The journal of Apple technology.

Volume Number: 21 (2005)
Issue Number: 6
Column Tag: Programming

Indexing Darwin Files

Extending Spotlight's Reach

by Rich Morin

Spotlight is quite willing to search the user's home directory tree. It also looks into some other areas, including /Applications, /Developer, /Library, and /System. It stops short, however, at looking into the "Darwin" directories (e.g., /bin, /private, /sbin, and /usr).

As a result, many interesting files (e.g., control and include files, man pages) are not indexed. This seems like an unfortunate limitation. Most users aren't interested in these files, but some of us (e.g., administrators, power users, and programmers) find them to be very useful, indeed. Wouldn't it be nice to be able to use Spotlight to index these files?

In fact, it is possible to override Spotlight's built-in restrictions. Each volume's root directory contains a sub-directory (named .Spotlight-V100 in Mac OS X version 10.4). This contains a few "database" files and a property list file (_rules.plist). By editing this property list, you can instruct Spotlight to include or exclude directories.

Just locating a file isn't enough, however; we also want to be able to view it. And, if the computer can find connections between the files, some navigation aids (e.g., hyperlinks, linkage diagrams) would be lovely to have. Although the interface could be implemented in Cocoa, HTML is an arguably simpler (and far more portable) approach.

As an experiment in extending Spotlight's reach, I decided to write Morinfo, a suite of scripts that allows man pages and include files to be located by Spotlight and presented by a web browser. The results are useful, even in their current state, and offer interesting possibilities for future enhancements.

Background and Competition

Turning man pages into HTML is not a new idea. There are existing facilities (e.g., Internet servers and downloadable applications) that perform this function. However, their approaches (and consequently, results) vary from mine.

The FreeBSD Project (http://www.freebsd.org) provides access to the "FreeBSD Hypertext Man Pages" (http://www.FreeBSD.org/cgi/man.cgi). Despite their name, these pages cover a variety of operating systems (including Darwin). Of course, the site cannot display arbitrary man pages that a user might have installed locally.

The pages are created by a free (i.e., libre) Perl CGI script, available via the FAQ. The script runs the man(1) command on an appropriate input file, reformatting the results into HTML. Search is available within the CGI; Internet search engines (e.g., Google) also index the pages (e.g., FreeBSD chmod).

Bwana (http://www.bruji.com/bwana) is a free (i.e., gratis) application which bills itself as a "man page viewer for

Safari, Firefox, Camino & IE". To use Bwana, the user enters a URL into one of these browsers, using the scheme man (e.g., man:chmod). Bwana puts the generated HTML in a temporary file (e.g., file:///...) and tells the browser to display it.

Because Bwana generates its web pages as files, it cannot support a forms-based "search" facility. And, because the files are temporary, Spotlight will not index them. Bwana does, however, provide an "index page" (accessible via man:) which lists the known man pages (with short descriptions, if available).

Both of these facilities provide links to other man pages, but only within a limited context. Bwana only links to pages on the local machine; the FreeBSD site only links to pages for the current OS variant and version. Neither facility links to anything other than man pages (e.g., include files).

Objective and Approach

My initial objective was to make the files convenient to use and searchable by Spotlight. The usage requirement was met by handing the files (suitably edited) to the user's default web browser. The "search" requirement was met by (a) using an extension that Spotlight indexes and (b) putting the files into a directory that is traversed by Spotlight.

HTML makes it possible to generate hyperlinks, using implicit linkage information that is present in the files. The entries in a man page's "SEE ALSO" section, for example, can become links to other man pages. References to include files (e.g., in sections 2 and 3) can also be turned into links.

Even with the addition of client-side technologies (e.g., Java, Javascript, SVG), web pages do not compete with Aqua for attractiveness, interactive performance and richness, etc. On the other hand, they have significant advantages for this (exploratory) project.

HTML gives me a familiar and powerful user interface, with both local and Internet-wide navigation. Better yet, it requires very little effort on my part. Finally, HTML is a much more portable technology than, say, Cocoa.

Resource Usage

Careful Reader may have noted that I haven't discussed the resource usage of this approach, whether in online storage or processing time. This is not because there isn't any; indeed, it uses hundreds of megabytes of storage and (initially) several hours of processing time. However, I don't think this will matter for most of our readers.

The storage impact mirrors, almost exactly, the storage used by the original files. I add a tiny bit of HTML, of course, but this mostly gets lost in the noise. The edited man pages and include files (including BSD, Cocoa, X11, etc.) occupy about 350 MB of storage. Even if the edited versions used a gigabyte, most users wouldn't notice the impact.

The processing time, as noted above, is quite substantial. On a B&W G3, the initialization script runs for an hour or so. Spotlight's importers also take a while to index the material. On the other hand, these activities have little impact on the interactive performance of the machine, so what's the problem?

Online storage and background processing time, in the amounts Morinfo uses, are essentially free. If having a nicely indexed and cross-linked set of reference pages is useful, most users will gladly provide the necessary resources.

File-based HTML

My first attempt created HTML files in a folder that Spotlight is known to search. Mac OS X "knows" that files with an extension of html should be viewed by a web browser, so double-clicking these files "Just Worked". Unfortunately, this approach has significant limitations.

When an HTML file is double-clicked in the Finder, the resulting browser window shows a URL of the form file:///.... This indicates that the browser is reading the file directly (rather than through a web server).

Because no web server is part of the process, HTML files cannot make use of CGI scripts, HTTP header magic (e.g., cookies, redirects), Server-Side Includes (SSIs), or other server-based technologies. They can, however, take advantage of many client-side features (e.g., CSS, images and client-side image maps, Javascript).

An HTML file can also link to any type of web page, located anywhere on the Web. So, for example, it could link to a CGI-based page that post-processes the file itself, adding information or dynamic capabilities. I considered this approach, but (fortunately) found a cleaner solution.

CGI-mediated HTML

I now create "template files", using a CGI script to post-process them into web pages. This is more complex than my initial approach, but it allows me to use the full power of the web server. The following diagram summarizes our data and control flow, in the context of Spotlight and the Finder:

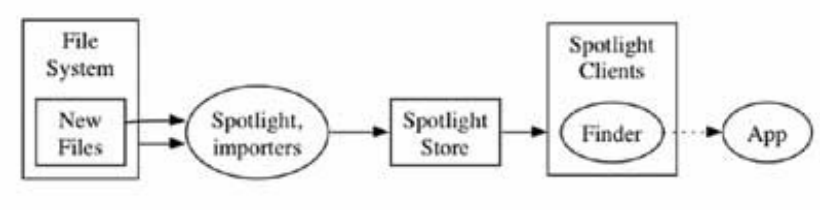

Figure 1. Data and control flow

Spotlight and its importers gather information from the file system and from new (or changed) files. This information goes into the Spotlight Store, where it can be accessed by Spotlight clients (e.g., the Finder). Similarly, m_init reads include files and manual pages, generating template files for use by morinfo.

The template files are "new files", stored in an area that Spotlight indexes, so any keywords found in them are entered into the Spotlight Store. This means that the user can find relevant template files, using the Finder's Spotlight mode.

If the user double-clicks on a template file, m_link gets called into action. Using the "POSIX path" of the template file, it directs the user's browser to display a generated URL. This causes morinfo to start up, read the template file, and produce a web page.

m_link

m_link is a tiny snippet of AppleScript, but it performs a vital service. When a template file is double-clicked (i.e., "opened"), m_link generates a URL that (a) invokes the CGI script and (b) includes the path to the template file.

The initial version of the AppleScript code below was provided by Chris Nandor, of Mac::Glue and MacPerl fame. The on open block is run when the script is asked to open a file. The enclosed code retrieves the file's "POSIX path", appends it to a prefix string, and hands it to an open location command:

```
-- m_link
on open this_item
  set filepath to POSIX path of this_item
  set pre to "http://localhost/cgi-bin/"
```

```
  set pre to pre & "morinfo?file="
  open location pre & filepath
end open
```

Launch Services

When a file is double-clicked, Launch Services attempts to determine which application should be "launched" to work on it. Traditionally, Mac OS files had Creator and Type codes (stored in their resource forks) which supplied this information. More recently, Apple has allowed file extensions to be linked to particular applications.

Because my files are managed by command-line tools (e.g., perl, vi), creating and maintaining resource forks would be quite a nuisance. So, I use the second method, asserting priority over any file with a custom extension (.ht4m - "HTML Template for Morinfo"). Any user can assert this priority, using the Finder's File > Get Info dialog, but the process is a bit involved:

- Click on a representative file, selecting it.
- Type Command-I (or use the Finder's File > Get Info menu item).
- In the Open with: section of the dialog, navigate to Other..., then to the desired application.
- Click on the Change All... button, telling the system to use the same script for all files with this extension.
- Close the dialog.

I didn't want to burden our users with all this effort, just to get Morinfo installed. So, I put some assertions into m_link's Info.plist file:

```
<key>CFBundleTypeExtensions</key>
<array>
  <string>ht4m</string>
</array>
<key>LSIsAppleDefaultForType</key>
<true/>
```

My installation script copies a "Morinfo" directory into /Applications, using CPMac(1). During the course of this copy, the OS examines m_link's Info.plist file and notes these assertions.

Template Files

My template files are (typically) stored in /Library/Documentation/Morinfo. To support the widest possible range of initial files, I append the full path name of the initial file onto the base path. I also add an intervening level (e.g., HF, MP), to designate the "name space" (e.g., Header File, Manual Page) that is being displayed.

We also adjust the file names in assorted ways (e.g., removing any gz extensions, adding a consistent extension of our own). The resulting path names look like

```
/Library/Documentation/Morinfo/HF/
  usr/include/stdio.h.ht4m
/Library/Documentation/Morinfo/MP/
  usr/share/man/man1/cp.1.ht4m
```

The editing of the file content depends on the format of the initial file. In the case of include files, I simply create a navigational header and enclose the include file text in a PRE block in an HTML page. I also insert some "placeholder" strings for the CGI script to expand, as appropriate. Finally, as discussed below, I generate off-page

links.

In the case of man pages, I also need to (expand and) format the pages into an ASCII representation. Fortunately, the man(1) command does this quite readily. I then turn any nroff(1) backspace sequences (e.g., f^Hfo^Hoo^Ho) into appropriate HTML sequences (e.g., <B>foo</B>).

Link Generation

The final step is to recognize references to other pages and convert them into links. Include files and man pages have fairly predictable formats, so recognizing SEE ALSO entries and #include references isn't particularly challenging. Conversion into links, on the other hand, can be.

Because of Darwin's eclectic nature, man pages have somewhat irregular names (e.g., ls.1, cal.1.gz, tic.1m, md5.1ssl). When I adjust the names of the man pages, I normalize these names. This simplifies the process of generating links to man pages, because I don't have to keep track of oddball extensions.

Generating links for #include references is where things get sticky. References don't specify the exact directory where the include file is located. At most, they provide hints (e.g., specifying a parent directory or a hint about the file's Framework). Also, many include files enclose #include references within conditional blocks.

None of this presents a problem for the compiler; Xcode and/or make(1) files keep track of include directories and pre-processor definitions, handing them to compilers as needed. In my case, however, I have no such help.

m_init deals with the problem in a somewhat "heuristic" manner. First, it scans directories where include files reside, recording the full path names of all files found. Armed with this information, it can make "educated guesses" about the path name that matches a given "hint".

If a conflict arises, an error message is generated. This allows the programmer to add an "override" rule to deal with the ambiguity. No, it's not very elegant, but ambiguous information is a common problem in automated metadata processing. Elegance is nice, but functionality is the overriding concern.

Incidentally, the use of offline processing allows me to do some cute things with the link information that I detect. For example, I could add "back links" (e.g., from an include file's page to any pages that reference it). I could also generate "context diagrams" to assist the user in understanding the (local) link structure. These features are currently under development.

Contextual Issues

By double-clicking on a file, the user jumps into a different "context". In our case, the new context is a web page (e.g., in Safari). This can cause confusion for the unwary: if the user makes a follow-on search, using the browser's "magnifying glass" text area, she might be expecting a Spotlight search to occur.

Imagine her dismay when a Google (i.e., Internet-wide) search happens, instead. To be fair, the browser's magnifying glass is accompanied by the word "Google". On the other hand, the user might be busy, or distracted, or...

Even disregarding the issue of confusion, how is the user supposed to search local files from within our web page? The "obvious" answer is to provide a forms-based interface to Spotlight, but this might cause even more confusion. Sigh.

Privacy and Security

The first incarnation of Morinfo copied (edited forms of) system files to a different location on the local file system.

This did not introduce any obvious security holes, but it wasn't very powerful, either. The current version is more powerful, but it introduces some possible privacy and security holes:

- Web server - Web servers can have exploitable bugs. Administrative errors and oversights can allow security holes.
- CGI script - CGI scripts can have exploitable bugs.
- Publication - Publishing local files on the web may reveal private information.

If the server does not need to be visible from other machines, a firewall can provide some protection. Mac OS X has a built-in software firewall which should be turned on and configured to provide the desired balance between security and convenience. Unfortunately, at this writing, it does not allow Personal Web Sharing (i.e., Apache) to be blocked. Strong protection can be provided, however, by the use of a 10-BaseT "router" (e.g., using NAT).

Some applications (e.g., CUPS) provide their own HTTP servers, using different ports than the default 80. So, a firewall can block access to CUPS, while allowing access to the "normal" web server offerings. Unfortunately, there is nothing to prevent two programs from claiming the same port number.

CGI scripts receive untrusted information from queries, cookies, etc. Scripts should check all incoming information, making sure that only legitimate requests are honored. The "Taint checking" feature (e.g., in Perl) can ensure that the script has done this, but it cannot tell whether the script's testing code was bulletproof.

Although read-only access to "Darwin files" seems pretty innocent, complexities can arise. For example, a developer might put proprietary files into a "system" area, then be unpleasantly surprised when the files show up on Google.

Current Status

The current version of Morinfo is largely a "proof of concept" implementation of a semi-mechanized tool for documentation collection and mechanization. The functions it employs or provides include:

- Collection (e.g., File System to Spotlight Store) - Spotlight (and its importers) perform dynamic collection of metadata about files and the file system. Morinfo analyzes and reformats two types of files: man pages and include files.
- Search (Query to File List) - Spotlight (and its clients) use the Spotlight Store's metadata for analysis, search, etc.
- Launch (File to Application) - Launch Services provides assorted ways to launch an application, based on the attributes of a double-clicked file. Morinfo uses this to bind its generated files to an AppleScript (m_link).

Morinfo uses m_link.app'sInfInfo.plist file to assert jurisdiction of all files which have an unusual extension: ht4m. Some Mac programmers dislike this technique, but it has the advantage of not requiring the creation and maintenance of resource forks.

- UI Startup - m_link adds a preface string to the "POSIX path" of the double-clicked file, creating a URL string. It uses this to "invoke" an arbitrary application (the user's default web browser), directed to a particular document (i.e., the file).
- Presentation - morinfo (the CGI script) uses the file as a template, adding eye candy, navigational aids, etc.
- Feedback - morinfo's web pages include mailto links whose Subject lines reflect the current topic, etc.

Connections

To review a bit, Morinfo makes use of some interesting "connections". Mac OS X provides some of these by default; others require a bit of hacking:

- File content is connected (via importers) to the Spotlight Store.
- The Spotlight Store is connected (via the Spotlight APIs and clients) to user queries.
- File icons (e.g,, in query results) are connected to "appropriate applications", as specified in Get Info dialogs, Info.plist files, etc..
- An AppleScript application can launch another application, feeding arguments to it as needed.
- A CGI script can connect system files to manual pages.

Because of "network effects" (http://en.wikipedia.org/wiki/Network_effect), each new connection takes advantage of existing connections (and makes them more powerful). At the same time, new connections may expose existing resources to new security holes.

Wish List

There are any number of directions in which this work could be extended. Let's begin with the functions listed above:

- Collection - Morinfo needs to have an importer, in order to track changes and additions to its monitored file areas. Spotlight tracks files, but not processes. If Mac OS X supported DTrace (Sun's Open Source data collection system), it could collect metadata at arbitrary system call boundaries.
- Search - Spotlight's query language is missing features such as regular expressions and "order" comparisons for strings. The Finder's "Spotlight mode" lacks convenient ways to specify compound matches or exclusions. As programmers and users make their wishes known, we can expect to see some enhancements.
- Launch - The current (extension-based) link has philosophical problems for some. There may be a "cleaner" way to do this.
- UI Startup - m_link is acceptably sprightly (even on a G3!), easy to modify, and quite flexible.
- Presentation - morinfo is almost a "blank slate" at this point. I foresee diagrams (for context and navigation), interaction support, and much more.
- Feedback - Morinfo needs a wiki-like subsystem to collect page-specific user feedback. Sharing of locally-collected data (e.g., for trend analysis) is also possible.

Rich Morin has been using computers since 1970, Unix since 1983, and Mac-based Unix since 1986 (when he helped Apple create A/UX 1.0). When he isn't writing this for MacTech, Rich provides documentation and programming services, specializing in mechanized document production. Feel free to write to Rich at rdm@cfcl.com.