How Shall We Connect Our Software Tools?

Eric Grosse
AT&T Bell Laboratories
Murray Hill NJ 07974 USA.
ehg@research.att.com.

Abstract

Traditionally we connect our software tools using human-readable files. This is a conscious decision to buy flexibility and understandability at some cost in performance relative to binary file formats. This note explores the possibility of using shared memory functions to retain most of the existing style while leapfrogging the speed of reading binary files, at least in some environments and for some applications.

1 The performance issue

Imagine you are a scientist running a three-dimensional differential equation simulation and needing to look at the solution at various time steps. Either you have a huge monolithic program to build, or else you have a huge amount of data to transfer between two programs. The latter alternative is generally regarded as superior, and there will usually be so much computation going on in both phases that the cost of even a slow Fortran formatted input/output (i/o) library is not prohibitive. But as the overall task is broken into a pipeline of more and more processes, either for software engineering or parallel processing reasons, i/o costs can become significant.

Experience building and using a collection of visualization tools[2, 3] which tend to do a modest amount of arithmetic on large arrays has been the immediate cause for examining this topic, though it is clearly a general software issue. Profiling reveals that as much as half the cpu time goes to i/o. Converting to binary i/o is undesirable, because past experience has taught that binary formats tend to become indecipherable after a few years and represent an immediate hurdle when trying to collaborate with others. Is there another way?

Consider the problem of a writer process which has generated a large array and several reader processes that each need to access some parts of the array. The most common case is that there is a single reader that needs the entire array, but the more general case is also important. For example, a graphics program may only need coefficients on the external faces of a domain, but as the user interactively slices the domain, the graphics needs rapid access to new parts of the array. Or imagine a kanji translation program that needs fast, scattered access to a dictionary.

2 Shared memory as a candidate solution

Many UNIX systems provide the ability to create segments of shared memory that may be passed between processes. These segments act somewhat like temporary files, in that they continue to exist even after the process that created them dies.

The particular functions that do this (called `shmget` and `shmat`) have calling sequences specified by the System V Interface Definition (SVID)[1]. Hence programs calling `shmget` are portable to many machines (e.g. SGI, Stardent, Sun) of interest to the visualization community. Some machine architectures, however, require a different approach. On a Cray, it might be appropriate to use the Solid State Disk; on distributed memory machines, explicit message-passing may be needed. The following three C functions are therefore proposed, intended to capture the few features of shared memory that we need while allowing efficient alternative implementations.

The writer process calls

```
char* shm_create(char*name,0666,int nbytes);
```

here `name` is a descriptive string; `0666` can be replaced by an arbitrary protection mode as for files; `nbytes` is the amount of space to be written. The return value, `shm`, is a pointer to an area of `128+nbytes`, of which the first 128 contains a string composed of `name` and implementation-dependent information to yield a unique label L for the shared memory segment. After writing what it wishes in the remaining `nbytes`, the writer process calls

```
int shm_close(char* shm)
```

or may depend on the default closing of all segments upon process exit. The reader process gets L from the writer by some other channel, then calls

```
char* shm_open(char* L)
```

to get a `shm` pointer like before. When done the reader calls `shm_close` or exits. The design is modeled after that for handling UNIX files. Both `shm_create` and `shm_open` return a null pointer if they encounter an error; `shm_close` returns a non-zero on error. The library call `perror` may provide additional diagnostic information.

If our simulation and graphics programs ordinarily communicate by passing a file of the form

```
npts 10
coeff 0 1 2 3 4 5
6 7 8 9
```

then we may adopt the convention that the coefficient array may be replaced by a pointer to a shared memory segment. The file

```
npts 10
coeff shmid 402 simdat-3/23-seriesb
```

continues to pass through pipelines and be saved to disk just like the old one; the only change is that the library routine for reading `coeff` must now check if the first coefficient is `shmid` and if so replace `malloc` and `read` calls by `shm_open`. This does mean the simulation program will need a flag to specify when to use shared memory, since we certainly want to retain the default of simple ASCII, i.e. printable, output files. But the complication is restricted to a small part of the program and requires little extra effort on the part of the user.

Filters should always be provided to expand the `shmid` fields, for the benefit of programs (and people!) that do not care to deal with specialized formats. The philosophy is that *the master copy is ASCII*, which is the only format that should be archived. Since shared memory segments are so transitory, internal data structures may safely be changed without risk of making old files unreadable.

There is one mundane administrative issue with direct use of `shmget`, which returns simple integer labels. If, say, a dozen segments are created in the course of a simulation run and several simulations are active, it may not be easy to keep track of all these integers and clean up properly. Hence the convention of starting each shared memory segment with 128 bytes containing the integer as part of a descriptive character string. A tiny program that prints the string, user id, creation and access times (analogous to `ls`) and the system-provided command `ipcrm -m` *label* (analogous to `rm`) can together reduce the administrative burden to something like cleaning up the `/tmp` disk directory.

General use of shared memory would require careful synchronization to avoid timing bugs and deadlock. For the limited objective of accelerating i/o in pipelines, no such issues arise.

3 Benchmarking

Straight binary i/o (using the read and write UNIX system calls) is a plausible alternative to shared memory. To investigate the relative performance, a simple experiment was performed involving two processes (a) setting and writing and (b) reading and summing $200,000$ `doubles`. The size of the transfer is typical of current scientific computation, though use of only one floating point operation per array element (to emphasize i/o cost) is not! To bias the case in favor of binary i/o, this test only considers the case of reading a large contiguous array, not random access to a subset.

Five mechanisms for passing the array were tried. First, `shmget` was used to create memory for the array; the writer process (a) exited before the reader (b) began. The shared memory primitive `mmap`, which is provided by some operating systems instead of or in addition to `shmget` and which involves explicit mapping of memory to disk files, was also timed. In the third and fourth cases, space for the array was created by calling `malloc` in both processes and then using `write` and (multiple) `reads`. For comparison, an ASCII pipeline was also timed, as was a program that merely called `malloc` and set the array, but did not write it out. Wall clock times given in Table 1 are for the best of five runs; there was not much variation over the repetitions.

As expected, binary is much faster than ASCII. The fact that intermediate files and pipelines run at essentially the same speed shows that the operating systems' disk buffer cache in main memory is doing its job. In this particular implementation, anyway, `shmget` beats `mmap`; since the operating system is making less promise of permanence, this is not unreasonable.

One conclusion stands out most clearly. The speed of the reader under `shmget`, which would be the interactive graphics process under the scenario in §1, is

Table 1: Elapsed time (in seconds) for transferring $200,000$ `doubles`.

	`shmget`	`mmap`	`a|b`	`a>x;b<x`	`ascii`	`malloc`
(a) writer	0.20	0.43	0.73	0.43	10.5	0.20
(b) reader	0.07	0.12	0.80	0.36	10.7	
total	.3	.7	.8	.9	10.7	

stunning. Reading the data this way is a factor of three faster than just allocating space in the normal fashion!

The tests were run on an SGI 4D/240 with 128MB of main memory, running IRIX 4D1-3.3.2; process wall clock times were measured by `gettimeofday`. Changing between SMD and SCSI disks for the temporary file made no discernible difference; this was no surprise since the system had plenty of main memory for its disk buffer cache. Whether the programs were loaded with `-lmalloc` or not also made no difference. Similar times for binary i/o were observed on a MIPS M2000, but on our Vax 8550 running Version 10 Research Unix, the individual times for the pipeline are 3 to 4 times faster than using intermediate files. On a i386-based machine running SystemV, the total times for $20,000$ doubles were 8.0, NA, 8.3, 11.6, 19.8, 2.9.

4 Practical trial

Encouraged by the results of the previous section that meaningful speedups are possible on at least some important classes of machines, the approach was tested on a real application, the tensor-spline toolbox[2, 3]. Before the current project was conceived, 35 commands had already been implemented and editing the source for each one of them would have been intolerable. Fortunately, the entire conversion only required adding a dozen lines of code to two i/o functions and relinking.

The results were satisfying. On a typical 50^3 array, a trivial range computation dropped from 9 seconds elapsed time to less than a second. A graphics-intensive command improved from 17 seconds elapsed time to 6 seconds. Obviously, if we had been using binary i/o previously the improvement would have been less dramatic.

An implementation of the `shm_create` interface in terms of the vendor-supplied `shmget` was easy; the code is displayed in Figure 1 and is available by sending the message

```
send svid from research/shm
```

by electronic mail to `netlib@research.att.com` or `uunet!research!netlib`. Implementations for other systems would be welcome; for portability, a POSIX version based on binary disk files is planned.

The code is straightforward, though cluttered a little by error checking. The newline added to the label is perhaps obscure; in systems where shared memory segments are in the ordinary file system, that simplifies displaying the label.

5 Caveats

I wish to emphasize that I whole-heartedly endorse the long UNIX tradition of ASCII intermediate files. Their performance penalty is amply compensated by their self-documenting nature and the ease of connecting programs not deliberately designed to work together. In discussions on file exchange formats in the visualization community, I've fought for the ASCII camp against the binary camp, noting that one user of our toolbox even converted his binary, delta-encoded files into an ASCII format, ran compress, and wound up with a *smaller* file than the original. Netnews contains numerous queries from users of binary systems, asking how to convert their data; for ASCII systems, the solution is usually an obvious editor script.

The RenderMan Interface Bytestream[4] (RIB) adopts the useful approach of defining both an ASCII and a fully equivalent binary form, so that adherents of both approaches are satisfied. With i/o libraries that read either form and stand-alone filters that provide easy conversion, such a scheme meets diverse needs. (Ironically, in my experience a compressed ASCII RIB file is often smaller than the corresponding compressed binary RIB file.) The point being made here, however, is that if the only reason to give up ASCII is raw performance, one might wish to switch to shared memory rather than binary.

Although many commercial UNIX systems provide these shared memory functions, it should be noted that the SVID[1] states that these are optional and "may not be present in all implementations of the

```c
#include <stdio.h>
#include <sys/types.h>
#include <sys/ipc.h>
#include <sys/shm.h>
#define BAD (char *)(-1)

char* shm_create(char* name, mode_t mode, int nbytes)
{
     char*      shm = 0;
     shmid = shmget(IPC_PRIVATE, 128+nbytes, mode|IPC_CREAT);
     if (shmid >= 0) {
          shm = (char *)shmat(shmid, 0, 0);
          if (shm == BAD) {
               fprintf(stderr,"couldn't attach %d\n",shmid);
               shm = 0;
          } else  sprintf(shm, "%d %.100s%c\n", shmid, name, 0);
     }
     return(shm);
}

char* shm_open(char* id)
{
     char*      shm = 0;
     n = sscanf(id, "%d", &shmid);
     if (n == 1) {
          shm = (char*)shmat(shmid, 0, SHM_RDONLY);
          if (shm == BAD || shm == 0) shm = 0;
          else { /* finally, check that shm matches id */
               n = strlen(id);
               if (id[n-1] == '\n') n--;
               if (strncmp(shm, id, n) != 0) shm = 0;
          }
     }
     return(shm);
}

int shm_close(char* shm)
{
     return( shmdt(shm) );
}
```

Figure 1: Implementation of proposed bulk-transfer primitives using `shmget`.

Kernel Extension." Also, as noted above for pipeline speeds, your own mileage may vary.

There is certainly a place for binary i/o in applications needing high-speed, large-volume communication across networks. But for most purposes, human-readable formats are the way to go, with optional use of shared memory for transferring selected large chunks of data.

I thank Bill Coughran, David Gay, Cleve Moler, Dennis Ritchie, Tom Szymanski, and especially Brian Kernighan for illuminating remarks. RenderMan is a registered trademark of Pixar. UNIX is a registered trademark of UNIX System Laboratories, Inc. VAX is a trademark of Digital Equipment Corporation.

References

[1] AT&T, *System V Interface Definition: Issue 2*, vol. I, 1985.

[2] W. M. Coughran, Jr. and E. Grosse, *A philosophy for scientific computing tools*, SIGNUM Newsletter, 24:2/3 (1989), pp. 2–9.

[3] ——, *Techniques for scientific animation*, SPIE Proceedings, 1259 (1990), pp. 72–79.

[4] Pixar, *The RenderMan Interface, Version 3.1*, 1989. 3240 Kerner Blvd, San Rafael CA 94901.