SLURP-FILE Performance Comparison

gene m. stover

created Wednesday, 2004 February 18
updated Wednesday, 2005 June 15

Contents

1 Introduction

I recently needed a function in Common Lisp that returned the entire contents of a text file as a string. I call the function SLURP-FILE.

I could imagine many ways to implement a SLURP-FILE function. Which implementation would be faster? I'm too experienced to trust my guts when it comes to optimization, so I implemented many versions & ran some performance tests.

The performance of a SLURP-FILE function isn't very important, but I know that I'll forget the results before long, & I also know the issue will arise again in the future, so I figure it's best to write it all now so I don't need to repeat the work later.

2 slurp-file & slurp-stream

I implemented SLURP-FILE on top of a function called SLURP-STREAM. I actually implemented many versions of SLURP-STREAM, not SLURP-FILE.

All of the source code is in Appendix A.

3 The different versions

Here are short descriptions of each version of each of the SLURP-STREAM... functions work.

slurp-stream-cons Push characters onto a list, then reverse the list & coerce it to a string.

slurp-stream-vector-push Push characters onto an adjustable vector with a default increment size.

slurp-stream-vector-push2 Push characters onto an adjustable vector with an increment size of 1024.

slurp-stream-line Concatenate one line at a time into a single string.

slurp-stream-string-stream Write one line at a time to a string stream.[1]

slurp-stream-string-stream2 Write one character at a time to a string stream.

4 Performance Comparison

I ran them on SBCL 0.8.8 on some kind of Linux on my computer called Plague which has a 750 MHz Pentium & 128 megaoctets of memory, I guess.

Table 1 shows the performance results from evaluating (testall "slurp-file.lisp"). The "slurp-file.lisp" file is small, about 4 kilooctets.

Table 2 shows the performance results from evaluating `(testall "big.tmp")` which is a file of 819,999 characters created by evaluating (MAKE-BIG-FILE).

For the "big.tmp" test, I wanted to use a larger file, but as you can see, SLURP-STREAM-LINE was so slow that it would have taken too much time (or too much patience).

The rows are exactly the output from TESTALL (see the source code in Appendix A) except that I re-ordered them from fastest to slowest.

[1] Thanks for Shawn Betts for writing this function & telling me about it. It's new to this version of this essay.

function	characters	seconds	rate
SLURP-STREAM-STRING-STREAM	26103789	10	2.61e+6
SLURP-STREAM-STRING-STREAM2	21732732	10	2.17e+6
SLURP-STREAM-CONS	15532458	10	1.55e+6
SLURP-STREAM-VECTOR-PUSH	6359139	10	6.36e+5
SLURP-STREAM-VECTOR-PUSH2	5142687	10	5.14e+5
SLURP-STREAM-LINE	154326	10	1.54e+4

Table 1: The performance results from (TESTALL "SLURP-FILE.LISP")

function	characters	seconds	rate
SLURP-STREAM-STRING-STREAM	31980000	10	3.20e+6
SLURP-STREAM-STRING-STREAM2	24599970	10	2.46e+6
SLURP-STREAM-CONS	11479986	10	1.15e+6
SLURP-STREAM-VECTOR-PUSH	7379991	10	7.38e+5
SLURP-STREAM-VECTOR-PUSH2	819999	184	4.46e+3
SLURP-STREAM-LINE	820000	289	2.84e+3

Table 2: The performance results from (TESTALL "BIG.TMP")

Clearly, the fastest of the functions is SLURP-STREAM-STRING-STREAM, but it has a flaw, so before you grab it to use as your SLURP-STREAM function, keep reading.

5 Flaws

Two of the functions, SLURP-STREAM-STRING-STREAM & SLURP-STREAM-LINE, use Common Lisp's READ-LINE function to gobble-up a line at a time. READ-LINE discards the end-of-line character(s), so when appending the newly consumed line to the collected characters, those two SLURP-STREAM... functions assume that the line ended with end-of-line characters.

This is fine most of the time, but if the last line of the file ends with the end-of-file, not with end-of-line characters, SLURP-STREAM-STRING-STREAM & SLURP-STREAM-LINE will not return the precise contents of the file.

For this reason, I think one of the character-at-a-time functions is a better choice.

6 Analysis

The fastest of the SLURP-STREAM... functions is SLURP-STREAM-STRING-STREAM, but it has a bug[2], so I recommend ignoring it.

[2]The bug is described in Section 5.

The next fastest is SLURP-STREAM-STRING-STREAM2, which does not share the bug. That's the one to use.

SLURP-STREAM-CONS is the third fastest. It pushes individual characters onto a list, then reverses the list before returning it. That idiom, which I call "Accumulate a list" in [Sto04], is usually a decent choice. Here in SLURP-STREAM... land, we have evidence which supports that heuristic. Sure, SLURP-STREAM-CONS isn't the fastest algorithm we look at here, but it's better than one third the speed of the fastest, & it's almost 1,000 times faster than the slowest.

I'm surprised that the two VECTOR-PUSH-EXTEND algorithms are so slow. I would have guessed that string streams were implemented with VECTOR-PUSH-EXTEND, so I expected that those two functions would have about the same performance as the two string stream functions. Not so.

Finally, there's SLURP-STREAM-LINE. When I wrote the first version of this article, I thought that function would be the fastest, but it's the slowest. It has barely one thousandth the performance of the fastest two functions. That's probably from all the copying it does. Also notice that it's the longest, most complicated, of the SLURP-FILE... functions. This is yet one more datum (for me, anyway) in support of my own insistence that you should always use the least complicated algorithm that is correct because when you think you need the complexity for performance, you're dead wrong.

7 Conclusion

SLURP-STREAM-STRING-STREAM2 is the best choice.

8 Addendum 2005 April 06

If you make NEXT-CHAR inlined, with "`(declaim (inline next-char))`", you might improve the performance of SLURP-STREAM-STRING-STREAM2 by about 9 percent.[3]

9 Addendum 2005 June 15

Thanks to all the people[4] who e-mailed me about http://www.emmett.ca/~sabetts/slurp.html[5]. Mr. Insane 3000's function is definitely faster & more elegant than any in my performance test.

Actually, I think that function should be called SLURP-FILE because it only operates on files with pathnames. (It can operate on a stream object, but only

[3] Thanks to Eric Moss for pointing this out.

[4] There were at least a half-dozen of you, but the only name I remember is Henrik Hjelte, who sent the message that pushed me over the edge to update this essay.

[5] a SLURP-STREAM4 function by Mr. Insane 3000

if that stream object is opened to a file with a pathname.) With that in mind, his could be re-written as

```
(defun slurp-file-3000 (pathname)
  "A SLURP-FILE function inspired Mr. Insane 3000's
SLURP-STREAM4."
  (with-open-file (strm pathname)
    (let ((string (make-string (file-length strm))))
      (read-sequence string strm)
      string)))
```

Nice. And fast. Thanks, Mr. Insane 3000.

Something to think about: What is the behaviour of `(file-length *standard-input*)`?

"Okay, I'm thinking about it, but when would this matter?" Answer: In CGI programs & un*x-style filters.

A Source Code

Here is the source code from `slurp-file.lisp`. It is also available at `http://lisp-p.org/sfpc/slurp-fil`

```
;;; -*- Mode: Lisp -*-
;;;
;;; $Header: /home/gene/library/website/docsrc/sfpc/RCS/sfpc.tex,v 395.1 2008/04/20 17:25:50 gene
;;;
;;; Copyright (C) 2004, 2005  Gene Michael Stover.  All rights reserved.
;;;
;;; This library is free software; you can redistribute it
;;; and/or modify it under the terms of version 2.1 of the GNU
;;; Lesser General Public License as published by the Free
;;; Software Foundation.
;;;
;;; This library is distributed in the hope that it will be
;;; useful, but WITHOUT ANY WARRANTY; without even the implied
;;; warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR
;;; PURPOSE.  See the GNU Lesser General Public License for more
;;; details.
;;;
;;; You should have received a copy of the GNU Lesser General
;;; Public License along with this library; if not, write to the
;;; Free Software Foundation, Inc., 59 Temple Place, Suite 330,
;;; Boston, MA 02111-1307 USA
;;;

(defun next-char (strm)
  (read-char strm nil strm))

(defconstant newline-string (format nil "~%"))
```

```
(defun xread-line (strm)
  "Returns a string containing the next line in the file
with the end-of-line character(s).  On end-of-file,
returns STRM."
  (read-line strm nil strm))

(defun slurp-stream-cons (strm)
  (do ((x (next-char strm) (next-char strm))
       (lst () (cons x lst)))
      ((eq x strm) (coerce (nreverse lst) 'string))))

(defun slurp-stream-vector-push (strm)
  "Return a string containing the entire
contents of the stream."
  (let ((str (make-array 1024
                         :element-type 'character
                         :adjustable t
                         :fill-pointer 0)))
    (do ((ch (next-char strm) (next-char strm)))
        ((eq ch strm) str)
        (vector-push-extend ch str))))

(defun slurp-stream-vector-push2 (strm)
  "Return a string containing the entire
contents of the stream."
  (let ((str (make-array 1024
                         :element-type 'character
                         :adjustable t
                         :fill-pointer 0)))
    (do ((ch (next-char strm) (next-char strm)))
        ((eq ch strm) str)
        (vector-push-extend ch str 1024))))

(defun slurp-stream-line (strm)
  "Slurp the contents of the stream.  Return them in a
string.  This turned out to be slower than the 'simple'
version."
  (do ((str "" (concatenate 'string
    (concatenate 'string str line)
    newline-string))
       (line (xread-line strm) (xread-line strm)))
      ((eq line strm) str)))

;;; Suggested by Sean Belts.
(defun slurp-stream-string-stream (stream)
  "Return the contents of file as a string."
  (with-output-to-string (out)
    (do ((line (xread-line stream) (xread-line stream)))
        ((eq line stream))
      (write-line line out))))
```

```
(defun slurp-stream-string-stream2 (stream)
  "Return the contents of file as a string."
  (with-output-to-string (out)
    (do ((x (next-char stream) (next-char stream)))
        ((eq x stream))
      (write-char x out))))

(defun slurp-file (pathname fn)
  (declare (type function fn))
  (with-open-file (strm pathname)
    (funcall fn strm)))

(defun timetest (slurper pathname)
  (declare (type symbol slurper) (type (or pathname string) pathname))
  (format t "~&~A" slurper)
  (force-output)
  (let* ((start-time (get-universal-time))
         (stop-time (get-universal-time))
 (fn (symbol-function slurper))
         (count 0))
    (declare (type function fn))
    ;; Side-effect of this loop is to bind values to
    ;; COUNT & STOP-TIME.
    (do ()
        ((>= (- stop-time start-time) 10))
(incf count (length (slurp-file pathname fn)))
        (setq stop-time (get-universal-time)))
    (format t " & ~A & ~A & ~,2E \\\\ \\hline" count
            (- stop-time start-time)
            (/ count (- stop-time start-time)))
    (force-output))
  slurper)

(defun testall (pathname)
  (declare (type (or pathname string) pathname))
  (mapc #'(lambda (symbol)
    (timetest symbol pathname))
(list 'slurp-stream-cons
      'slurp-stream-vector-push
      'slurp-stream-vector-push2
      'slurp-stream-line
      'slurp-stream-string-stream
      'slurp-stream-string-stream2))
  'testall)

(defvar *chars* "0123456789abcdefghijklmnopqrstuvwxyz")
(defun random-char ()
  (char *chars* (random (length *chars*))))
```

```
(defun make-big-file ()
  (with-open-file (strm "big.tmp" :direction :output
:if-exists :rename-and-delete
:if-does-not-exist :create)
    (dotimes (line 800)
      (format strm "~&")
      (dotimes (char 1024)
(format strm "~C" (random-char)))))
  'make-big-file)

;;; --- end of file ---
```

References

[Sto04] Gene Michael Stover. Lisp idioms. *cybertiggyr.com*, Jan 2004. `http://cybertiggyr.com/lid/lid.pdf`.