System Library

Written by Reid Spencer

Abstract

This document provides some details on LLVM's System Library, located in the source at `lib/System` and `include/llvm/System`. The library's purpose is to shield LLVM from the differences between operating systems for the few services LLVM needs from the operating system. Much of LLVM is written using portability features of standard C++. However, in a few areas, system dependent facilities are needed and the System Library is the wrapper around those system calls.

By centralizing LLVM's use of operating system interfaces, we make it possible for the LLVM tool chain and runtime libraries to be more easily ported to new platforms since (theoretically) only `lib/System` needs to be ported. This library also unclutters the rest of LLVM from #ifdef use and special cases for specific operating systems. Such uses are replaced with simple calls to the interfaces provided in `include/llvm/System`.

Note that the System Library is not intended to be a complete operating system wrapper (such as the Adaptive Communications Environment (ACE) or Apache Portable Runtime (APR)), but only provides the functionality necessary to support LLVM.

The System Library was written by Reid Spencer who formulated the design based on similar work originating from the eXtensible Programming System (XPS). Several people helped with the effort; especially, Jeff Cohen and Henrik Bach on the Win32 port.

Keeping LLVM Portable

In order to keep LLVM portable, LLVM developers should adhere to a set of portability rules associated with the System Library. Adherence to these rules should help the System Library achieve its goal of

shielding LLVM from the variations in operating system interfaces and doing so efficiently. The following sections define the rules needed to fulfill this objective.

Don't Include System Headers

Except in `lib/System`, no LLVM source code should directly `#include` a system header. Care has been taken to remove all such `#includes` from LLVM while `lib/System` was being developed. Specifically this means that header files like "unistd.h", "windows.h", "stdio.h", and "string.h" are forbidden to be included by LLVM source code outside the implementation of `lib/System`.

To obtain system-dependent functionality, existing interfaces to the system found in `include/llvm/System` should be used. If an appropriate interface is not available, it should be added to `include/llvm/System` and implemented in `lib/System` for all supported platforms.

Don't Expose System Headers

The System Library must shield LLVM from *all* system headers. To obtain system level functionality, LLVM source must `#include "llvm/System/Thing.h"` and nothing else. This means that `Thing.h` cannot expose any system header files. This protects LLVM from accidentally using system specific functionality and only allows it via the `lib/System` interface.

Use Standard C Headers

The *standard* C headers (the ones beginning with "c") are allowed to be exposed through the `lib/System` interface. These headers and the things they declare are considered to be platform agnostic. LLVM source files may include them directly or obtain their inclusion through `lib/System` interfaces.

Use Standard C++ Headers

The *standard* C++ headers from the standard C++ library and standard template library may be exposed through the `lib/System` interface. These headers and the things they declare are considered to be platform agnostic. LLVM source files may include them or obtain their inclusion through lib/System interfaces.

High Level Interface

The entry points specified in the interface of lib/System must be aimed at completing some reasonably high level task needed by LLVM. We do not want to simply wrap each operating system call. It would be preferable to wrap several operating system calls that are always used in conjunction with one another by LLVM.

For example, consider what is needed to execute a program, wait for it to complete, and return its result code. On Unix, this involves the following operating system calls: `getenv, fork, execve,` and `wait`. The correct thing for lib/System to provide is a function, say `ExecuteProgramAndWait`, that implements the functionality completely. what we don't want is wrappers for the operating system calls involved.

There must *not* be a one-to-one relationship between operating system calls and the System library's interface. Any such interface function will be suspicious.

No Unused Functionality

There must be no functionality specified in the interface of lib/System that isn't actually used by LLVM. We're not writing a general purpose operating system wrapper here, just enough to satisfy LLVM's needs. And, LLVM doesn't need much. This design goal aims to keep the lib/System interface small and understandable which should foster its actual use and adoption.

No Duplicate Implementations

The implementation of a function for a given platform must be written exactly once. This implies that it must be possible to apply a function's implementation to multiple operating systems if those operating systems can share the same implementation. This rule applies to the set of operating systems supported for a given class of operating system (e.g. Unix, Win32).

No Virtual Methods

The System Library interfaces can be called quite frequently by LLVM. In order to make those calls as efficient as possible, we discourage the use of virtual methods. There is no need to use inheritance for implementation differences, it just adds complexity. The `#include` mechanism works just fine.

No Exposed Functions

Any functions defined by system libraries (i.e. not defined by lib/System) must not be exposed through the lib/System interface, even if the header file for that function is not exposed. This prevents inadvertent use of system specific functionality.

For example, the `stat` system call is notorious for having variations in the data it provides. `lib/System` must not declare `stat` nor allow it to be declared. Instead it should provide its own interface to discovering information about files and directories. Those interfaces may be implemented in terms of `stat` but that is strictly an implementation detail. The interface provided by the System Library must be implemented on all platforms (even those without `stat`).

No Exposed Data

Any data defined by system libraries (i.e. not defined by lib/System) must not be exposed through the lib/System interface, even if the header file for that function is not exposed. As with functions, this prevents inadvertent use of data that might not exist on all platforms.

Minimize Soft Errors

Operating system interfaces will generally provide error results for every little thing that could go wrong. In almost all cases, you can divide these error results into two groups: normal/good/soft and abnormal/bad/hard. That is, some of the errors are simply information like "file not found", "insufficient privileges", etc. while other errors are much harder like "out of space", "bad disk sector", or "system call interrupted". We'll call the first group "*soft*" errors and the second group "*hard*" errors.

lib/System must always attempt to minimize soft errors. This is a design requirement because the minimization of soft errors can affect the granularity and the nature of the interface. In general, if

you find that you're wanting to throw soft errors, you must review the granularity of the interface because it is likely you're trying to implement something that is too low level. The rule of thumb is to provide interface functions that *can't* fail, except when faced with hard errors.

For a trivial example, suppose we wanted to add an "OpenFileForWriting" function. For many operating systems, if the file doesn't exist, attempting to open the file will produce an error. However, lib/System should not simply throw that error if it occurs because its a soft error. The problem is that the interface function, OpenFileForWriting is too low level. It should be OpenOrCreateFileForWriting. In the case of the soft "doesn't exist" error, this function would just create it and then open it for writing.

This design principle needs to be maintained in lib/System because it avoids the propagation of soft error handling throughout the rest of LLVM. Hard errors will generally just cause a termination for an LLVM tool so don't be bashful about throwing them.

Rules of thumb:

1. Don't throw soft errors, only hard errors.
2. If you're tempted to throw a soft error, re-think the interface.
3. Handle internally the most common normal/good/soft error conditions so the rest of LLVM doesn't have to.

No throw Specifications

None of the lib/System interface functions may be declared with C++ `throw()` specifications on them. This requirement makes sure that the compiler does not insert additional exception handling code into the interface functions. This is a performance consideration: lib/System functions are at the bottom of many call chains and as such can be frequently called. We need them to be as efficient as possible. However, no routines in the system library should actually throw exceptions.

Code Organization

Implementations of the System Library interface are separated by their general class of operating system. Currently only Unix and Win32 classes are defined but more could be added for other operating system classifications. To distinguish which implementation to compile, the code in lib/System uses the LLVM_ON_UNIX and LLVM_ON_WIN32 #defines provided via configure through the llvm/Config/config.h file. Each source file in lib/System, after implementing the generic (operating system independent) functionality needs to include the correct implementation using a set of `#if defined(LLVM_ON_XYZ)` directives. For example, if we had lib/System/File.cpp, we'd expect to see in that file:

```
#if defined(LLVM_ON_UNIX)
#include "Unix/File.cpp"
#endif
#if defined(LLVM_ON_WIN32)
#include "Win32/File.cpp"
#endif
```

The implementation in lib/System/Unix/File.cpp should handle all Unix variants. The implementation in lib/System/Win32/File.cpp should handle all Win32 variants. What this does is quickly differentiate the basic class of operating system that will provide the implementation. The specific details for a given platform must still be determined through the use of `#ifdef`.

Consistent Semantics

The implementation of a lib/System interface can vary drastically between platforms. That's okay as long as the end result of the interface function is the same. For example, a function to create a directory is pretty straight forward on all operating system. System V IPC on the other hand isn't even supported on all platforms. Instead of "supporting" System V IPC, lib/System should provide an interface to the basic concept of inter-process communications. The implementations might use System V IPC if that was available or named pipes, or whatever gets the job done effectively for a given operating system. In all cases, the interface and the implementation must be semantically consistent.

Bug 351

See bug 351 for further details on the progress of this work

Reid Spencer
LLVM Compiler Infrastructure
Last modified: $Date: 2011-11-03 01:43:23 -0500 (Thu, 03 Nov 2011) $