The journal of Apple technology.

Volume Number: 19 (2003)
Issue Number: 3
Column Tag: Mac OS X

File-Based Dataflow

building robust systems without explicit file locking

by Rich Morin

Last month, I discussed "File Change Watcher", a Perl daemon that wakes up periodically, checks for file-system changes, copies files, and gathers metadata. To save space, I ignored a critical issue: how can another process know that a metadata file is "ready" to be read?

Race Conditions

This question comes up in the design of any system where one process is writing a file and another is accessing it. In this type of "race condition", the reading process can overtake the writing process, arriving prematurely at the end of the data (and ignoring any data the other process may write thereafter).

Alternatively, if two processes try to write the same file at the same time, assorted damage can ensue. Depending on the circumstances, changes might get lost, output could get intermingled, etc.

One way to deal with the problem is to use a "lock file". By common agreement, if the lock file is in place, the file it "locks" isn't available for access. An inquiring process can test for this by trying to create the lock file. If the process fails, it just waits for a while before trying again.

The reason that this test works is that the kernel won't let two processes open the same file with exclusive access (O_EXCL). So, the first attempt succeeds and all others fail (until the first process closes the file). The technique works quite well, as long as everyone plays nicely; the BSD side of OSX contains a number of lock files (e.g., /var/spool/lock).

In the general case, lock files (or some equivalent technique) are necessary. If, for instance, two users want to edit a file, you really don't want them doing it at the same time. So, some Unix editors (e.g., the BSD version of vi) implement file locking.

Unfortunately, lock files add complexity and room for error. All of the processes have to honor the lock files; what if an existing program doesn't want to play? Also, if the system crashes, the lock file has to be explicitly removed when things start up again. And, for all of this pain, they only solve one part (simultaneous access) of the file-based dataflow problem set.

Atomic Actions

Fortunately, there are a number of alternatives to lock files. Most of them are based on some kind of "atomic action"; that is, something that can only be done completely or not at all. BSD provides several atomic actions as system calls, including chmod(2), chown(2), link(2), open(2), and rename(2).

These same facilities are available from Perl, but its chmod and chown are only atomic for single file nodes. Corresponding shell commands are available, albeit with some caveats:

- chgrp(1), chmod(1), chown(1) of a single file system node
- link(1), but only for hard links
- mv(1), to another name on the same file system

So, although you can't assume that a process will finish writing before some other process starts to read the output file, you can safely rename(2) an existing file (or mv(1) it to another name on the same file system) without worrying about race conditions.

Because any hard link to a file is simply a directory entry that points to a common inode(5), both the data and file system metadata are shared among a set of hard links. This lets a single atomic action (e.g., chmod) change the status of any number of links.

Finally, Cocoa's Application Kit Framework's NSFileWrapper class includes methods such as writeToFile:atomically:updateFilenames:. In short, a wealth of solutions is at hand.

Emitters and Consumers

The system I'm building is based on a data-flow model, similar to Unix pipelines, but it uses files instead of pipes. The method is far from new; mail transfer agents and print spoolers use variations of it in OSX. I am simply generalizing the idea into a framework for creating sets of file- and time-based tasks.

By specifying that only one program will ever write to a file, I can eliminate the issue of simultaneous writing. This means that I only need to prevent processes from opening or removing files prematurely and make sure that every file gets processed to completion.

The rules below allow the safe (i.e., no race conditions) use of files by any number of "emitters" and "consumers", without the need for lock files.

- Files are created and written by emitters, read and deleted by consumers. No modification of files is allowed, including appending or read/write usage.
- Files are created under temporary names (on the destination file system), then "published" (e.g., renamed) for use by consumers.
- A file can only be used by one consumer, which removes it just before exiting.

  Note: This restriction applies only to consumers. Other programs (e.g., more) may read published files at any time. Also, the consumer is not required to read the file, just remove it.

- If an emitter is creating files for multiple consumers, a separate link must be created for each consumer. All of the links must then be published in a single, atomic operation. One method uses a common directory:
    - Create a temporary directory.
    - For each file with N consumers, make N-1 links in the directory.
    - Rename the temporary file(s) into the directory, as the Nth file link(s).
    - Rename the temporary directory, publishing all of the links.

      Another method, which can only "protect" a single output file, has the advantage that the output links do not have to be in the same directory:

    - Turn off read access (using chmod) on the temporary file.
    - For a file with N consumers, make N-1 links.
    - Rename the temporary file as the Nth link.
    - Make any (and thereby, every) link readable.

Although these rules might complicate the life of programmers, the resulting programs aren't any more complicated. In fact, they tend to be quite simple: discover an input file, process it (writing any output to temporary files); when you're done, publish (e.g., rename) the output files and remove the input file.

Better yet, the file-discovery and -management details can be handed off to a scheduling daemon, allowing most of the "tasks" (working code) to be written as "filters": read from standard input; write to standard output.

A Data Collection System

Let's apply this to a data collection system and see how it plays out. A ps(1)-monitoring task is supposed to run once a minute, writing a report. Another task reads the report, writing a YAML (www.yaml.org) version. Other tasks produce hourly and daily summaries.

This is a fairly complex set of tasks, but it's only a small fraction of the workload for a full-scale operating system monitor. So, the amount of specification for each task should be as simple as possible. Here's a first cut at a configuration file:

```
# Every minute, collect raw ps(1) data.
{ ps_raw, type: cron, min: every,
  out: 'ps/raw/$time'
}
# Process the raw ps(1) data.
# Create two output links.
{ ps_rare, type: file, patt: 'ps/raw/*',
  out: ['ps/rare/1.$time',
        'ps/rare/2.$time' ]
}
# Every hour, create a summary.
{ ps_hour, type: cron, hour: every,
  out: 'ps/hour/$time'
}
# Every day, create a summary.
```

```
{ ps_day, type: cron, day: every,
  out: 'ps/day/$time'
}
```

This is fairly concise, but there's a lot of repetition. We could "boil it down" by taking advantage of the fact that it describes a tree of processes:

```
{ ps_raw, type: cron, min: every,
  out: 'ps/raw/$time',
  { ps_rare, type: file, patt: 'ps/raw/*',
    out: ['ps/rare/1.$time',
          'ps/rare/2.$time' ]
    { ps_hour, type: cron, hour: every,
      out: 'ps/hour/$time'
    },
    { ps_day, type: cron, day: every,
      out: 'ps/day/$time'
} } }
```

But that only kills off two lines (excluding comments), so it's not a huge win. Also, I'm not convinced that it's as easy to read, modify, etc. If we are willing to let the scheduling infrastructure generate file names for us, however, we can get away with "idioms" like:

```
{ ps_raw,      type: cron, min:  every,
  { ps_rare,   type: file,
    { ps_hour, type: cron, hour: every },
    { ps_day,  type: cron, day:  every }
} }
```

That brings the overhead back under control. And, if we need to access a file, we can follow standardized naming rules to find it. The output of ps_hour, for instance, would have a name of the form "ps_hour/1.<time>".

So much for theoretical hand-waving and speculative descriptions. Next month, I'll discuss some daemons that can actually make all of this work.

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 column, Rich runs Prime Time Freeware (www.ptf.com), a publisher of books and CD-ROMs for the Free and Open Source software community. Feel free to write to Rich at rdm@ptf.com.