5.3 Mapped Memory

Mapped memory permits different processes to communicate via a shared file. Although you can think of mapped memory as using a shared memory segment with a name, you should be aware that there are technical differences. Mapped memory can be used for interprocess communication or as an easy way to access the contents of a file.

Mapped memory forms an association between a file and a process's memory. Linux splits the file into page-sized chunks and then copies them into virtual memory pages so that they can be made available in a process's address space. Thus, the process can read the file's contents with ordinary memory access. It can also modify the file's contents by writing to memory. This permits fast access to files.

You can think of mapped memory as allocating a buffer to hold a file's entire contents, and then reading the file into the buffer and (if the buffer is modified) writing the buffer back out to the file afterward. Linux handles the file reading and writing operations for you.

There are uses for memory-mapped files other than interprocess communication. Some of these are discussed in Section 5.3.5, "Other Uses for mmap."

5.3.1 Mapping an Ordinary File

To map an ordinary file to a process's memory, use the `mmap` ("Memory MAPped," pronounced "em-map") call. The first argument is the address at which you would like Linux to map the file into your process's address space; the value NULL allows Linux to choose an available start address. The second argument is the length of the map in bytes. The third argument specifies the protection on the mapped address range. The protection consists of a bitwise "or" of `PROT_READ`, `PROT_WRITE`, and `PROT_EXEC`, corresponding to read, write, and execution permission, respectively. The fourth argument is a flag value that specifies additional options. The fifth argument is a file descriptor opened to the file to be mapped. The last argument is the offset from the beginning of the file from which to start the map. You can map all or part of the file into memory by choosing the starting offset and length appropriately.

The flag value is a bitwise "or" of these constraints:

- `MAP_FIXED`— If you specify this flag, Linux uses the address you request to map the file rather than treating it as a hint. This address must be page-aligned.
- `MAP_PRIVATE`— Writes to the memory range should not be written back to the attached file, but to a private copy of the file. No other process sees these writes. This mode may not be used with `MAP_SHARED`.
- `MAP_SHARED`— Writes are immediately reflected in the underlying file rather than buffering writes. Use this mode when using mapped memory for IPC. This mode may not be used with `MAP_PRIVATE`.

If the call succeeds, it returns a pointer to the beginning of the memory. On failure, it returns `MAP_FAILED`.

When you're finished with a memory mapping, release it by using `munmap`. Pass it the start address and length of the mapped memory region. Linux automatically unmaps mapped regions when a process terminates.

5.3.2 Example Programs

Let's look at two programs to illustrate using memory-mapped regions to read and write to files. The first program, Listing 5.5, generates a random number and writes it to a memory-mapped file. The second program, Listing 5.6, reads the number, prints it, and replaces it in the memory-mapped file with double the value. Both take a command-line argument of the file to map.

Listing 5.5 (*mmap-write.c*) Write a Random Number to a Memory-Mapped File

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <time.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

/* Return a uniformly random number in the range [low,high]. */

int random_range (unsigned const low, unsigned const high)
{
  unsigned const range = high - low + 1;
  return low + (int) (((double) range) * rand () / (RAND_MAX + 1.0));
}

int main (int argc, char* const argv[])
{
  int fd;
  void* file_memory;

  /* Seed the random number generator. */
  srand (time (NULL));

  /* Prepare a file large enough to hold an unsigned integer. */
  fd = open (argv[1], O_RDWR | O_CREAT, S_IRUSR | S_IWUSR);
  lseek (fd, FILE_LENGTH+1, SEEK_SET);
  write (fd, "", 1);
  lseek (fd, 0, SEEK_SET);

  /* Create the memory mapping. */
  file_memory = mmap (0, FILE_LENGTH, PROT_WRITE, MAP_SHARED, fd, 0);
```

```
  close (fd);
  /* Write a random integer to memory-mapped area. */
  sprintf((char*) file_memory, "%d\n", random_range (-100, 100));
  /* Release the memory (unnecessary because the program exits). */
  munmap (file_memory, FILE_LENGTH);

  return 0;
}
```

The `mmap-write` program opens the file, creating it if it did not previously exist. The third argument to `open` specifies that the file is opened for reading and writing. Because we do not know the file's length, we use `lseek` to ensure that the file is large enough to store an integer and then move back the file position to its beginning.

The program maps the file and then closes the file descriptor because it's no longer needed. The program then writes a random integer to the mapped memory, and thus the file, and unmaps the memory. The `munmap` call is unnecessary because Linux would automatically unmap the file when the program terminates.

Listing 5.6 (*mmap-read.c*) Read an Integer from a Memory-Mapped File, and Double It

```
#include <stdlib.h>
#include <stdio.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <sys/stat.h>
#include <unistd.h>
#define FILE_LENGTH 0x100

int main (int argc, char* const argv[])
{
    int fd;
    void* file_memory;
    int integer;

    /* Open the file. */
    fd = open (argv[1], O_RDWR, S_IRUSR | S_IWUSR);
    /* Create the memory mapping. */
    file_memory = mmap (0, FILE_LENGTH, PROT_READ | PROT_WRITE,
                        MAP_SHARED, fd, 0);
    close (fd);
    /* Read the integer, print it out, and double it.  */
    sscanf (file_memory, "%d", &integer);
```

```
    printf ("value: %d\n", integer);
    sprintf ((char*) file_memory, "%d\n", 2 * integer);
    /* Release the memory (unnecessary because the program exits). */
    munmap (file_memory, FILE_LENGTH);

    return 0;
}
```

The `mmap-read` program reads the number out of the file and then writes the doubled value to the file. First, it opens the file and maps it for reading and writing. Because we can assume that the file is large enough to store an unsigned integer, we need not use `lseek`, as in the previous program. The program reads and parses the value out of memory using `sscanf` and then formats and writes the double value using `sprintf`.

Here's an example of running these example programs. It maps the file `/tmp/integer-file`.

```
% ./mmap-write /tmp/integer-file
% cat /tmp/integer-file
42
% ./mmap-read /tmp/integer-file
value: 42
% cat /tmp/integer-file
84
```

Observe that the text `42` was written to the disk file without ever calling `write`, and was read back in again without calling `read`. Note that these sample programs write and read the integer as a string (using `sprintf` and `sscanf`) for demonstration purposes only—there's no need for the contents of a memory-mapped file to be text. You can store and retrieve arbitrary binary in a memory-mapped file.

5.3.3 Shared Access to a File

Different processes can communicate using memory-mapped regions associated with the same file. Specify the `MAP_SHARED` flag so that any writes to these regions are immediately transferred to the underlying file and made visible to other processes. If you don't specify this flag, Linux may buffer writes before transferring them to the file.

Alternatively, you can force Linux to incorporate buffered writes into the disk file by calling `msync`. Its first two parameters specify a memory-mapped region, as for `munmap`. The third parameter can take these flag values:

- `MS_ASYNC`— The update is scheduled but not necessarily run before the call returns.
- `MS_SYNC`— The update is immediate; the call to `msync` blocks until it's done. `MS_SYNC` and `MS_ASYNC` may not both be used.
- `MS_INVALIDATE`— All other file mappings are invalidated so that they can see the updated values.

For example, to flush a shared file mapped at address `mem_addr` of length `mem_length` bytes, call this:

```
msync (mem_addr, mem_length, MS_SYNC | MS_INVALIDATE);
```

As with shared memory segments, users of memory-mapped regions must establish and follow a protocol to avoid race conditions. For example, a semaphore can be used to prevent more than one process from accessing the mapped memory at one time. Alternatively, you can use `fcntl` to place a read or write lock on the file, as described in Section 8.3, "fcntl: Locks and Other File Operations," in Chapter 8.

5.3.4 Private Mappings

Specifying `MAP_PRIVATE` to `mmap` creates a copy-on-write region. Any write to the region is reflected only in this process's memory; other processes that map the same file won't see the changes. Instead of writing directly to a page shared by all processes, the process writes to a private copy of this page. All subsequent reading and writing by the process use this page.

5.3.5 Other Uses for *mmap*

The `mmap` call can be used for purposes other than interprocess communications. One common use is as a replacement for `read` and `write`. For example, rather than explicitly reading a file's contents into memory, a program might map the file into memory and scan it using memory reads. For some programs, this is more convenient and may also run faster than explicit file I/O operations.

One advanced and powerful technique used by some programs is to build data structures (ordinary `struct` instances, for example) in a memory-mapped file. On a subsequent invocation, the program maps that file back into memory, and the data structures are restored to their previous state. Note, though, that pointers in these data structures will be invalid unless they all point within the same mapped region of memory and unless care is taken to map the file back into the same address region that it occupied originally.

Another handy technique is to map the special `/dev/zero` file into memory. That file, which is described in Section 6.5.2, "/dev/zero, " of Chapter 6, "Devices," behaves as if it were an infinitely long file filled with 0 bytes. A program that needs a source of 0 bytes can `mmap` the file `/dev/zero`. Writes to `/dev/zero` are discarded, so the mapped memory may be used for any purpose. Custom memory allocators often map `/dev/zero` to obtain chunks of preinitialized memory.