The protection specified for a region has the open mode of the file.

**Figure 12.18** Protection of a memory mapped region

<table>
<thead>
<tr>
<th>Region</th>
<th>Protection</th>
<th>Notes</th>
</tr>
</thead>
<tbody>
<tr>
<td>root</td>
<td>REPEXEC</td>
<td>Region is the root of the process.</td>
</tr>
<tr>
<td>root</td>
<td>REPEXEC</td>
<td>Region can only be opened in the process.</td>
</tr>
<tr>
<td>root</td>
<td>REPEXEC</td>
<td>Region can be opened in the process.</td>
</tr>
<tr>
<td>root</td>
<td>REPEXEC</td>
<td>Region can be opened in the process.</td>
</tr>
<tr>
<td>root</td>
<td>REPEXEC</td>
<td>Region can be opened in the process.</td>
</tr>
</tbody>
</table>

The protection of the region is specified as either 'REPEXEC', 'REPEXEC', or 'REPEXEC'.

In this figure, the region is shown in a memory mapped file. The memory mapped region is a virtual memory region that is mapped to a file. The region can be opened in the process, and the process can access the region.

**Figure 12.19** Example of a memory mapped file

```
+----------+  +----------+  +----------+
| region   |  | region   |  | region   |
| protection|  | protection|  | protection|
| REPEXEC   |  | REPEXEC   |  | REPEXEC   |
| REPEXEC   |  | REPEXEC   |  | REPEXEC   |
| REPEXEC   |  | REPEXEC   |  | REPEXEC   |
| REPEXEC   |  | REPEXEC   |  | REPEXEC   |
| REPEXEC   |  | REPEXEC   |  | REPEXEC   |
```

The figure shows an example of a memory mapped file. The file is divided into regions, each with a protection level. The regions are mapped to virtual memory, allowing the process to access the file as if it were memory.

Before looking at the mapping address, let's see where it's going on here. The value of the address (in memory) and if it's the starting address in the file of the process to map. There are some restrictions.
Memory I/O

Example

The figure above shows the various attributes of the mapped region.

The memory region is defined in the code, as indicated in the above code snippet. The code also specifies the lower and upper addresses of the mapped region.

By modifying the upper boundary of the mapped region, the code sets the size of the region to 1GB (1024MB).

The memory region is accessed by specifying the address of the mapped region.

The mapped region is accessed by specifying the address of the mapped region.
The times are given in seconds and the size of the file being copied was almost 3 million bytes. For the SPARC, the total CPU time (user + system) is the same for both types of copies, 2.5 seconds. (This is familiar to what we found for wc.ter in Figure 12.8. For

![Figure 12.21 Timing results comparing read/write to memory mapped I/O.](image)

**Example**

Program 12.14 copies a file (similar to the cdf(1) command) using memory mapped I/O.

```c
#include <sys/types.h>

In summary (caddr_t addr, size_t size):

Remain 0 OK - on error

*Memory Mapped I/O*

A memory mapped region is automatically unmapped when the process terminates, or by calling munmap directly. Closing the file descriptor, on the other hand, does not unmmap.

Some systems provide an msync function that is similar to sync(2). Section 5.4.1, but only on

---

(Continued on next page)

---

(Continued on previous page)

---

(Continued on next page)

---

(Continued on previous page)

---

(Continued on next page)

---

(Continued on previous page)

---
Program 12.14 Copy a single memory mapped I/O

/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
/*

*/
In the chapter, we described numerous advanced I/O functions, most of which are used in the example in later chapters.

12.10 Summary

process

The read and write functions (also used in many of the later examples)
by the kernel require
the I/O multiplexing—see section 12.7.4, since there is only one device.
Your device driver will need to understand how to use the I/O system
and the system library in Chapter 16.

Reading I/O—issuing an I/O operation without blocking a block (will need
nonblocking I/O)—issuing an I/O operation without blocking a block.

In summary, we describe the read and write functions, their
use in various contexts, and how to use them in your device driver.

12.11 Advanced Topics

In this chapter, we describe several advanced topics, including
I/O multiplexing, nonblocking I/O, and various I/O utilities.

I/O multiplexing is a technique used to manage multiple
I/O requests simultaneously. This allows the processor to
switch between different I/O operations, improving
performance and reducing
response time.

Nonblocking I/O is a feature that allows
an I/O operation to continue without
waiting for the operation to complete. This
is useful in situations where
performance is critical,
and the system cannot
afford to wait for I/O
operations to complete.

I/O utilities are system
programs that manage
I/O operations. These
utilities can be used
to facilitate
communication
between the system
and the user, allowing
users to perform
operations
without
interrupting
the system.