function searches the list of loaded NLM applications, comparing their names to the specified `moduleName`. If a match is found, the handle for that NLM is returned.

See Also

GetNLMHandle (page 71)

Example

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

unsigned int   moduleHandle;
moduleHandle = FindNLMHandle ("TEST.NLM");
if (moduleHandle == NULL)
   printf ("This NLM is not loaded!\n");
```

getcmd

Returns the command line parameters in their original format (unparsed), obtaining

Local Servers: nonblocking

Remote Servers: N/A

Classification: Other

Service: Thread

Syntax

```
#include <nwthread.h>

char *getcmd  (
   char   *originalCmdLine);
```

Parameters

originalCmdLine

(OUT) Points to a buffer into which to copy the command line parameters.

Return Values

The address of `originalCmdLine` is returned if `originalCmdLine` is not NULL. Otherwise, the address of the system's copy of the original command line is returned.

Remarks

The getcmd function copies the command line, with the "load" command and the program name removed, into a buffer specified by `originalCmdLine` (if `originalCmdLine` is not NULL).

If `originalCmdLine` is NULL, getcmd returns a pointer to the system's copy of the original command line (without the "load" command or the program name). The system's copy should not be written over.

The information is terminated with a \0 character. This provides a method of obtaining the original parameters to a program unchanged (with the white space intact).

This information can also be obtained by examining the `argv` vector of program parameters passed to the main function in the program.

See Also

spawnlp, spawnvp (page 115)

Example

```
#include <stdio.h>
#include <string.h>
#include <nwthread.h>

main()
{
   char   originalCmdLine[80];
   char   cmdPtr;
   getcmd(originalCmdLine)
   printf("%s\n",originalCmdLine);
   cmdPtr=getcmd(NULL);
   printf("%s\n", cmdPtr);

/*    If the load command is:
   "load test param1 param2 param3"
   The output is:
   param1 param2 param3
   param1 param2 param3
*/
}
```

getenv

Searches the environment area for the environment variable and returns the environment variable's value (presently, environment variables are not supported)

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <stdlib.h>

char *getenv  (
   const char   *varname);
```

Parameters

varname

(IN) Points to an environment variable.

Return Values

This function returns a pointer to the string assigned to the environment variable if found, and NULL if no match was found. Currently, NULL is always returned.

Remarks

The matching is case-insensitive; all lowercase letters are treated as if they were uppercase.

GetNLMHandle

Returns the handle of the current NLM

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

unsigned int GetNLMHandle  (void);
```

Return Values

Returns the handle of the current NLM.

Remarks

Ordinarily, the current NLM is the NLM that owns the currently running thread.

See impsymbl.c (../../../samplecode/clib_sample/nlm/impsymbl/impsymbl.c.html).

See Also

FindNLMHandle (page 67)

GetNLMID

Returns the ID of the current NLM

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMID  (void);
```

Return Values

This function returns the ID of the current NLM.

Remarks

Ordinarily, the current NLM is the NLM that owns the currently running thread. The current NLM identifies which NLM owns any subsequently allocated resources.

NOTE: The current NLM is usually the client even though the client might be executing a library's code.

See Also

SetNLMID (page 104)

GetNLMIDFromNLMHandle

Returns the ID of an NLM specified by its NLM handle

Local Servers: nonblocking

Remote Servers: N/A

Classification: 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMIDFromNLMHandle  (
   int   NLMHandle);
```

Parameters

NLMHandle

(IN) Specifies the NLM handle of the NLM whose ID is to be returned

Return Values

The ID of an NLM whose NLM handle is passed in

Remarks

GetNLMIDFromNLMHandle is used to get the ID of an NLM (other than the current NLM) if its handle is already known. To get the ID of the currently running NLM, call GetNLMID (page 72).

See Also

GetNLMHandle (page 71), GetNLMIDFromThreadID (page 74), SetNLMID (page 104)

GetNLMIDFromThreadID

Returns the ID of the NLM that the specified thread currently belongs to

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMIDFromThreadID  (
   int     threadID,
   char   *fileName );
```

Parameters

`threadID`

(IN) Specifies the ID of the thread.

`fileName`

(OUT) Points to the name of the file that the associated NLM was loaded from.

Return Values

If successful, GetNLMIDFromThreadID returns the ID of the NLM that the thread is associated with. On failure, it returns EFAILURE and `errno` is set to EBADHNDL.

Remarks

An NLM, such as a library NLM, can take over ownership of another NLM's threads by calling SetNLMID or SetThreadGroupID. However, an NLM must ensure that it owns all of its threads before unloading.

An NLM can keep track of the IDs of the threads it originates and can use GetNLMIDFromThreadID to determine if it still owns the threads. If at unload time, an NLM determines that it does not own its threads, the NLM must wait until ownership of the threads is returned to it. Then it can safely unload.

GetNLMIDFromThreadID returns the NLM ID only for threads that have CLIB context. This function returns EFAILURE if passed the ID of a thread that is running as an OS thread.

An example of an OS thread is a procedure scheduled with ScheduleSleepAESProcessEvent and with the registering thread's context specifier set to NO_CONTEXT. The registered thread does not have CLIB context when it runs.

NOTE: The interface to this function might change in the next release of the NetWare API to also return the name of the NLM. Currently, the name it returns is the name of the file that the NLM was loaded from. (An NLM can have a different name from its file name.)

See Also

GetThreadID (page 80)

GetNLMNameFromNLMID

Returns the name of a C Library NLM

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetNLMNameFromNLMID  (
   int     NLMID,
   char   *NLMFileName,
   char   *NLMName);
```

Parameters

NLMID

(IN) Specifies an NLM ID.

NLMFileName

(OUT) Points to the NLM filename used in the linker file.

NLMName

(OUT) Points to the descriptive name of the NLM.

Return Values

This function returns the NLM name. If an invalid NLM ID is passed, it returns a value of -1 and `errno` is set to EBADHNDL.

Remarks

This function returns the long name (as it appears on the module listing) and the short name (as specified in the NAME directive in the linker directive file) of the NLM. For example, if you specify the ID for CLIB.NLM for the `NLMID` parameter, on return `NLMFileName` points to CLIB.NLM and `NLMName` points to NLM.

See Also

MapNLMIDToHandle (page 85)

GetThreadContextSpecifier

Returns the CLIB context that is used by callback routines scheduled by the specified thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadContextSpecifier  (
   int   threadID);
```

Parameters

threadID

(IN) Specifies the ID of the thread whose context specifier you want to get.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name
-1		EFAILURE
0	(0x00)	NO_CONTEXT
1	(0x01)	USE_CURRENT_CONTEXT

Other values are valid thread group IDs.

Remarks

Many of the functions that are registered as callbacks run as OS threads. These threads need CLIB context to use the NetWare API functions. The function SetThreadContextSpecifier can be set to give these threads context when the callbacks are registered. This function lets you find out what those settings were.

If additional callbacks are registered, their context run as part of the thread group that corresponds to the thread group ID that is returned by this function.

See Also

SetThreadContextSpecifier (page 106)

GetThreadGroupID

Returns the ID of the current thread group

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadGroupID  (void);
```

Return Values

This function returns the ID of the current thread group.

Remarks

Ordinarily, the current thread group is the thread group that the currently running thread belongs to. The current thread group identifies which thread group's current connection, current task, current screen, and so on is active. (See Section 1.4, "Context and Thread Groups," on page 16.)

See Also

SetThreadGroupID (page 108)

GetThreadHandicap

Gets the number of context switches a thread is delayed before being rescheduled

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

LONG GetThreadHandicap  (
   int   threadID);
```

Parameters

threadID

(IN) Specifies a thread ID.

Return Values

This function returns the current handicap for the specified thread.

Remarks

A context switch is the task switching enacted by the OS when swapping the current thread.

See Also

ThreadSwitchWithDelay (page 124)

GetThreadID

Returns the thread ID of the currently executing thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadID  (void);
```

Return Values

The thread ID is returned.

Remarks

This function is used to get a thread ID for those functions which require a thread ID.

See Also

GetNLMID (page 72), GetThreadGroupID (page 78), SuspendThread (page 119)

GetThreadName

Returns the name of a C Library thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int GetThreadName  (
   int     threadID,
   char   *tName);
```

Parameters

`threadID`

(IN) Specifies a thread ID.

`tName`

(OUT) Points to the name of the thread.

Return Values

This function returns the name of a thread. If an invalid thread ID is passed, it returns an EBADHNDL error.

Remarks

This function returns the name of the specified C Library thread in `tName`. The `tName` parameter can hold up to 17+1 characters.

See Also

GetThreadID (page 80), RenameThread (page 91)

longjmp

Restores a saved environment

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <setjmp.h>

void longjmp  (
   jmp_buf   env,
   int       value);
```

Parameters

env

(IN) Specifies the environment to be restored.

value

(IN) Specifies the value to return.

Return Values

This function returns no value.

Remarks

The longjmp function restores the environment saved by the most recent call to setjmp with the corresponding `jmp_buf` argument. After longjmp restores the environment, program execution continues as if the corresponding call to setjmp has just returned the value specified by `value`.

The setjmp function must be called before longjmp. The routine that called setjmp and set up `env` must still be active and cannot have returned before longjmp is called. If this happens, the results are unpredictable. If `value` is 0, the value returned is 1.

See Also

setjmp (page 101)

main

A user-supplied function where NLM execution begins

Local Servers: blocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <nwthread.h>

int main  (
   int           argc,
   const char   *argv []);

int main  (void);

void main  (
   int           argc,
   const char   *argv []);

void main  (void);
```

Parameters

argc

(IN) Specifies the number of arguments on the command line.

argv

(IN) Points to the array of command line arguments pointers.

Syntax

The syntax for the main function can be any of the following:

Return Values

Currently, the return code from main is ignored.

Remarks

The main function is a user-written function that is executed as the initial thread of the NLM.

Prior to the main function receiving control, the _Prelude function (in PRELUDE.OBJ) does the following:

- The current connection is set to 0 and a unique task number is allocated.

- A new screen is created. The screen name is the name specified by the linker directive SCREENNAME. If the screen name is not specified, the description text specified in the FORMAT directive is used as the screen name. If the screen name is "none," "default," or "System Console," no new screen is created.
- A new thread is started with the specified stack size. If no stack size is specified, then the default stack size (8192 bytes) is used.
- The thread's name is the name specified by the linker directive THREADNAME. The thread name can be up to 16 characters long. The first thread name is generated by appending "0" to the specified thread name, the second by appending "1", and so on. If the thread name is not specified, the name specified with the linker directive NAME (with .NLM appended) is used as the pattern for generating thread names.
- If the main function returns with a return code of rc, it is equivalent to its executing ExitThread (EXIT_THREAD, rc). See the discussion of the ExitThread function.
- The command line to the program is assumed to be a sequence of tokens separated by blanks. The tokens are passed to the main function as an array of pointers to character strings in the `argv` parameter. The first element of `argv` is a pointer to a character string containing the program name, including the full path. The last element of the array pointed to by `argv` is a NULL pointer (`argv[argc]` is NULL). Arguments that contain blanks can be passed to the main function by enclosing them within double quote characters (which are removed from that element in the `argv` vector).

The command line can also be obtained in its original format by using the getcmd function.

See Also

abort (page 41), _exit (page 62), exit (page 61), ExitThread (page 65)

Example

```
#include <nwthread.h>

int main (int argc, char *argv[])
{
   /* Do the work */
   .
   .
   .
   /* Terminate thread and NLM */
   return 0;
}
```

MapNLMIDToHandle

Returns the handle associated with the NLM ID

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int MapNLMIDToHandle  (
   int   NLMID);
```

Parameters

NLMID

(IN) Specifies an NLM ID (C Library structure).

Return Values

This function returns the handle associated with the specified NLM ID. It returns a value of -1 if an invalid NLM ID was passed.

Remarks

This function can be used in registered LOAD and UNLOAD event handlers to compare known C Library NLM IDs with the NLM handle that the event handler function is passed with.

See Also

FindNLMHandle (page 67), GetNLMID (page 72), RegisterForEvent (*NDK: NLM Development Concepts, Tools, and Functions*)

NWSMPIsLoaded (obsolete 9/2001)

Returns whether the SMP kernel is loaded but is now obsolete.

Local Servers: N/A

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x and above SMP

Service: Thread

Syntax

```
#include <nwsmp.h>

int NWSMPIsLoaded  (
   void);
```

Return Values

The following table lists return values and descriptions.

TRUE	Returned if the SMP kernel is loaded
FALSE	Returned if the SMP kernel is not loaded

Remarks

The SMP kernel is loaded by loading smp.nlm in the startup.ncf file of a NetWare 4.x, 5.x, and 6.x server.

The value returned by NWSMPIsLoaded is not an indicator that more than one processor has been brought online. It is only an indicator the kernel is loaded.

MPDRIVER.NLM must be loaded in order to have multiple processors available to the SMP kernel.

NWThreadToMP (obsolete 9/2001)

Migrates a NetWare thread from the Netware kernel to the SMP kernel but is now obsolete

Local Servers: N/A

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x and above SMP

Service: Thread

Syntax

```
#include <nwsmp.h>

void NWThreadToMP  (
   void);
```

Return Values

None

Remarks

NWThreadToMP takes the currently running thread that was created by BeginThread, BeginThreadGroup, or ScheduleWorkToDo and migrates it to the SMP kernel. The SMP kernel then owns the thread scheduling and state context switches for that thread.

A thread migrated to the SMP kernel is still capable of having a CLib context. You can use the same CLib context functions to manipulate the CLib context.

Should an SMP thread call NWThreadToMP, the thread remains an SMP thread.

NWThreadToMP is exported by THREADS.NLM of the CLib suite.

See Also

NWThreadToNetWare (obsolete 9/2001) (page 88)

NWThreadToNetWare (obsolete 9/2001)

Migrates an SMP thread managed by the SMP kernel to the NetWare kernel as a NetWare thread but is now obsolete

Local Servers: N/A

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x and above SMP

Service: Thread

Syntax

```
#include <nwsmp.h>

void NWThreadToNetWare  (
   void);
```

Return Values

None

Remarks

NWThreadToNetWare takes a thread previously migrated to the SMP kernel, and puts it back on the NetWare kernel as a NetWare thread on the end of the run queue. A NetWare thread that calls NWThreadToNetWare will remain on the NetWare run queue.

NWThreadToNetWare is exported by THREADS.NLM of the CLib suite.

SMP threads are created by first creating a NetWare thread by calling BeginThread, BeginThreadGroup, or ScheduleWorkToDo and then migrating the thread to the SMP kernel by calling NWThreadToMP.

See Also

NWThreadToMP (obsolete 9/2001) (page 87)

OpenLocalSemaphore

Allocates a local semaphore and gives the NLM access to it

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

LONG OpenLocalSemaphore  (
   LONG   initialValue);
```

Parameters

`initialValue`

(IN) Specifies the value to assign the semaphore.

Return Values

If successful, returns a semaphore handle. On failure, returns zero.

Remarks

This function creates and initializes the semaphore to `initialValue`. The `initialValue` parameter indicates the number of threads that can access the resource at a time. A call to SignalLocalSemaphore increments this value. A call to WaitOnLocalSemaphore decrements this value.

WARNING: Developers must make sure to close local semaphores that are opened in an NLM because they are not automatically closed when an NLM unloads. If a local semaphore is opened but not closed before the NLM unloads, the server abends.

See Also

CloseLocalSemaphore (page 55), ExamineLocalSemaphore (page 60), SignalLocalSemaphore (page 114), TimedWaitOnLocalSemaphore (page 125), WaitOnLocalSemaphore (page 126)

raise

Sends a signal to the executing program

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <signal.h>

int raise  (
   int   condition);
```

Parameters

`condition`

(IN) Specifies the condition for which to raise a signal.

Return Values

This function returns a value of 0 when the condition is successfully raised and a nonzero value otherwise.

Remarks

The raise function signals the exception condition indicated by the `condition` parameter. The signal function can be used to specify the action to take place when a signal is raised. See the discussion of the signal function for a list of the possible conditions.

There can be no return of control following a call to the raise function if the action for that condition is to terminate the program or to transfer control using longjmp .

See Also

longjmp (page 82), signal (page 111)

RenameThread

Renames a C Library thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.11, 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int RenameThread  (
   int     threadID,
   char   *newName);
```

Parameters

threadID

(IN) Specifies a thread ID.

newName

(IN) Points to the new thread name.

Return Values

This function returns ESUCCESS if it completes successfully. It returns EBADHNDL if an invalid thread ID is passed.

Remarks

This function renames a C Library thread. The new name can be up to 17 characters long.

ResumeThread

Allows a previously suspended thread to run

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ResumeThread  (
   int   threadID);
```

Parameters

threadID

(IN) Specifies the ID of the thread to be resumed.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Thread was resumed.
9		EINVAL	Thread tried to resume itself.
19		EWRNGKND	Thread was not suspended.
22	(0x16)	EBADHNDL	Bad thread ID was passed in.

Remarks

The ResumeThread function reverses the effect of SuspendThread.

IMPORTANT: Your application must be aware of how its threads are suspended and must call ResumeThread only when it is appropriate. For example, if your application suspends a thread, it is appropriate for your application to resume the thread. However, it is not appropriate for your application to call ResumeThread for one of its threads that has been suspended by the OS, while the thread is running in OS code. Calling ResumeThread at an inappropriate time will cause unpredictable behavior.

See Also

ExitCritSec (page 64), SuspendThread (page 119)

ReturnNLMVersionInfoFromFile

Returns version information for a loaded NLM that corresponds to a specified file

Local Servers: blocking

Remote Servers: blocking

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ReturnNLMVersionInfoFromFile  (
   BYTE   *__pathName
   LONG   *majorVersion,
   LONG   *minorVersion,
   LONG   *revision,
   LONG   *year,
   LONG   *month,
   LONG   *day,
   BYTE   *copyrightString
   BYTE   *description);
```

Parameters

__pathName

(IN) Points to the path to the NLM file whose version information is to be returned.

majorVersion

(OUT) Points to the major version number of the NLM.

minorVersion

(OUT) Points to the minor version number of the NLM.

revision

(OUT) Points to the revision number of the NLM.

year

(OUT) Points to the number of the year that the NLM was created.

month

(OUT) Points to the number of the month that the NLM was created.

day

(OUT) Points to the number of the day that the NLM was created.

copyrightString

(OUT) Points to a buffer that receives an ASCIIZ string containing the copyright string of the NLM. Buffer size should be 256 bytes.

description

(OUT) Points to a buffer that receives an ASCIIZ string containing the name that is displayed when the NLM is loaded. Buffer size should be 128 bytes.

Return Values

The following table lists return values and descriptions.

Value	Name	Description
–1	EFAILURE	An invalid NLM handle was specified.
0	ESUCCES	

Remarks

While `__pathName` must be supplied, the other parameters can be set to NULL if you do not want the information they return.

The NLM specified by `__pathName` does not need to be running for this function to retrieve its information.

The information for `majorVersion`, `minorVersion`, `revision`, `copyrightString`, and `description` are set with linker options when the NLM applications are linked. For more information about the linker options, see Using a Linker.

The buffer `description` points to should be at least 128 bytes.

For the NetWare 3.11 OS, this function was made available in CLIB.NLM, version 3.11b.

See Also

MapNLMIDToHandle (page 85), FindNLMHandle (page 67), ReturnNLMVersionInformation (page 96)

ReturnNLMVersionInformation

Returns version information for a loaded NLM that corresponds a specified NLM handle

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ReturnNLMVersionInformation  (
   int     NLMHandle,
   LONG   *majorVersion,
   LONG   *minorVersion,
   LONG   *revision,
   LONG   *year,
   LONG   *month,
   LONG   *day,
   BYTE   *copyrightString,
   BYTE   *description);
```

Parameters

NLMHandle

(IN) Specifies the handle of the NLM for which to return version information. This handle can be obtained by calling FindNLMHandle or MapNLMIDToHandle.

majorVersion

(OUT) Points to the major version number of the NLM.

minorVersion

(OUT) Points to the minor version number of the NLM.

revision

(OUT) Points to the revision number of the NLM.

year

(OUT) Points to the number of the year that the NLM was created.

month

(OUT) Points to the number of the month that the NLM was created.

day

(OUT) Points to the number of the day that the NLM was created.

copyrightString

(OUT) Points to a buffer that receives an ASCIIZ string containing the copyright string of the NLM. Buffer size should be 256 bytes.

`description`

(OUT) Points to a buffer that receives an ASCIIZ string containing the name that is displayed when the NLM is loaded. Buffer size should be 128 bytes.

Return Values

The following table lists return values and descriptions.

Value	Name	Description
–1	EFAILURE	An invalid NLM handle was specified.
0	ESUCCESS	

Remarks

While `NLMHandle` must be supplied, the other parameters can be set to NULL if you do not want the information they return.

The information for `majorVersion`, `minorVersion`, `revision`, `copyrightString`, and `description` are set with linker options when the NLM applications are linked. For more information about the linker options, see Using a Linker (*NDK: NLM Development Concepts, Tools, and Functions*).

For the NetWare 3.11 OS, this function was made available in CLIB.NLM, version 3.11b.

See Also

MapNLMIDToHandle (page 85), FindNLMHandle (page 67), ReturnNLMVersionInfoFromFile (page 94)

ScheduleWorkToDo

Schedules a routine as work, which puts it on the highest priority queue, the WorkToDoList

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int ScheduleWorkToDo  (
   void       (*ProcedureToCall) ( ) ,
   void       *workData,
   WorkToDo   *workToDo);
```

Parameters

`ProcedureToCall`

(IN) Points to the routine being scheduled as work.

`workData`

(IN) Points to the data to be passed to the worker thread.

`workToDo`

(IN) Points to a WorkToDo structure.

Return Values

The following table lists return values and descriptions.

0	Success
5	ENOMEM

Remarks

This function schedules work to be executed by an OS worker thread.

The `ProcedureToCall` parameter points to the procedure to be scheduled as work. Work is a high-priority, low overhead procedure. See "When to Schedule a Routine as Work" on page 15.

The `workData` parameter contains the data to be passed to `ProcedureToCall`.

The `workToDo` parameter is a structure used by CLIB.NLM to set up WorkToDo process scheduling. The structure must be allocated before calling this function and released afterward (for example, by calling malloc and free). Other than allocating `workToDo`, the developer does not need

to be concerned with the details of this stucture since the only user-defined fields in the structure are set by the `ProcedureToCall` and `workData` parameters. WorkToDo is defined in nwthread.h.

Since the work that is scheduled is done by an OS worker thread, it is not be able to use the NetWare API functions that use context, unless context is given to the OS worker thread.

The context that is given to the OS worker thread is determined by the value in the registering thread's context specifier. You can set the context specifier to one of the following options:

- NO_CONTEXT-Callbacks registered with this option are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

  Once inside of your callback, you can manually give your callback thread CLIB context by calling SetThreadGroupID and passing in a valid thread group ID. If you manually set up your context, you need to reset its context to its original context, by setting the thread group ID back to its original value.
- USE_CURRENT_CONTEXT-Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread.
- A valid thread group ID-This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started with BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. You use SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

For more information on using CLIB context, see "Context Problems with OS Threads" on page 35.

See Also

BeginThread (page 47), BeginThreadGroup (page 49)

Example

```
#include <stdio.h>
#include <nwthread.h>
#include <nwconio.h>

int   count = 0;

/*.......................*/
void ScreenUpdater(void *data)
{
   data = data;
   count++;
   clrscr();
   printf("You could use a work to do thread\n\n");
   printf("to do screen updates.  %i.\n\n\n", count);
   printf("Work to do threads get into the\n\n");
```

```
   printf("system fast and are useful to\n\n");
   printf("accomplish finite definable tasks.");
}

/*.......................*/
main()
{
   WorkToDo   screenWork;
   char       ch = 0;
   SetAutoScreenDestructionMode(TRUE);
   while (ch != 'q')
   {
      ScheduleWorkToDo(ScreenUpdater, NULL, &screenWork);

      /* ThreadSwitch makes sure work to do gets a chance to run. */
      ThreadSwitch();
      if (!kbhit())
      ThreadSwitchWithDelay();
      else
      ch = getch();
   }
}
/*.......................*/
```

setjmp

Saves its calling environment in its `env` parameter for subsequent use by the longjmp function

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <setjmp.h>

int setjmp  (
   jmp_buf   env);
```

Parameters

env

(OUT) Specifies the buffer in which to save environment.

Return Values

This function returns a value of 0 when it is initially called. The return value is nonzero if the return is the result of a call to the longjmp function. An if statement is often used to handle these two returns. When the return value is 0, the initial call to setjmp has been made; when the return value is nonzero, a return from a longjmp has just occurred.

Remarks

In some cases, error handling can be implemented by using setjmp to record the point to which a return occurs following an error. When an error is detected in a called function, that function uses longjmp to jump back to the recorded position. The original function which called setjmp must still be active (it cannot have returned to the function which called it).

Special care must be exercised to ensure that any side effects that have occurred (such as allocated memory and opened files) are satisfactorily handled.

See Also

longjmp (page 82)

SetNLMDontUnloadFlag

Sets a flag in the header of an NLM to prevent the NLM from being unloaded with the UNLOAD command at the system console

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int SetNLMDontUnloadFlag  (
   int   NLMID);
```

Parameters

NLMID

(IN) Specifies the ID of the NLM that is to be made so it cannot be unload from the command line.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
–1		EFAILURE	NLMID was an invalid NLM ID.
0	(0x00)	ESUCCESS	The don't unload flag has been set.

Remarks

A console operator can unload an NLM from the system console command line by issuing the following command:

```
UNLOAD NLM_NAME
```

where NLM_NAME is the name of the NLM being unloaded.

If there is a check function for the NLM (declared with the OPTION CHECK directive), it is called when the "UNLOAD NLM_NAME" command is entered. This function then must decide if it is safe to unload the NLM. If it is safe to unload the NLM, the function returns 0 and the NLM is unloaded. If the function determines that the NLM should not be unloaded, it returns a nonzero value and the following prompt is displayed on the system console:

```
UNLOAD module anyway? n
```

In this case, the console operation can choose to unload the NLM anyway, by pressing the "y" key, instead of the "n" key.

If SetNLMDontUnloadFlag is called, the NLM can only be unloaded after ClearNLMDontUnloadFlag is called.

For more information about unloading NLM applications, see CHECK Function (*NDK: NLM Development Concepts, Tools, and Functions*).

See Also

ClearNLMDontUnloadFlag (page 53), GetNLMID (page 72)

Example

```
#include <nwconio.h>
#include <errno.h>
#include <stdio.h>
#include <nwthread.h>

main()
{
   int NLMID, result;
   NLMID=GetNLMID();
   result=SetNLMDontUnloadFlag(NLMID);
   if(result==ESUCCESS)
   {
      printf("DONTUNLD.NLM cannot be unloaded now.\n");
      printf("Press any key to be able to unload this NLM\n");
      getch();
      ClearNLMDontUnloadFlag(NLMID);
      printf("\nYou can unload DONTUNLD.NLM now.\n");
      getch();
   }
   else
      printf("Could not set the don't unload flag.\n");
}
```

SetNLMID

Changes the current NLM ID

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int SetNLMID  (
   int   newNLMID);
```

Parameters

`newNLMID`

(IN) Specifies the ID of NLM to make current (returned by a previous call to SetNLMID or GetNLMID.

Return Values

This function returns the ID of the previously current NLM if successful. Otherwise, it returns EFAILURE and sets `errno` to:

Value	Hex	Name	Description
22	(0x16)	EBADHNDL	Invalid NLM ID was passed in.

Remarks

SetNLMID changes the NLM context for the calling thread and its entire thread group. If a library NLM calls SetNLMID from a client thread, the NLM level context of the thread's entire thread group in the client NLM is changed. For this reason, library NLMs are discouraged from calling SetNLMID.

The current NLM determines which NLM "owns" resources that are subsequently allocated. (See Connection Number and Task Management Concepts (*NDK: Connection, Message, and NCP Extensions*) for a discussion of resources.)

The main implication of "ownership" of resources is the automatic cleanup performed by the NetWare API when an NLM terminates. In the case of a library/client relationship:

- If a library allocates resources while being called by a client, by default (without calling SetNLMID), the resources are owned by the client.

- If the library calls SetNLMID to make itself the current NLM and then allocates resources, the library owns the resources.

NOTE: A library should save and restore the client's NLM ID when it changes the current NLM ID.

See Also

GetNLMID (page 72)

SetThreadContextSpecifier

Determines the CLIB context that is to be used by all callback routines scheduled by the specified thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int SetThreadContextSpecifier (
   int   threadID,
   int   contextSpecifier);
```

Parameters

threadID

(IN) Specifies the ID of the thread whose context specifier you want to get.

contextSpecifier

(IN) Specifies the context to give callback threads.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name
–1		EFAILURE
0	(0x00)	ESUCCESS

Remarks

Many of the functions that are registered as callbacks run as OS threads. These threads need CLIB context to use the NetWare API functions, such as printf. This function is used to determine what context is given to callbacks when they are registered by the calling thread.

The default setting is for callbacks to have the context of the thread that calls them.

The thread context specifier is set on a per-thread basis. Changing the context specifier for one thread does not change it for any of the other threads. The `threadID` parameter specifies which thread should have its context specifier set.

The `contextSpecifier` parameter tells what the context should be. It can be one of the following:

- NO_CONTEXT-Do not give CLIB context to callback functions when they are registered. You would use this option when your callback is not going to use any NetWare API functions other than local semaphore calls and SetThreadGroupID, which then creates context. You could also use this if you want to manually set the callbacks function with a call to SetThreadGroupID.
- USE_CURRENT_CONTEXT-Set the context of the callback being scheduled to be the same as the thread that is scheduling the callback. This is the default setting that exists when a new thread is started.
- A valid thread group ID-Set the context of the callback to this thread group ID. The ID of the current thread group can be returned with a call to GetThreadGroupID.

See Also

GetThreadContextSpecifier (page 77), SetThreadGroupID (page 108), GetThreadGroupID (page 78)

SetThreadGroupID

Changes the thread group ID of the running thread

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int SetThreadGroupID  (
   int   newThreadGroupID);
```

Parameters

`newThreadGroupID`

(IN) Specifies the ID of the thread group to make current (returned by a previous call to SetThreadGroupID or GetThreadGroupID.

Return Values

This function returns the ID of the previously current thread group if successful. Otherwise, it returns EFAILURE and sets `errno` to:

Value	Hex	Name	Description
22	(0x16)	EBADHNDL	Invalid thread group ID was passed in.

Remarks

The SetThreadGroupID function determines which instance of the current connection, current task, current screen, and so on, is used. (See Section 1.4, “Context and Thread Groups,” on page 16.) Since a thread group is owned by a particular NLM, this function also sets the current NLM ID to the NLM that owns the thread group that is being made current.

NOTE: A library should save and restore the thread group ID whenever it changes the current thread group.

See Also

GetThreadGroupID (page 78), SetNLMID (page 104)

Example

```
#include <nwthread.h>

int   currentThreadGroupID;
int   newThreadGroupID;
currentThreadGroupID = SetThreadGroupID (newThreadGroupID);
```

SetThreadHandicap

Sets the number of context switches a thread is permanently handicapped (delayed) before being rescheduled

Local Servers: nonblocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void SetThreadHandicap  (
   int   threadID,
   int   handicap);
```

Parameters

threadID

(IN) Specifies a thread ID.

handicap

(IN) Specifies the number of context switches the thread waits before being put on the Run Queue.

Return Values

None

Remarks

This function sets the value used to determine the number of context switches a thread waits before being put on the Run Queue. This sets the permanent handicap.

See Also

GetThreadHandicap (page 79), ThreadSwitchWithDelay (page 124)

signal

Specifies an action to take place when certain conditions are detected (signalled) while a program executes

Local Servers: nonblocking

Remote Servers: N/A

Classification: ANSI

Service: Thread

Syntax

```
#include <signal.h>

void (*signal  (
   int    sig,
   void   (*func) (
      int)))
   (int);
```

Parameters

sig

(IN) Specifies the condition being signalled.

func

(IN) Points to the function to be called when the signalled condition occurs.

Return Values

Returns the previous setting if successful or SIG_ERR if a failure occurred.

Remarks

signal is used to specify an action to take place when certain conditions are detected while a program executes. These conditions are defined to be:

Signal	Description
SIGABRT	Abnormal termination, caused by the abort function.
SIGFPE	An erroneous floating-point operation occurs, such as division by zero, overflow and underflow (supported only for compiler option / fpc; not supported for options /fpi, /7, /fpi87).
SIGILL	An illegal instruction is encountered. (Currently not supported.)
SIGINT	Raised if the Ctrl+C keys are pressed during screen output (other than to the System Console Screen).

Signal	Description
SIGSEGV	An illegal memory reference is detected. (Currently not supported.)
SIGTERM	An UNLOAD command has been entered for the NL.

The `func` parameter is used to specify an action to take for the specified condition:

When the `func` parameter is a function name, that function is called equivalently to the following code sequence.

```
   /*"sig" is the condition being signalled*/
signal (sig, SIG_DFL);
(*func) (sig);
```

The function specified by the `func` parameter can terminate the program by calling the exit, _exit, ExitThread, or abort functions. It can also call the longjmp function or it can return. Because the next signal is handled with default handling, the program must again call signal if it is desired to handle the next condition of the type that has been signalled.

NOTE: The exit, _exit, ExitThread, and abort functions cannot be called from the context of a SIGTERM handler or the server console will be inoperational.

A registered SIGTERM signal handler in NetWare 3.11, 4.x, 5.x, and 6.x is on a per-NLM basis.You only have to register your SIGTERM handler once for the NLM. The other signals are on a per-threadgroup basis. For these signals, you have to register your signal handler every time you start a new thread group (by calling the BeginThreadGroup function). If not, your signal handler is not called.

Setting	Description
SIG_IGN	This value causes the indicated condition to be ignored.
SIG_DFL	This value causes the default action for the condition to occur.

The initial settings for the NetWare API are as follows:

Signal	Default Setting
SIGABRT	SIG_DFL
SIGFPE	SIG_IGN
SIGILL	SIG_IGN
SIGINT	SIG_DFL
SIGSEGV	SIG_IGN
SIGTERM	SIG_DFL

A condition can be generated by a program by calling the raise function.

The default action for the SIGABRT action is to call _exit (3). The default action for SIGINT is to call abort (). The default action for the other conditions is to ignore the condition.

The functions registered with signal run as callbacks, so CLIB context is an issue. If a callback does not have CLIB context, it cannot make calls to the NetWare API functions that require context.

The functions registered for SIGFPE, SIGILL, SIGINT, SIGSEGV, and SIGTERM have the thread group context of the thread that was running when the signal condition was detected. They can use the NetWare API functions without additional setup.

However, you do need to set up context for the functions registered for SIGABRT.

For 3.11 NLM applications, you must manually create the thread group context in your callback functions, by calling SetThreadGroupID and passing a valid thread group ID. Before this thread returns, it should reset its context to its original context, by setting the thread group ID back to its original value.

For 4.x, 5.x, and 6.x NLM applications, the context that is given to the callbacks when they are registered is determined by the value in the registering thread's context specifier. You can set the context specifier to one of the following options:

- NO_CONTEXT-Callbacks registered with this option are not given CLIB context. The advantage here is that you avoid the overhead needed for setting up CLIB context. The disadvantage is that without the context the callback is only able to call NetWare API functions that manipulate data or manage local semaphores.

  Once inside of your callback, you can manually give your callback thread CLIB context by calling SetThreadGroupID and passing in a valid thread group ID. If you manually set up your context, you need to reset its context to its original context, by setting the thread group ID back to its original value.
- USE_CURRENT_CONTEXT-Callbacks registered with a thread that has its context specifier set to USE_CURRENT_CONTEXT have the thread group context of the registering thread.
- A valid thread group ID-This is to be used when you want the callbacks to have a different thread group context than the thread that schedules them.

When a new thread is started with BeginThread, BeginThreadGroup or ScheduleWorkToDo, its context specifier is set to USE_CURRENT_CONTEXT by default.

You can determine the current setting of the registering thread's context specifier by calling GetThreadContextSpecifier. You use SetThreadContextSpecifier to set the registering thread's context specifier to one of the above options.

For more information on using CLIB context, see "Context Problems with OS Threads" on page 35.

See Also

abort (page 41), _exit (page 62), longjmp (page 82), raise (page 90), setjmp (page 101)

SignalLocalSemaphore

Increments the semaphore value of the specified semaphore

Local Servers: nonblocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

int SignalLocalSemaphore  (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle

(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns zero.

WARNING: A bad semaphore handle causes the server to abend.

Remarks

A thread normally call this function when finished accessing the resource associated with the semaphore.

A thread can also use this function to restart another thread waiting on the semaphore, as a means of interprocess synchronization.

If there are threads waiting on the semaphore (the semaphore value is negative), the first thread in the queue is released (made runnable).

A semaphore handle can be obtained by calling OpenLocalSemaphore.

See Also

CloseLocalSemaphore (page 55), ExamineLocalSemaphore (page 60), OpenLocalSemaphore (page 89), TimedWaitOnLocalSemaphore (page 125), WaitOnLocalSemaphore (page 126)

spawnlp, spawnvp

Executes a new NLM

Local Servers: blocking

Remote Servers: N/A

Classification: Other

Service: Thread

Syntax

```
#include <nwthread.h>

int spawnlp  (
   int           mode,
   const char   *path,
   char         *arg0,
   ...);

int spawnvp  (
   int           mode,
   const char   *path,
   char        **argv);
```

Parameters

mode

(IN) Specifies how the invoking program behaves after it is initiated.

path

(IN) Points to the name of the compiled program to be started.

arg0

(IN) Points to the first of a list of arguments to be passed to the invoked program.

argv

(IN) Points to an array of pointers to arguments to be passed to the invoked program.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Program was successfully loaded.
–1	(0xFF)	EFAILURE	The function failed.

On error, `errno` is set to one of the following values:

Value	Hex	Name
1	(0x01)	ENOENT
3	(0x03)	ENOEXEC
5	(0x05)	ENOMEM
6	(0x06)	EACCES
9	(0X09)	EINVAL (See note below)
16	(0x10)	EINUSE
19	(0x13)	EWRNGKND
21	(0x15)	ERESOURCE
28	(0x1C)	EIO
37	(0x25)	EALREADY

Through a Novell internal conversion process, `NwErrno` may also be set to one of the following values:

Value	Hex	Name
1	(0x01)	LOAD_COULD_NOT_FIND_FILE
2	(0x02)	LOAD_ERROR_READING_FILE
3	(0x03)	LOAD_NOT_NLM_FILE_FORMAT
4	(0x04)	LOAD_WRONG_NLM_FILE_VERSION
5	(0x05)	LOAD_REENTRANT_INITIALIZE_FAILURE
6	(x006)	LOAD_CAN_NOT_LOAD_MULTIPLE_COPIES
7	(0x07)	LOAD_ALREADY_IN_PROGRESS
8	(0x08)	LOAD_NOT_ENOUGH_MEMORY
9	(0x09)	LOAD_INITIALIZE_FAILURE
10	(0x0A)	LOAD_INCONSISTENT_FILE_FORMAT
11	(0x0B)	LOAD_CAN_NOT_LOAD_AT_STARTUP
12	(0x0C)	LOAD_AUTO_LOAD_MODULES_NOT_LOADED
13	(0x0D)	LOAD_UNRESOLVED_EXTERNAL
14	(0x0E)	LOAD_PUBLIC_ALREADY_DEFINED

NOTE: The `errno` code EINVAL indicates only that `code` was set to other than P_NOWAIT or P_WAIT.

IMPORTANT: P_WAIT is operative only in NLMs made with CLIB. Other NLMs indicate load completion only.

Remarks

The value of `mode` determines how the program is loaded and how the invoking program behaves after the it is initiated:

Mode	Description
P_NOWAIT	The invoked program is loaded into available memory and is executed. The original program executes simultaneously with the invoked program.
P_WAIT	The invoked program is loaded into available memory and is executed. The calling thread is suspended until the invoked program finishes (the NLM loads). Under NetWare 5.x and 6.x, the exit status of the invoked program is returned.

IMPORTANT: P_WAIT functionality is available only on NetWare 5.x, 6.x, and NetWare 4 systems updated to use CLib v. 4.11 libraries, the official update for NetWare v. 4.10 systems.

Arguments are passed to the child process by supplying one or more pointers to character strings as arguments in the spawn call. These character strings are concatenated with spaces inserted to separate the arguments to form one argument string for the child process. The length of this concatenated string must not exceed 128 bytes.

The arguments can be passed as a list of arguments (spawnlp) or as a vector of pointers (spawnvp). At least one argument, `arg0` or `argv` [0], must be passed to the child process. By convention, this first argument is the name of the program.

If the arguments are passed as a list, there must be a NULL pointer to mark the end of the argument list.

See Also

abort (page 41), atexit (page 43), exit (page 61), _exit (page 62), getcmd (page 68), getenv (page 70), main (page 83), system (page 120)

Example

spawnlp

```
#include <nwthread.h>

int   completionCode:
completionCode = spawnlp (P_NOWAIT, "helper.NLM", NULL);
```

spawnvp

```
#include <nwthread.h>

int    completionCode;
```

```
char   *argv[5];
completionCode = spawnvp (P_NOWAIT, "helper.nlm", argv);
```

SuspendThread

Prevents a specified thread in the NLM from being scheduled

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

int SuspendThread  (
   int   threadID);
```

Parameters

threadID

(IN) Specifies the ID of the thread to be suspended.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name	Description
0	(0x00)	ESUCCESS	Thread was suspended.
22	(0x16)	EBADHNDL	A bad thread ID was passed in.

Remarks

This function causes a specified thread to be suspended. ResumeThread makes the thread runnable once again.

SuspendThread maintains a count of the number of times a thread is suspended. An equal number of calls to ResumeThread must be performed for the thread to run again. This allows calls to SuspendThread and ResumeThread to be nested.

Blocking Information SuspendThread does not block when suspending other threads, but blocks when suspending itself.

See Also

EnterCritSec (page 58), ResumeThread (page 92)

system

Executes operating system commands

Local Servers: blocking

Remote Servers: N/A

Classification: ANSI

NetWare Server: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <stdlib.h>
#include <nwthread.h>

int  system  (
   const char   *command);
```

Parameters

command

(IN) Points to a command to execute.

Return Values

If successful in passing the command to the operating system, returns 0. Any errors in executing the operating system commands are shown on the system console screen.

If the command string is longer than 512 characters, returns -1 and sets errno to the following:

Decimal	Constant	Description
65	ENAMETOOLONG	The command parameter exceeds the maximum length of 512 characters.

Remarks

This function always echoes input directly to the system console screen. Errors in executing the operating system command are shown on the system console screen.

NOTE: If the console operator is typing, your string will be intermixed with the input from the console operator.

See Also

abort (page 41), atexit (page 43), exit (page 61), _exit (page 62), spawnlp, spawnvp (page 115)

Example

```
#include <stdio.h>
#include <stdlib.h>
#include <nwthread.h>

/*_____*/
main ()
{
    .
    .
    .
    system ("LOAD MONITOR");
    .
    .
    .
}
/*_____*/
```

ThreadSwitch

Allows other runnable threads a chance to get some work done, where no natural break in the currently running thread would normally occur

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitch  (void);
```

Return Values

None

Remarks

The NetWare 3.x, 4.x, 5.x, and 6.x environment is a nonpreemptive environment in which threads can only relinquish control via system calls. Unless an executing thread relinquishes control, other threads do not have the opportunity to work.

If no natural break occurs via a system call in a particular thread, ThreadSwitch can be used to cause that thread to relinquish control and allow other runnable threads to execute.

NOTE: If you are using "busy waiting" or "spin locks" you should use ThreadSwitchWithDelay instead of ThreadSwitch because threads preempted with ThreadSwitch still have higher priority than threads on the low priority queue. These low priority threads (such as those doing file compression in the OS) still need an opportunity to run in the nonpreemptive 4.x, 5.x, and 6.x environment.

ThreadSwitchLowPriority

Reschedules a thread onto the low-priority queue

Local Servers: blocking

Remote Servers: N/A

Classification: 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitchLowPriority  (void);
```

Return Values

None

Remarks

The ThreadSwitchLowPriority function can be used to schedule a thread to run only when there is nothing but hardware polling routines and temporarily handicapped threads to run. Routines suitable for this priority level would be once-a-week backup, file compression utilities, low-priority clean-up utilities, and so forth.

ThreadSwitchWithDelay

Reschedules the thread to be placed on the RunList after `n` number of context switches have taken place

Local Servers: blocking

Remote Servers: N/A

Classification: 3.12, 3.2, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwthread.h>

void ThreadSwitchWithDelay  (void);
```

Return Values

None

Remarks

If a thread needs a resource that will not be ready for a moment but does not want the overhead of sleeping on a semaphore, rather than rescheduling itself repetitively the thread can reschedule itself with a temporary handicap.

Temporarily handicapped threads are scheduled on a waiting queue, the DelayedList, until their handicap has expired. Upon expiration, they are rescheduled at the end of the RunList. Letting threads temporarily handicap themselves prevents needless rescheduling caused by a spin-waiting condition.

The number of switches in each temporary handicap is a tunable parameter inside the NetWare OS.

TimedWaitOnLocalSemaphore

Waits on a local semaphore until it is signalled or the specified timeout elapses

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

int TimedWaitOnLocalSemaphore  (
   LONG   semaphoreHandle,
   LONG   timeout);
```

Parameters

`semaphoreHandle`

(IN) Specifies the handle of the semaphore to wait on.

`timeout`

(IN) Specifies the maximum time, in milliseconds, to wait on the semaphore.

Return Values

The following table lists return values and descriptions.

Value	Hex	Name
0	(0x00)	ESUCCESS
254	(0xFE)	ERR_TIMEOUT_FAILURE

WARNING: A bad semaphore handle causes the server to abend.

Remarks

TimedWaitOnLocalSemaphore is similar to WaitOnLocalSemaphore except that a waiting time is specified. If the semaphore is not signalled prior to the expiration of the `timeout` parameter period, TimedWaitOnLocalSemaphore returns an error.

See Also

CloseLocalSemaphore (page 55), ExamineLocalSemaphore (page 60), OpenLocalSemaphore (page 89), SignalLocalSemaphore (page 114), WaitOnLocalSemaphore (page 126)

WaitOnLocalSemaphore

Decrements the semaphore value of the specified semaphore

Local Servers: blocking

Remote Servers: N/A

Classification: 3.x, 4.x, 5.x, 6.x

Service: Thread

Syntax

```
#include <nwsemaph.h>

int WaitOnLocalSemaphore  (
   LONG   semaphoreHandle);
```

Parameters

semaphoreHandle

(IN) Specifies the semaphore handle of an open semaphore.

Return Values

If successful, this function returns zero.

WARNING: A bad semaphore handle causes the server to abend.

Remarks

A thread would typically call this function before accessing the resource associated with the semaphore. An NLM can also use this function to cause a thread to wait until another thread signals it to resume.

If the semaphore value is still greater than or equal to zero after the function decrements it, the thread is not suspended. If the semaphore value is negative, the thread is suspended until the semaphore is signalled one more time than there are threads ahead of the current thread on the specified semaphore's queue.

A semaphore handle can be obtained by calling OpenLocalSemaphore.

See Also

CloseLocalSemaphore (page 55), ExamineLocalSemaphore (page 60), OpenLocalSemaphore (page 89), SignalLocalSemaphore (page 114), TimedWaitOnLocalSemaphore (page 125)

Revision History

A

The following table outlines all the changes that have been made to the NLM Threads Management documentation (in reverse chronological order):

October 11, 2006	Updated AdvertiseService to show it as an unsupported function.
March 1, 2006	Updated format.
October 5, 2005	Transitioned to revised Novell documentation standards.
March 2, 2005	Fixed legal information.
Februray 18, 2004	Fixed the return values for the OpenLocalSemaphore (page 89) function.
October 2002	Updated the documentation for the system (page 120) function, adding information about the maximum length for the command parameter.
May 2002	Added comments to Remarks section of SetNLMID (page 104) on the reasons not to call this function.
	Added information about true reentrant NLM programming functionality to "Shared Memory" on page 28.
February 2002	Updated links.
September 2001 (mid-release change)	Updated NWSMPIsLoaded (obsolete 9/2001) (page 86), NWThreadToMP (obsolete 9/2001) (page 87), and NWThreadToNetWare (obsolete 9/2001) (page 88) to be obsolete functions.
September 2001	Added support for NetWare 6.x to documentation.
June 2001	Changed classification of GetNLMIDFromNLMHandle (page 73) to be 5.x only.
	Changed example in FindNLMHandle (page 67).
	Added links to Section 1.5, "NetWare Global Data," on page 18.
	Made changes to improve document accessibility.
	Made minor formatting changes.
February 2001	Added "CLib" designation to delay (page 56) to distinguish it from setvbuf.
July 2000	Added documentation for GetNLMIDFromNLMHandle (page 73).
May 2000	Deleted the remaining SMP API documentation and changed the name of the documentation from *Thread and Multi-Processor Management* to *NLM Threads Management*.
	Added revision history.