Interacting with the Operating System

Cocoa > Process Management

2006-04-04

Apple Inc.
1 Infinite Loop
Cupertino, CA 95014
408-996-1010

Simultaneously published in the United States and Canada.

Contents

Introduction to Interacting with the Operating System

This topic describes a variety of classes that give you access to some of the functionality of the operating system. With them, you can launch subprocesses, obtain your process's environment variables, perform domain name lookups, locate the user's home directory, and more.

Organization of This Document

This document contains the following articles:

- "Host Information" (page 7) discusses how to perform domain name lookups.
- "Process Information" (page 9) discusses the types of information you can obtain about the current process.
- "Task Management" (page 11) discusses how to launch subprocesses and communicate with them.
- "Signals" (page 13) discusses operating-system signals and their behavior in processes.
- "Creating and Launching an NSTask" (page 15) shows an example of using an `NSTask` object.
- "Ending an NSTask" (page 17) discusses ways to detect when a task exits and how to terminate tasks before they are done.
- "Piping Data Between Tasks" (page 19) shows an example of how to move data from one task to another using pipes.

Limitations

Some classes are available for either Objective-C or Java, but not both. The functionality of those classes, though, are provided elsewhere in the other language.

Host Information

An NSHost object holds network name and address information for a host. You use this class to get the current host's name and address and to look up other hosts by name or by address. The class uses available network administration services (such as NetInfo or the Domain Name Service) to discover all requested names and addresses for the host. It does not attempt to contact the host itself, however. This avoids untimely delays due to a host being unavailable, but it may result in incomplete information about the host.

An NSHost object contains all of the network addresses and names discovered for a given host by the network administration services. Each NSHost object typically contains one unique address, but it may have more than one name. If the host has more than one name, the additional names are usually variations on the same name—typically the basic host name plus the fully qualified domain name. For example, with a host name "sales" in the domain "anycorp.com", an NSHost object can hold both the names "sales" and "sales.anycorp.com".

The NSHost class maintains a cache of previously created instances so that requests for an existing NSHost object return that object instead of creating a new one. Use the setHostCacheEnabled: method to turn the cache off, forcing lookup of hosts as they're requested. You can also use the flushHostCache method to clear the cache of its entries so that subsequent requests look up the host information and create new instances.

THe NSHost class is available in Objective-C only. Java developers can use the java.net.InetAddress class instead.

Process Information

The `NSProcessInfo` class provides methods to access process-wide information. An `NSProcessInfo` object can return such information as the current process's arguments, environment variables, host name, and process name.

The `NSProcessInfo` class is available in Objective-C only. In Java, the `NSSystem` class provides the same information as `NSProcessInfo` as well as information obtained from function calls in Objective-C. The `NSSystem` class object can return such additional information as the user's name, full name, and home directory. The class also provides a method, `log`, to send strings to `stderr`.

Task Management

Using the NSTask class, your program can run another program as a subprocess and can monitor that program's execution. An NSTask object creates a separate executable entity; it differs from NSThread in that it does not share memory space with the process that creates it.

A task operates within an environment defined by the current values for several items: the current directory, standard input, standard output, standard error, and the values of any environment variables. By default, an NSTask object inherits its environment from the process that launches it. If there are any values that should be different for the task, for example, if the current directory should change, you must change the value before you launch the task. A task's environment cannot be changed while it is running.

Arguments can be specified for the task you want to launch. These arguments do not undergo shell expansion, so you do not need to do special quoting, and shell variables, such as $PWD, are not resolved.

Your program can communicate with the task by attaching one or more NSPipe objects to the task's standard input, output, or error devices before launching the task. A pipe is a one-way communications channel between related processes; one process writes data while the other process reads that data. The data that passes through the pipe is buffered; the size of the buffer is determined by the underlying operating system. An NSPipe object represents both ends of a pipe.

The end points of the NSPipe object are instances of NSFileHandle. You read or write data from the appropriate NSFileHandle object to get the output from or send input to the task. Multiple tasks can be connected together by attaching an NSPipe object between one task's standard output and another task's standard input. The output from the first task is then automatically sent as input to the second task.

The task's standard input, output, and error devices can instead be attached to NSFileHandle objects directly to either provide the input data from a file or capture the output to a file.

If the task is an Objective-C Cocoa application, you can also communicate with it using the distributed objects system. For information on distributed objects, see Distributed Objects.

An NSTask object can be used to run its task only once. Subsequent attempts to run the task using the same object raise an error. While the task is running, you can send it terminate or interrupt signals (both cause termination by default). You can also suspend the task temporarily. When the task terminates, its exit status is recorded and NSTaskDidTerminateNotification is sent.

The NSTask class is available in Objective-C only. Java developers can use the java.lang.Runtime class to launch processes.

Signals

Signals are software interrupts that can be invoked on a specified process. The default signal handling behavior (provided by the system) usually terminates the process immediately on receipt of a signal. A process can override this behavior by installing a signal handler routine.

The most typical use of signals is by the kernel, which uses signals to notify a process of exceptional conditions such as invalid address errors and divide-by-zero errors. Another typical use is the command-line `kill` tool, which is capable of sending any user-specified signal to a process, though the most common use is to terminate a process with a hang-up signal (`SIGHUP`).

Because signals are complex to use effectively and they tend to behave differently (sometimes unreliably) on different operating systems, you should generally avoid installing signal handlers for your own applications. The default system handler usually provides the most appropriate response for a given signal. If you do want to handle signals in your application, see the `signal` man page for basic information about the signals that may be sent.

Creating and Launching an NSTask

There are two ways to create an `NSTask` object. If it is sufficient for the task to run in the environment that it inherits from the process that creates it, use the class method `launchedTaskWithLaunchPath:arguments:`. This method both creates and executes (launches) the task. If you need to change the task's environment, create the task using `alloc` and `init`, use `set...` methods to change parts of the environment, then use the `launch` method to launch the task. For example, the following method runs tasks that take an input file and an output file as arguments. It reads these arguments, the task's executable, and the current directory from text fields before it launches the task:

```
- (void)runTask:(id)sender
{
    NSTask *aTask = [[NSTask alloc] init];
    NSMutableArray *args = [NSMutableArray array];

    /* set arguments */
    [args addObject:[[inputFile stringValue] lastPathComponent]];
    [args addObject:[outputFile stringValue]];
    [aTask setCurrentDirectoryPath:[[inputFile stringValue]
            stringByDeletingLastPathComponent]];
    [aTask setLaunchPath:[taskField stringValue]];
    [aTask setArguments:args];
    [aTask launch];
}
```

If you create an `NSTask` object in this manner, you must be sure to set the executable name using `setLaunchPath:`. If you don't, an `NSInvalidArgumentException` is raised.

Ending an NSTask

Normally, you want the task that you've launched to run to completion. When the task exits, the corresponding `NSTask` object posts an `NSTaskDidTerminateNotification` to the default notification center. You can add one of the custom objects in your program as an observer of the notification and check the task's exit status (using `terminationStatus`) in the observer method. For example:

```
-(id)init {
    self = [super init];
    [[NSNotificationCenter defaultCenter] addObserver:self
            selector:@selector(checkATaskStatus:)
            name:NSTaskDidTerminateNotification
            object:nil];
    return self;
}

- (void)checkATaskStatus:(NSNotification *)aNotification {
    int status = [[aNotification object] terminationStatus];
    if (status == ATASK_SUCCESS_VALUE)
        NSLog(@"Task succeeded.");
    else
        NSLog(@"Task failed.");
}
```

If you need to force a task to end execution, send a `terminate` message to the `NSTask` object. If the `NSTask` object gets released, however, `NSTaskDidTerminateNotification` does not get sent, as the port the message would have been sent on was released as part of the task release.

Piping Data Between Tasks

Each end point of the pipe is a file descriptor, represented by an `NSFileHandle` object. You thus use `NSFileHandle` messages to read and write pipe data. A "parent" process creates the `NSPipe` object and holds one end of it. It creates an `NSTask` object for the other process and, before launching it, passes the other end of the pipe to that process; it does this by setting the standard input, standard output, or standard error device of the `NSTask` object to be the other `NSFileHandle` or the `NSPipe` itself (in the latter case, the type of `NSFileHandle`—reading or writing—is determined by the "set" method of `NSTask`).

Note: The file descriptors used by a pipe count against the maximum number of open file descriptors allowable in a task. In Mac OS X v10.4, the maximum number of open file descriptors is approximately 10240 but in older versions of Mac OS X, this number is much smaller (256 in Mac OS X v10.2).

The following example illustrates the above procedure:

```
- (void)readTaskData:(id)sender
{
    NSTask *pipeTask = [[NSTask alloc] init];
    NSPipe *newPipe = [NSPipe pipe];
    NSFileHandle *readHandle = [newPipe fileHandleForReading];
    NSData *inData = nil;

    // write handle is closed to this process
    [pipeTask setStandardOutput:newPipe];
    [pipeTask setLaunchPath:[NSHomeDirectory()
            stringByAppendingPathComponent:
                    @"PipeTask.app/Contents/MacOS/PipeTask"]];
    [pipeTask launch];

    while ((inData = [readHandle availableData]) && [inData length]) {
        [self processData:inData];
    }
    [pipeTask release];
}
```

The launched process in this example must get data and write that data (using the `writeData:` method of `NSFileHandle`), to its standard output device, which is obtained using the `fileHandleWithStandardOutput` method of `NSFileHandle`.

When the processes have no more data to communicate across the pipe, the writing process should simply send `closeFile` to its `NSFileHandle` end point. This causes the process with the "read" `NSFileHandle` to receive an empty `NSData` object, signalling the end of data. If the "parent" process created the `NSPipe` object with the `init` method, it should then release it.

Document Revision History

This table describes the changes to *Interacting with the Operating System*.

Date	Notes
2006-04-04	Updated the task management guidelines and added some high-level information about signals.
2002-11-12	Revision history was added to existing topic. It will be used to record changes to the content of the topic.