Technical Articles & Tips

A Performance Comparison of "read" and "mmap"

By Oyetunde Fadele, September 2002

Summary

This article is for developers interested in implementing `mmap` as an alternative to the conventional `read` method of performing file input/output (I/O). We include an example implementation of `mmap`, which goes beyond the information available in the man page.

A sample image-processing application was developed for this article. Both `mmap` and `read` were implemented and compared, using an Ultra 30 workstation running the Solaris 8.0 Operating Environment (OE). Results show that the `mmap` method consistently took less time than `read`, by up to 103 percent.

Introduction

Server-side applications such as image-processing and server-logging software perform operations that often involve heavy use of file I/O. For sufficiently large files, memory contention can become an issue. To alleviate this problem, file I/O can now be done by mapping files directly to a process's address space, using `mmap`. The focus of this article is the possible performance gain from using `mmap` instead of the traditional `read`.

In the sample image-processing application that we developed, the original image is read into memory and then scaled, either reduced or enlarged. Two distinct methods are used to read the file into memory, one method implementing `read` and the other implementing `mmap`.

In the first method, an image file is read completely into memory with a `read` system call and stored in a buffer. Once the file is read into memory, a scaling algorithm is applied to the buffer containing the image data. This is referred to as the `read/scale/free` alternative.

In the second method, the image file is mapped into the process's address space with the `mmap` call, resulting in memory allocation into a buffer. The same scaling algorithm is then applied to the buffer. This is referred to as the `mmap/scale/free` alternative. This article reports and compares the time each alternative took to perform these operations.

The objectives of this article are:

- To compare the performance of the two alternatives under different conditions.
- To present a sample implementation of `mmap` for file I/O.

The following sections describe the concepts involved and the sample application, and also present code examples and generated data.

Theoretical Concepts

File I/O is traditionally done with `read`, `write`, and `lseek` system calls, but can also be done by mapping a file directly to a process's address space, using `mmap`. In traditional file I/O involving `read`, data is copied from the disk to a kernel buffer, then the kernel buffer is copied into the process's heap space for use. In memory-mapped file I/O, data is copied from the disk straight into the process's address space, into the segment where the file is mapped. The file is then accessed by references to memory locations with pointers. This way, I/O on the file is done without the overhead of handling the data twice; this translates into an improvement in system performance.

In general, the `mmap` function definition is given by:

```
void *mmap(void *addr, size_t len, int prot, int flags, int fildes,
off_t off);
```

The `mmap` method's arguments include an address `addr`, total size to be mapped `len`, file protection `prot`, mapping `flags`, a file descriptor `fildes` for an open file, and an `offset`. The flags allow different options to be passed to `mmap` to control the way a file is mapped to the process address space.

`mmap` options that are of interest in the sample application include MAP_SHARED for `flags` and PROT_READ for `prot`. MAP_SHARED was used so that, for a multiprocess application, changes made by one process are reflected across others. For more details, please refer to The Open Group's `mmap` man page and the chapter on file I/O in *Solaris Internals*.

When two processes map the same file, segments are created within each process that point to the same `vnode`. Each process has virtual memory mapping to the file, but they all share the same physical memory pages. The first segment to cause a page fault reads the page into physical memory, while each subsequent segment creates a reference to the existing physical memory page. Each physical page of memory is identified by its `vnode` and `offset`, used for tracking a page when it's not in physical memory. Anonymous memory allocation (pages not directly associated with a `vnode`) occurs when a zero-fill-on-demand (ZFOD) page fault occurs.

In conventional I/O using the system call, a buffer has to be created, typically by a malloc call. This buffer must then be freed using a `free` call. Memory mapping with `mmap` results in memory allocation and the kernel needs to be advised to free this memory. This is done with an `madvise` call, using a MADV_DONTNEED argument. A `munmap` call is used in conjunction with the `madvise` call to remove the reference to the previously mapped file.

Sample Application

A multithreaded sample application that performs scaling on an image file was developed for this article. The input parameters are the image file to be scaled, an optional output image, and a scaling factor. The image file is read into memory in two ways, as described in the introduction, that is, either with `read` or with `mmap`. After the file is read into memory, a scaling algorithm is applied to the image buffer. The scaled image object can then be written to an output file, although this was not implemented in the sample application.

Using the same input file, we measured and compared the total time taken to read the object into memory (store it in a buffer) in both cases (for `read` and for `mmap`), as well as the time taken to scale the data in the buffer and to free the buffer. PPM image files were used in this application.

The sample application is presented here mainly for illustration purposes. In practice, it may map to an image-processing web application that takes an image file (or a set of image files) as input. The set of image files could either be homogenous (the same file) or heterogeneous (different files). Other input parameters would include a scaling factor. The output of the application would be a set of scaled image files.

Implementation

Here we provide descriptions of key implementation details for the sample application, as well as code examples.

"read" and "mmap"

In the following example, for the 'type==READ' condition, the image file of interest is opened and a file descriptor is obtained. Memory is then allocated to the image buffer, which is populated by the read call after the `lseek` call moves the file descriptor to the correct position. For the 'type==MMAP' condition, the image data buffer is allocated with a `mmap` call, based on read-only, shared mapping, and an open file descriptor. The `mmap` call returns the address where the file is mapped, which is assigned to the image data pointer.

```
IImageP *
readData (IImageP * dataimg, int type)
{
? IImageP *readimage;
? int w, h;
? int i;
? unsigned int temp;
? unsigned char *ptr;
? char *p;
? char *comments = NULL;
? int maxcolors = 255;
? int greyscale = 0;
? int bytesperpixel = 3;
? int offset = dataimg->offset -1;
? int fildes;
? int whence;
? hrtime_t beginreadData, endreadData ;
? int size = dataimg->width * dataimg->height *
      bytesperpixel;
? beginreadData = gethrtime();
?
? if (type==READ) {
??? if ((fildes = open(dataimg->filename, O_RDONLY))
        < 0) {
????? fprintf(stderr, "can't open %sn",
          dataimg->filename);
?
??? }
??? dataimg->data = (unsigned char *) malloc (size);
??? fprintf(stderr, "in readData: dataimg->data =
        %pn", dataimg->data );
??? lseek(fildes, (offset-1), SEEK_SET);
??? read(fildes, dataimg->data, size);
??? close (fildes);
? }
? if (type==MMAP){
?
??? if ((fildes = open(dataimg->filename, O_RDONLY)) < 0) {
????? fprintf(stderr, "can't open %sn",
          dataimg->filename);
??? }
?
??? fprintf(stderr, "file = %sn", dataimg->filename);
??? dataimg->data = (unsigned char *)mmap( (caddr_t)0,
        size, PROT_READ , MAP_PRIVATE,fildes, 0) + (offset-1) ;
??? fprintf(stderr, "page size = %d", sysconf (_SC_PAGESIZE));
```

```
??? if (dataimg->data == MAP_FAILED) {
?fprintf(stderr, "mmap failed");
?perror ("mmap failed due to: ");
?exit(0);
??? }
?
? }
?
? endreadData = gethrtime();
? printf("Total readData() time?? = %lld nsecn",
     (endreadData - beginreadData) );
? return dataimg;
}
```

Scale

A scaling algorithm for reduction is presented here for obtaining a scaled image buffer from an original image buffer, based on pointer arithmetic. This same method is used for both the `read` and the `mmap` alternatives.

```
IImageP *
scaleImage(int scalingvalue, IImageP * origimage )
{
? hrtime_t beginscaleImage, endscaleImage;
? IImageP *scaledimage;
? unsigned int src_width, src_height, dest_width, dest_height;
? unsigned char *ptr;
? unsigned char *ptrX;
? int scalex, scaley, tempx, tempy;
? int stride;
? int x,y,x2,y2, i, j,k,l;
? beginscaleImage = gethrtime();
? scaledimage = (IImageP *) malloc ( sizeof ( IImageP ) );?
      /* need to do a free soon? */
? if (scalingvalue > 0) {
??? scaledimage->width =?? scalingvalue * (origimage->width);
??? scaledimage->height =? scalingvalue * (origimage->height);
??? /* Algorithm for scaling up? */
??? scalex =?? scalingvalue;
??? scaley =?? scalingvalue;
??? stride =? scalex * origimage->width;
? }
? if (scalingvalue < 0) {
??? scaledimage->width =??? (int)
        (origimage->width)/(double)abs(scalingvalue);
??? scaledimage->height =? (int)?
        (origimage->height)/(double)abs(scalingvalue);
??? /* Algorithm for scaling down? */
??? scalex =??? abs(scalingvalue);
??? scaley =??? abs(scalingvalue);
??? stride =? origimage->width/scalex;?? /*
        losing some accuracy here?? */
? }
? scaledimage->offset = origimage->offset;
? scaledimage->filename = origimage->filename;
? scaledimage->comments = origimage->comments;
? scaledimage->data = (unsigned char *) malloc
      ( scaledimage->width * scaledimage->height * 3 );
? if (scaledimage == NULL) {
???? fprintf (stderr, "Could not create scaled image");
? }
? src_width = origimage->width;
? src_height = origimage->height;
? dest_width =? scaledimage->width;
? dest_height =? scaledimage->height;
? /*? for each pixel in the image, enlargement???? */
? if (scalingvalue > 0) {
??? for ( i = 0; i <? src_height; i++ ) {
????? for ( j = 0; j < src_width; j++ ) {
??????? for (k = 0; k < scaley; k++) {
????????? for (l = 0; l < scalex; l++) {
???? *(scaledimage->data + ((i*scaley + k)*stride) +
         (j*scalex) + l) = *(origimage->data + i*src_width + j);
?? }
??????? }
????? }
??? }
? }
? /*? for each pixel in the image, reduction , skipping
      some pixels??? */
? if (scalingvalue < 0) {
??? for ( i = 0; i <? src_height; i = i + scalex ) {
????? for ( j = 0; j < src_width; j = j + scaley ) {
```

```
???? *(scaledimage->data + ((i/scaley)*stride) +
         (j/scalex)) = *(origimage->data + i*src_width + j);
????? }
??? }
? }
? endscaleImage = gethrtime();
? printf("Total ScaleImage() time?? = %lld nsecn",
      (endscaleImage - beginscaleImage) );
? return scaledimage;
}
```

Tests

A 296-MHz, 512-Mbyte Ultra 30 workstation running Solaris 8.0 OE generated the results shown in Table 1 and Figure 1. The performance metric is the total time taken to read an image into a buffer, scale it, and free the image. In the first alternative, this is the time it takes to do `read/scale/free`, while in the second alternative, this is the time for `mmap/scale/free`.

A 24-bit, 3.75-Mbyte, color PPM image (1280 X 1024) was used for testing. Reduction was performed. In Table 1, a factor of two refers to reduction to one-quarter of the original image, that is, half-width by half-height. `vmstat` output was measured during the tests for one-second intervals. We monitored paging, scanning, and minor page faults in the system, with `read` as well as with `mmap`.

Tests were run in alternating order for 10 test cases, that is, `read` before `mmap`, and vice versa. The repeatability of the tests was then measured with data for the mean and standard deviation. The performance data was then reported as the ratio of the time taken to perform `read/scale/free` to the time taken to perform `mmap/scale/free`, for a single, multithreaded process. This is reported as "read time/mmap time." From the data in Table 1, this ratio is always greater than one, indicating that the time taken to perform `read/scale/free` is always longer than the time taken to do `mmap/scale/free`.

Data

```
Nthreads = 10
Nruns = 10
```

Results

Figure 1 is a plot of the ratio of `read` time to `mmap` time, as a function of scale factor. Table 1 provides the data for Figure 1. As the scale factor increases, the "read time/mmap time" ratio increases, indicating that the `read/scale/free` alternative consistently takes more time than the `mmap/scale/free` alternative. The ratio of `read` time to `mmap` time ranges from about 15 percent at a scale factor of 2 to about 103 percent at a scale factor of 8.

Table 1: Performance Data for Multithreaded Operation Within a Single Process

Scale Factor	Mean "read" Time (sec)	Mean "mmap" Time (sec)	Mean "read" Time (sec)/Mean "mmap" Time (sec)
2	5.48	4.79	1.14
3	3.11	2.44	1.28
4	2.25	1.52	1.48
8	1.26	0.62	2.03

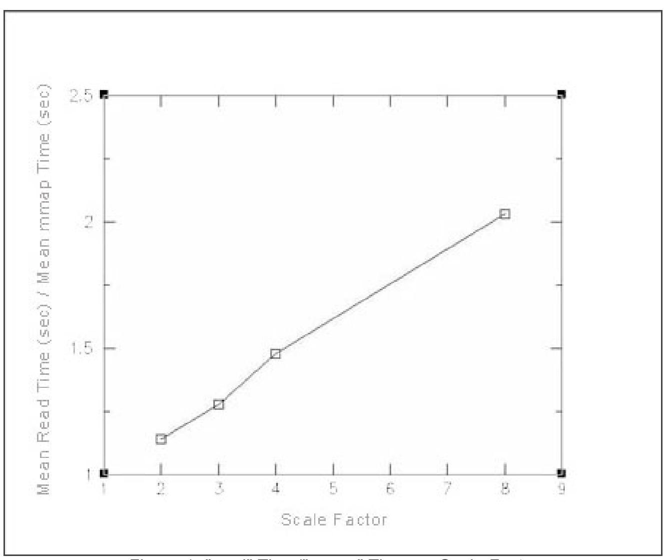

Figure 1: "read" Time/"mmap" Time vs. Scale Factor

Conclusion

We have shown that file I/O performs better with memory mapping than with traditional system calls. Results from the sample application show that the `mmap/scale/free` alternative consistently takes less time than the equivalent `read/scale/free` alternative. The ratio of the time taken for each method ranges from about 1.15 when the image is reduced by half, to about 2.03 when reduced by one-eighth.

References

1. Technical Article on Solaris Developer Connection, "File I/O"
2. The Open Group's `mmap` man page
3. Ilib source
4. XIL Home Page
5. *Solaris Internals*, by Richard McDougall and Jim Mauro, 2000

About the Author

Oyetunde Fadele has been at Sun for more than two years. A software engineer in Market Development Engineering (MDE), he works with ISVs in the areas of content management, software tools, and media/image processing. Projects include performance tuning, sizing and scaling, development support, and porting on the Java and Solaris platforms. You can reach him at: oyetunde.fadele@sun.com.

September 2002