News Need Not Be Slow

Geoff Collyer

Department of Statistics[*]
University of Toronto
utzoo!utcsri!utfraser!geoff

Henry Spencer

Zoology Computer Systems
University of Toronto
utzoo!henry

C news is a re-write, from scratch, of the ‘transport layer’ of the Usenet software. C rnews runs at over 19 times the speed of B rnews; C expire runs in minutes rather than the hours taken by B expire. These performance improvements were (mostly) quite simple and straightforward, and they exemplify general principles of performance tuning.

1. History and Motivation

In the beginning (of Usenet) (1979) was A news, written at Duke University by Steve Bellovin, Stephen Daniel, Tom Truscott and others. A single program, *news* , received, relayed, perused and cleaned out news articles. All articles were stored in a single UNIX[†] directory, which made A news suitable for local news and low volumes of network news. News articles were exchanged using a simple message format in which the first five lines of a message contained the information nowadays found in the article headers: unique article-id, newsgroup(s), author, subject, date posted.

As Usenet began to grow (1981), people in and around the University of California at Berkeley, including Matt Glickman and Mark Horton, modified A news extensively. The articles of each newsgroup were now stored in a separate directory. The message format was changed from the rigid and inextensible A news header format to one conforming to ARPA RFC 822 (the current ARPA mail-message format standard). *News* was broken into separate programs: *readnews*, *inews* (aka *rnews*), and *expire*. The authors dubbed the result ‘‘B news’’. Since the release of B news, it has replaced A news almost[‡] everywhere on Usenet.

* Work done mostly while at U of T Computing Services.
† UNIX is a registered trademark of AT&T.
‡ AT&T Bell Laboratories Research still runs A news for local newsgroups.

It soon became clear that sending individual articles from machine to machine as separate *uucp* transactions was unacceptably slow, in part because it produced large *uucp* spool directories, which are searched quite slowly* by the kernel. Sites began to *batch* articles into batches of (typically) 50,000−100,000 bytes for transmission to other machines.

At about this time, B news was changed to file news articles in a tree, as (for example) /usr/spool/news/net/women/only/123, rather than as /usr/spool/news/net.women.only/123. The motive for this was primarily elimination of problems with long newsgroup names, but shortening directories (and thus speeding searches) was also a consideration.

As Usenet traffic continued to grow explosively, sites began to use data compression on news batches. The main objective was to reduce expensive long-distance phone time, but again performance improved a bit: the extra processor time used for compression and decompression was more than gained back by the reduction in processor time used by *uucp* itself.

Unfortunately, B news has been modified by many people since 1981, and has mutated repeatedly to match the changing nature of Usenet. It has become complex, slow, and difficult to maintain.

During 1985, we observed that the nightly arrival of new news and expiry of old news were consuming resources far out of proportion to the volume of data involved†. *Expire* often ran for 90 minutes, and *rnews* processing averaged 10 seconds or more per article. Both programs tended to cripple systems by performing much disk i/o and eating much system-mode CPU time. **Utcs** was running B 2.10.1 news then and **utzoo** was running B 2.10 news. Although newer B news releases were available, they offered little beyond B 2.10, and it was often necessary to regression-test new B news releases to verify that reported, published bug fixes had in fact been applied.

Spencer acted first and rewrote *expire* from the ground up. Though it initially lacked any form of selective expiry, this *expire*, when run each night, finished in about 15 minutes. (This was on 750-class machines holding all Usenet news and expiring after 14 days.)

Collyer observed in November 1985 that B *rnews*, upon receiving a batch of news, immediately *exec*ed a trivial unbatcher which copied each article into a temporary file and then forked and *exec*ed B rnews again. Such a technique is clearly overkill for articles averaging about 3,000 bytes each. Preliminary experiments failed to produce a modified B rnews that could unravel a batch without forking. Consultation with Rick Adams, the current B-news maintainer, revealed that this same technique remained in the upcoming B news release (variously B 2.10.3 or B 2.11). Within one week‡, a from-scratch C *rnews* prototype was working well enough to run experimentally on a ‘leaf’ machine receiving a subset of news.

This prototype version lacked a good many necessary amenities, and over the next eight months it was enhanced to bring it up to full functionality. It was also tuned heavily to improve its performance, since it was faster than B *rnews* but still not fast enough to make us happy.

Once the *rnews* newsgroup name matching routines were working, Spencer revised *expire* to add selective expiry, specified in a control file. Recently, we have also revised our old batcher heavily, largely to add capability but with an eye on performance.

2. Rnews Performance

The basic objective of C news was simpler code and higher performance. This may sound trite, but note that performance was an explicit objective. That was important. *Programs will seldom run faster unless you care about making them run faster.*

---

* Recent *uucp*s (notably Honey DanBer) provide spool sub-directories, and recent 4BSD (4.3BSD and later) kernels provide linear (as opposed to quadratic) directory searching, both of which help this problem.

† Never mind the cost/benefit ratio.

‡ 40 hours, Collyer didn’t have to work hard.

-- --

‘Faster’ implies comparison to a slower version. Knowing the value of improvements, and assessing this in relation to their cost, requires knowing the performance of the unimproved version. Collyer kept detailed records of his work on *rnews*, so he could see how much progress he was making. See the Appendix for the final result. *To know how to get somewhere, you must know where you are starting from.*

The first functional C *rnews* ran at about 3 times the speed of B *rnews*. We had assumed that merely eliminating the fork/exec on each article would give a factor of 10 improvement, so this was disappointing. *Avoiding obvious performance disasters helps... but it’s not always enough.*

Profiling, first with *prof*(1) and later with 4.2BSD’s *gprof*(1), and rewriting of the bottlenecks thus discovered, eventually brought the speed up to over 19 times the speed of B *rnews*. This required a number of write-profile-study-rewrite cycles. There is undoubtedly still a lot of code which could be faster than it is, but since profiling shows that it doesn’t have a significant impact on overall performance, who cares? *To locate performance problems, look through the eyes of thy profiler.*

Collyer first experimented with using *read* and *write* system calls instead of *fread* and *fwrite*, and got a substantial saving. Though the usage of system calls in this experiment was unportable, the saving eventually lead him to rewrite *fread* and *fwrite* from scratch to reduce the per-byte overheads. This helped noticeably, since pre-System-V *fread* and *fwrite* are really quite inefficient. *If thy library function offends thee, pluck it out and fix it.*

At the time, C *rnews* was doing fairly fine-grain locking, essentially locking each file independently on each use. News doesn’t need the resulting potential concurrency, especially when *rnews* runs relatively quickly, and the locking was clearly a substantial fraction of the execution time. C *rnews* was changed to use B-news compatible locking, with a single lock for the news system as a whole. *Simplicity and speed often go together.*

When sending articles to a site using batching, *rnews* just appends the filename of each article to a *batch file* for that site. The batch file is later processed by the batcher. In principle, batching is an option, and different sites may get different sets of newsgroups. In practice, few articles are ever sent unbatched, and most articles go to all sites fed by a given system. This means that *rnews* is repeatedly appending lines to the same set of batch files. Noticing this, Collyer changed C *rnews* to keep these files open, rather than re-opening them for every article*. *Once you’ve got it, hang onto it.*

These two simple changes—coarser locking and retaining open files—cut system time by about 20% and real time by still more.

On return from Christmas holidays, after considerable agonizing over performance issues, Collyer turned some small, heavily-used character-handling functions into macros. This reduced user-mode time quite a bit. *A function call is an expensive way to perform a small, quick task.*

Rnews was always looking up files by full pathnames. Changing it to *chdir* to the right place and use relative names thereafter reduced system time substantially. *Absolute pathnames are convenient but not cheap.*

Studying the profiling data revealed that *rnews* was spending a lot of time re-re-re-reading the *sys* and *active* files. These files are needed for processing every article, and they are not large. Collyer modified *rnews* to simply read these files in once and keep them in core. This change alone cut system time and real time by roughly 30%. *Again, once you’ve got it, don’t throw it away!*

There is a more subtle point here, as well. When these files were re-read every time, they were generally processed a line at a time. The revised strategy was to *stat* the file to determine its size, *malloc* enough space for the whole file, and bring it in with a single *read*. This is a vastly more efficient way to read a file! *Tasks which can be done in one operation should be.*

* The price for this tactic is that the code has to be prepared for the possibility that the number of sites being fed exceeds the supply of file descriptors. Fortunately, that is rare.

At this point (mid-January 1986), C *rnews* was faster than B *rnews* by one order of magnitude, and there was much rejoicing.

In principle, the ‘Newsgroups:’ header line, determining what directories the article will be filed in, can be arbitrarily far from the start of the article. In practice, it is almost always found within the first thousand bytes or so. By complicating rnews substantially, it became possible in most cases to *creat* the file in the right place (or the first of the right places) in */usr/spool/news* before writing any of the article to disk, eliminating the need for temporary files or even temporary links. The improvement in system time was noticeable, and the improvement in user time was even more noticeable. *Prepare for the worst case, but optimize for the typical case.*

There are certain circumstances, notably control-message articles, in which it is necessary to re-read the article after filing it. *Rnews* originally re-opened the article to permit this. Changing the invocation of *fopen* to use the **w+** mode made it possible to just seek back to the beginning instead, which is *much* faster. This, plus some similar elimination of other redundant calls to *open*, reduced system time by over 30%. *Get as much mileage as possible out of each call to the kernel.*

Both scanning the in-core *active* and *sys* files and re-writing the *active* file are simpler if the in-core copies are kept exactly as on disk, but this implied frequent scans to locate the ends of lines. It turned out to be worthwhile to pre-scan the *active* file for line boundaries, and remember them. *When storing files in an unstructured way, a little remembered information about their structure goes a long way in speeding up access.*

We already had a *STREQ* macro, just a simple invocation of *strcmp*, as a convenience. As a result of some other experience by Spencer, Collyer tried replacing some calls of *strncmp* by a *STREQN* macro, which compared the first character of the two strings in-line before incurring the overhead of calling *strncmp*. This sped things up noticeably, and later got propagated through more and more of the code. String-equality tests usually fail on the very first character. *Test the water before taking the plunge.*

While looking at string functions, Collyer noticed that *strncmp*s to determine whether a line was a particular header line had the comparison length computed by applying *strlen* to the prototype header. With a little bit of work, the prototypes were isolated as individual character arrays initialized at compile time. This permitted substituting the compile-time *sizeof* operation for the run-time *strlen*. *Let the compiler do the work when possible.*

At this point, profiling was turned off temporarily for speed tests. Profiling does impose some overhead. The speed trials showed that C *rnews* was now running at over 15 times the speed of B *rnews*.

After months of adding frills, bunting and B 2.11 compatibility*, Collyer again returned to performance tuning in August 1986. The 4.2BSD kernel on **utcs** now included the 4.3BSD *namei* caches, which improve filename-lookup performance considerably. Unfortunately, considerations of crash recovery dictated some loss in performance: it seemed desirable to put batch-file additions out by the line rather than by the block. *Performance is not everything.*

Gprof revealed that newsgroup name matching was an unexpected bottleneck, so that module was extensively tweaked by adding **register** declarations, turning functions into macros, applying *STREQN* and such more widely, and generally tuning the details of string operations. The code that handled *sys*-file lines got similar treatment next. The combination cut 40% off user-mode time. *Persistent tuning of key modules can yield large benefits.*

Newsgroup matching remained moderately costly, and an investigation of where it was being used revealed two separate tests for a particular special form of name. It proved awkward to combine the two, so the testing routine was changed to remember having done that particular test already. *If the same question is asked repeatedly, memorize the answer.*

* And supposed B 2.11 compatibility, as those who remember the short-lived cross-posting restrictions will recall.

By this time, the number of system calls needed to process a single article could be counted on one's fingers, and their individual contributions could be assessed. At one point it was desirable for a *creat* to fail if the file already existed, so this was being checked with a call to *access* first. John Gilmore pointed out that on systems with a 3-argument *open* (4.2BSD, System V), this test can be folded into the *open*. The elimination of the extra name→file (*namei*) mapping cut both system time and real time by another 15%. (Note that this system *does* have *namei* cacheing!) *File name lookups are expensive; minimize them.*

The development system (**utcs**, a 750) is now filing 2-3 articles per second on average; **utfraser** (a Sun 3/160 with an Eagle disk) is typically filing 6-7 articles per second. C *rnews* runs over 19 times as fast in real time as B *rnews*, over 25 times as fast in system-mode CPU time, roughly 3.6 times as fast in user-mode CPU time, and over 10 times as fast in combined CPU times.

With one exception (see *Future Directions*), it now appears that very little can be done to speed up *rnews* without changing the specifications. It seems to be executing nearly the bare minimum of system calls, and the user-level code has been hand-optimised fairly heavily.

3. Expire Performance

The rewrite of *expire* that started this whole effort was only partly motivated by performance problems. Performance was definitely bad enough to require attention, but the B *expire* of the time also had some serious bugs. Worse, the code was a terrible mess and was almost impossible to understand, never mind fix. Early efforts were directed mainly at producing a version that would *work*; rewriting *expire* from scratch simply looked like the easiest route. Decisions made along the way, largely for other reasons, nevertheless produced major speedups.

The first of these decisions was a reduction in the scope of the program. B *expire* had several options for doing quite unrelated tasks, such as rebuilding news's history file. The code for these functions was substantial and was somewhat interwoven with the rest. C *expire* adheres closely to a central tenet of the 'Unix Philosophy': *a program should do one task, and do it well.* This may appear unrelated to performance, but better-focussed programs are generally simpler and smaller, reducing their resource consumption and making performance tuning easier (and hence more likely). In addition, a multipurpose program almost always pays some performance penalty for its generality.

The second significant decision had the biggest effect on performance, despite being made for totally unrelated reasons. For each news article, the B news history file contained the arrival date and an indication of what newsgroups it was in. This is *almost* all the information that *expire* needs to decide whether to expire an article or not. The missing* data is whether the article contains an explicit expiry date, and if so, what it is. B *expire* had to discover this for itself, which required opening the article and parsing its headers. A site which retains news for two weeks will have upwards of 5,000 articles on file. A few dozen of them will have explicit expiry dates. *But B expire opened and scanned all 5,000+ articles every time it ran!* This was a performance disaster.

We actually did not want to parse headers in *expire* at all, because the B news header-parsing code was (and is) complex and was known to contain major bugs. The performance implications of this were obvious, although secondary at the time. Header parsing is itself a non-trivial task, and accessing 5,000+ files simply cannot be made cheap. *Information needed centrally should be kept centrally.*

The C news history file has the same format as that of B news, with one addition: a field recording the explicit expiry date, if any, of each article. If no expiry date is present in the article, the field contains '−' as a placemarker†. In this way, the header parsing is done *once* per article, on arrival. In fact, the extra

* Recent versions of B news have made some attempt to redress this lack, but haven't gone as far as C expire. The discussion here applies to the B expire that was current at the time C expire was written.

† It would be possible to simply compute a definitive expiry date for an article when it arrives, and record that. This would eliminate the decision-making overhead in *expire*, but would greatly slow the response to changes in expiry policy. Since one reason to change policy is time-critical problems like a shortage of disk space, this loss of flexibility was judged unacceptable. It is better to leave the expiry decision to *expire* and concentrate on making *expire* do it quickly.

effort involved is essentially nil, since *rnews* does full header parsing at arrival time anyway. *Rnews* had to be changed to write out the expiry date, and code which knew the format of the history file had to be changed to know about the extra field. Perhaps a dozen lines of code outside *expire* were involved.

A crude first version of C *expire*, incorporating these decisions in the most minimal way, ran an order of magnitude faster than B *expire*. Precise timing comparisons were not practical at the time, since the original motive for C *expire* was that B *expire* had stopped working completely, crippled by bugs in its header parsing. Later versions of B *expire* did cure this problem, but we were no longer interested in putting up slow, buggy software just to make an accurate comparison.

Further work on C *expire* mostly concentrated on cleaning up the hasty first version, and on incorporating desired features such as selective expiry by newsgroup. Selective expiry caused a small loss in performance by requiring *expire* to check the newsgroup(s) of each article against an expiry-control list. Here, *expire* benefitted from the work done to speed up the newsgroup-matching primitives of *rnews*, since *expire* uses the same routines. *If you re-invent the square wheel, you will not benefit when somebody else rounds off the corners*[‡].

One improvement that was made late in development was in the format of the dates stored in the history file. B *rnews* stored the arrival date in human-readable form, and *expire* converted this into numeric form for comparisons of dates. Date conversion is a complex operation, and the widely-distributed *getdate* function used by news is not fast. Inspection of the code established that *expire* was the only program that ever looked at the dates in the history file. There is some potential use of the information for debugging, but this is infrequent, and a small program that converts decimal numeric dates to human-readable ones addresses the issue. Both C *rnews* and C *expire* now store the dates in decimal numeric form. *Store repeatedly-used information in a form that avoids expensive conversions.*

Actually, C *expire* bows to compatibility by accepting either form on input, but outputs only the decimal form as it regenerates the history file. Thus, in the worst case, *expire* does the conversion only once for each history line, rather than once per line per run. *"If they hand you a lemon, make lemonade".*

If *expire* is archiving expired articles, it may need to create directories to hold them. This is an inherently expensive operation, but it is infrequently needed. However, checking to see whether it *is* in fact needed is also somewhat expensive... and the answer is almost always 'no'. The same is true of checking to see whether the original article really still exists: it almost always does. (This cannot be subsumed under generic 'archiving failed' error handling because a missing original is just an article that was cancelled, and does not call for a trouble report.) Accordingly, C *expire* just charges ahead and attempts to do the copying. Only if this fails does *expire* analyze the situation in detail. *Carrying a net in front of you in case you trip is usually wasted effort.*

Archiving expired articles often requires copying across filesystem boundaries, since it's not uncommon to give current news and archived news rather different treatment for space allocation and backups. Copying from one filesystem to another can involve major disk head movement if the two filesystems are on the same spindle. Since head movement is expensive, maximizing performance requires getting as much use as possible out of each movement[*]. *Expire* is not a large program, and even on a small machine it can spare the space for a large copying buffer. So it does its archiving copy operations using an 8KB buffer. *Buying in bulk is often cheaper.* Since 8KB accommodates most news articles in one gulp, there is little point in enlarging it further. *The law of diminishing returns does apply to buying in bulk.*

Since *expire* is operating on the history file at (potentially) the same time that *rnews* is adding more articles to it, some form of locking is necessary. Given that *expire* has to look over the whole database of news, and typically has to expire a modest fraction of the articles, it is a relatively long-running process compared to *rnews*. Contention for the history-file lock can be minimized by noting that *rnews* never does

‡ A corollary of this is: *know thy libraries, and use them*.

* As witness the progressive increase in filesystem block size that produced major performance improvements in successive versions of 4BSD.

anything other than append to the file. So *expire* can leave the file unlocked while scanning it; the contents will not change. When (and only when) *expire* reaches end-of-file, it locks the news system, checks for and handles any further entries arriving on the end of the history file meanwhile, and finishes up. *Locking data that won't change is wasteful.*

After careful application of these various improvements, C *expire* is fast enough that further speedup is not worth much effort. However, an analysis of where it spends its time does suggest one area that might merit attention in the future. *Expire* rebuilds the history file to reflect the removal of expired articles. The history file is large. *Expire* must also rebuild the *dbm* indexing data base, since it contains offsets into the history file. This data base is comparable in size to the history file itself, and is generated in a less orderly manner that requires more disk accesses.

Much of the time needed for these operations could be eliminated if *expire* could mark a history line as 'expired' without changing its size. This could be done by writing into the history file rather than by rebuilding the whole file, and the indexing database would not need alteration. This would also permit retaining information about an article after the article itself expires, which would simplify rejecting articles that arrive again (due to loops in the network, etc.) after the original has expired. The history file should still be cleaned out, and the indexing database rebuilt, occasionally. C *expire* contains some preliminary 'hooks' for this approach, but to date full implementation does not seem justified: C *expire* is already fast enough. *Know when you are finished.*

4. Batcher Performance

The C batcher is descended from a very old version written to add some minor functionality that was not present in the B batcher of the time. It is small and straightforward, and contains only a couple of noteworthy performance hacks.

The batcher works from a list of filed articles, to be composed into batches. The list is by absolute pathname. All of these files reside in the same area of the system's directory tree, and referring to them with absolute pathnames every time implies repeatedly traversing the same initial pathname prefix. To avoid this, the batcher initially *chdir*s to a likely-looking place such as */usr/spool/news*. Thereafter, before using an absolute pathname to open an article, it checks whether the beginning of the pathname is identical to the directory where it already resides. If so, it strips this prefix off the name before proceeding. *If you walk the same road repeatedly, consider moving to the other end.*

The batcher's input is usually in fairly random order, with little tendency for successive files to be in the same directory. If this were not the case, it would be worthwhile for the batcher to actually move around in the directory tree to be closer to the next file.

The batcher used to copy data using *putc(getc())* loops. This has been replaced by *fread/fwrite* which is significantly faster, especially if using the souped-up *fread/fwrite* mentioned earlier. *If you need to move a mountain, use a bulldozer, not a teaspoon.*

5. Future Directions

The one improvement we are still considering for *rnews* is a radical revision of the newsgroup-matching strategy. Newsgroup matching still consumes about 18% of user-mode processor time. The key observation is that the information that determines which newsgroups go to which sites seldom changes. It would probably be worth precompiling a bit array indexed by newsgroup and site, and recompiling it only when the *active* file or the *sys* file changes in a relevant way. This would cut the newsgroup-matching time to essentially zero.

Rnews would be faster (and simpler) if 'Newsgroups:' and 'Control:' were required to be the first two headers (if present) of each article. At present *rnews* tries to find them before starting to write the article out, so that it can put the article in the right place from the start, but it has to allow for the possibility that vast volumes of other headers may precede them.

Hashing *active*-file lookups in *rnews* would be fun, but profiling suggests that it's not worthwhile unless the number of newsgroups is in the thousands.

When PDP-11's are truly dead on Usenet, the use of large per-process memories *may* allow further speedups to *rnews* by reading the entire batch into memory at once and writing each article to disk in a few *writes* (it can't easily be reduced to a single *write* because headers must be modified before filing).

One optimization we have *not* considered is re-coding key parts in assembler. C news already runs on five different types of machine. Use of assembler would be a maintenance nightmare, and probably would not yield benefits comparable to those of the more high-level changes.

6. Acknowledgements

Ian Darwin ran the very earliest alpha versions of *rnews* and gave helpful feedback. Mike Ghesquiere, Dennis Ferguson and others have run later versions and prodded Collyer to fix or implement assorted things. John Gilmore and Laura Creighton read and criticized an early alpha version of *rnews*.

7. Appendix: rnews Times

Measurements have been taken on a VAX 750 running 4.2BSD under generally light load, using a batch of 297,054 bytes of net.unix-wizards containing 171 articles and ˜104 cross-postings. All times are in seconds per article.

time	real	user	sys	comments
85 Dec 6 00:54	4.68	0.3	1.29	**B news** rejecting all. (b.1.rej)
85 Dec 6 00:54	3.184	0.69	0.67	first timing trial; *profiling on* (c.1)
85 Dec 6 00:54	0.66	0.175	0.199	rejecting all (c.2.rej)
85 Dec 6 03:25	0.58	0.175	0.175	still rejecting all (c.3.rej)
85 Dec 6 23:46	9.058	0.631	2.251	**B news** using private directories, rejected 53 of the 171 articles as "too old" (b.2)
85 Dec 7 00:24	2.0 (est)	-	-	on a 10 MHz 68000 with slow memory and slow disk (crude timings) (c.darwin.1)
85 Dec 7 00:40	7.576	0.684	2.403	**B news** without the "too old" reject code and having cleared out history (b.3)
85 Dec 7 04:43	1.99	0.49	0.53	accepting the articles, using read and write for bulk copies (c.4)
85 Dec 7 06:10	2.261 (!)	0.497	0.449	optimized by less locking & keeping batch files open (c.5)
85 Dec 7 07:32	1.383	0.491	0.414	same as the last one, but with a lower load average (around 1.5) (c.6)
85 Dec 16 03:43	1.380	0.447	0.374	for calibration after misc. cleanup (c.7, c.8)
86 Jan 13 00:23	1.232	0.349	0.301	turned hostchar() into a macro (c.9)
86 Jan 13 04:26	1.36	0.333	0.242 (!)	using in-core active file, under heavy load (c.10)
86 Jan 13 08:24	1.94	0.349	0.253	using in-core sys file too, under heavy load. Re-run this trial! (c.11)
86 Jan 13 08:42	0.892 (!)	0.332	0.245	re-run at better nice. Not striking, except for real time. Was run in a large directory; ignore. (c.12)
86 Jan 13 08:59	0.861 (!)	0.333	0.212 (!)	re-run at good nice & in a small directory. Have beaten B news by *one order of magnitude* on real & sys times! Beat it by more than twice on user time. (c.13)
86 Jan 21 19:15	1.208	0.349	0.245	creat 1st link under final name, only link to make cross-postings; with HDRMEMSIZ too small (c.14)
86 Jan 21 19:57	0.728	0.318	0.193	previous mod, with HDRMEMSIZ of 4096 (c.15)

86 Jan 22 01:20	0.719	0.315	0.166	fewer opens (just rewind the spool file), but Xref(s): not working (c.16)
86 Jan 22 01:53	0.637 (!)	0.314	0.154 (!)	fewer opens fixed to spell Xref: right; Xref: not working (c.17)
86 Jan 22 04:00	0.874	0.325	0.174	fewer opens with Xref: fixed (times may be high due to calendar) (c.18)
86 Jan 22 05:45	0.694	0.309	0.159	under lighter load, times are better (c.19)
86 Jan 24 04:29	0.715	0.317	0.129 (!)	turn creat & open into just creat, under slightly heavy load (c.20)
86 Jan 24 06:06	0.628 (!)	0.288 (!)	0.129 (!)	reduce number of calls on index (by noting line starts at the start) and strncmp (via macro) in active.mem.c, but still profiling and writing stdout and stderr to the tty (c.21)
86 Jan 24 07:22	0.653	0.209 (!)	0.123 (!)	fewer strlen calls (by using **sizeof** s - 1), writing stdout to /dev/null and with *profiling off*, but under moderate load; try again (c.23)
86 Jan 24 07:35	0.574 (!)	0.216 (!)	0.123 (!)	as last time, but stdout to tty(!) and under light load. *running 15.67 times as fast as B rnews* (c.24)
86 Aug 8 04:23	0.839	0.51	0.124	performance hit: fflush after each history line for crash-resilience; run for *gprof* output and calibration with later runs. running under 4.2.1BSD (has 4.3 namei cache) now. real and user times are way up; due to gprof profiling? (c.25)
86 Aug 8 04:24	0.962	0.438 (!)	0.131	run with faster ngmatch, with **register** decl.s and wordmatch and STREQN macros; saved 15% user. User time is better than c.25, but still up from c.24. (c.26)
86 Aug 10 07:35	0.805	0.345 (!)	0.135	further speedups: ngmatch has more **register** decl.s and in-line index; more use of STREQ(N) macro for str(n)cmp in hdrmatch, ngmatch.c and transmit.c; faster ishdr without index. real & user times are better than both c.26 and c.25 (c.27)
86 Aug 11 04:19	1.012	0.303 (!)	0.146	rewrote sys.c, used INDEX and STREQ(N) macros throughout rnews. real and sys times are up, but user continues to decline. (c.28)
86 Aug 12 03:51	1.315	0.315	0.154	minor tweaks: all.all.ctl caching, etc. (c.29)
86 Aug 30 17:56	0.564	0.189 (!)	0.112 (!)	light load, thought we had 3-arg open in fileart, but didn't. Odd. *Stopped using gprof.* (c.30)
86 Aug 30 17:57	0.475 (!)	0.191	0.095 (!)	Really and truly use the 3-arg open. *19 times B rnews speed.* (c.31)