While address spaces these days are large, their sizes used to be a serious problem. Even today, though, separate I- and D-spaces are still common. However, rather than for the normal address spaces, they are now used to divide the L1 cache. After all, in the L1 cache, memory is still plenty scarce.

3.5.5 Shared Pages

Another design issue is sharing. In a large multiprogramming system, it is common for several users to be running the same program at the same time. Even a single user may be running several programs that use the same library. It is clearly more efficient to share the pages, to avoid having two copies of the same page in memory at the same time. One problem is that not all pages are sharable. In particular, pages that are read-only, such as program text, can be shared, but for data pages sharing is more complicated.

If separate I- and D-spaces are supported, it is relatively straightforward to share programs by having two or more processes use the same page table for their I-space but different page tables for their D-spaces. Typically in an implementation that supports sharing in this way, page tables are data structures independent of the process table. Each process then has two pointers in its process table: one to the I-space page table and one to the D-space page table, as shown in Fig. 3-25. When the scheduler chooses a process to run, it uses these pointers to locate the appropriate page tables and sets up the MMU using them. Even without separate I- and D-spaces, processes can share programs (or sometimes, libraries), but the mechanism is more complicated.

When two or more processes share some code, a problem occurs with the shared pages. Suppose that processes *A* and *B* are both running the editor and sharing its pages. If the scheduler decides to remove *A* from memory, evicting all its pages and filling the empty page frames with some other program will cause *B* to generate a large number of page faults to bring them back in again.

Similarly, when *A* terminates, it is essential to be able to discover that the pages are still in use so that their disk space will not be freed by accident. Searching all the page tables to see if a page is shared is usually too expensive, so special data structures are needed to keep track of shared pages, especially if the unit of sharing is the individual page (or run of pages), rather than an entire page table.

Sharing data is trickier than sharing code, but it is not impossible. In particular, in UNIX, after a fork system call, the parent and child are required to share both program text and data. In a paged system, what is often done is to give each of these processes its own page table and have both of them point to the same set of pages. Thus no copying of pages is done at fork time. However, all the data pages are mapped into both processes as READ ONLY.

As long as both processes just read their data, without modifying it, this situation can continue. As soon as either process updates a memory word, the violation of the read-only protection causes a trap to the operating system. A copy is then

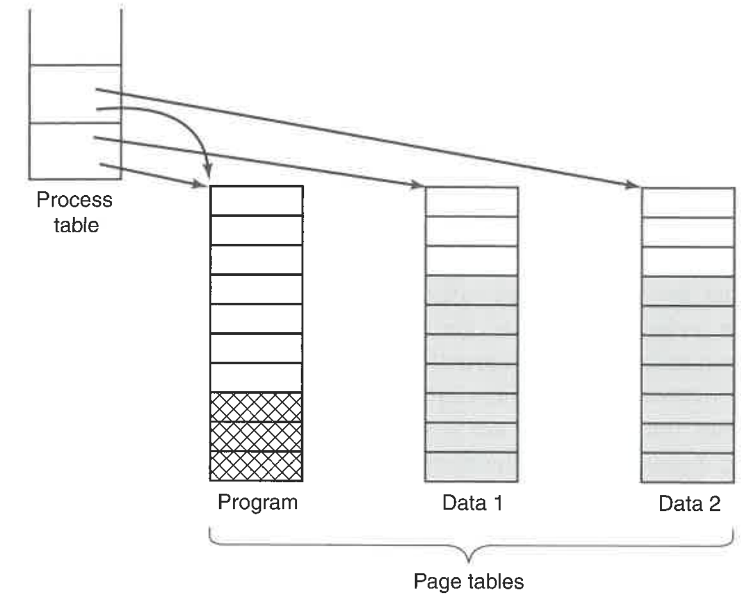

Figure 3-25. Two processes sharing the same program sharing its page tables.

made of the offending page so that each process now has its own private copy. Both copies are now set to READ/WRITE, so subsequent writes to either copy proceed without trapping. This strategy means that those pages that are never modified (including all the program pages) need not be copied. Only the data pages that are actually modified need to be copied. This approach, called **copy on write**, improves performance by reducing copying.

3.5.6 Shared Libraries

Sharing can be done at other granularities than individual pages. If a program is started up twice, most operating systems will automatically share all the text pages so that only one copy is in memory. Text pages are always read only, so there is no problem here. Depending on the operating system, each process may get its own private copy of the data pages, or they may be shared and marked read only. If any process modifies a data page, a private copy will be made for it, that is, copy on write will be applied.

In modern systems, there are many large libraries used by many processes, for example, multiple I/O and graphics libraries. Statically binding all these libraries to every executable program on the disk would make them even more bloated than they already are.

Instead, a common technique is to use **shared libraries** (which are called **DLLs** or **Dynamic Link Libraries** on Windows). To make the idea of a shared library clear, first consider traditional linking. When a program is linked, one or more object files and possibly some libraries are named in the command to the linker, such as the UNIX command

```
ld *.o –lc –lm
```

which links all the *.o* (object) files in the current directory and then scans two libraries, */usr/lib/libc.a* and */usr/lib/libm.a*. Any functions called in the object files but not present there (e.g., *printf*) are called **undefined externals** and are sought in the libraries. If they are found, they are included in the executable binary. Any functions that they call but are not yet present also become undefined externals. For example, *printf* needs *write*, so if *write* is not already included, the linker will look for it and include it when found. When the linker is done, an executable binary file is written to the disk containing all the functions needed. Functions present in the libraries but not called are not included. When the program is loaded into memory and executed, all the functions it needs are there.

Now suppose common programs use 20–50 MB worth of graphics and user interface functions. Statically linking hundreds of programs with all these libraries would waste a tremendous amount of space on the disk as well as wasting space in RAM when they were loaded since the system would have no way of knowing it could share them. This is where shared libraries come in. When a program is linked with shared libraries (which are slightly different than static ones), instead of including the actual function called, the linker includes a small stub routine that binds to the called function at run time. Depending on the system and the configuration details, shared libraries are loaded either when the program is loaded or when functions in them are called for the first time. Of course, if another program has already loaded the shared library, there is no need to load it again—that is the whole point of it. Note that when a shared library is loaded or used, the entire library is not read into memory in a single blow. It is paged in, page by page, as needed, so functions that are not called will not be brought into RAM.

In addition to making executable files smaller and also saving space in memory, shared libraries have another important advantage: if a function in a shared library is updated to remove a bug, it is not necessary to recompile the programs that call it. The old binaries continue to work. This feature is especially important for commercial software, where the source code is not distributed to the customer. For example, if Microsoft finds and fixes a security error in some standard DLL, *Windows Update* will download the new DLL and replace the old one, and all programs that use the DLL will automatically use the new version the next time they are launched.

Shared libraries come with one little problem, however, that has to be solved, however. The problem is illustrated in Fig. 3-26. Here we see two processes sharing a library of size 20 KB (assuming each box is 4 KB). However, the library is located at a different address in each process, presumably because the programs themselves are not the same size. In process 1, the library starts at address 36K; in

process 2 it starts at 12K. Suppose that the first thing the first function in the library has to do is jump to address 16 in the library. If the library were not shared, it could be relocated on the fly as it was loaded so that the jump (in process 1) could be to virtual address 36K + 16. Note that the physical address in the RAM where the library is located does not matter since all the pages are mapped from virtual to physical addresses by the MMU hardware.

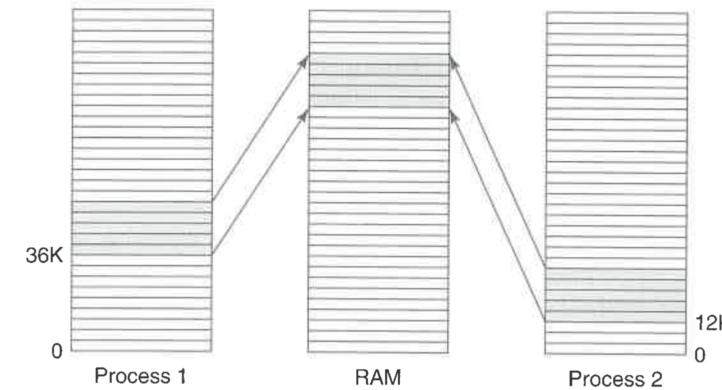

Figure 3-26. A shared library being used by two processes.

However, since the library is shared, relocation on the fly will not work. After all, when the first function is called by process 2 (at address 12K), the jump instruction has to go to 12K + 16, not 36K + 16. This is the little problem. One way to solve it is to use copy on write and create new pages for each process sharing the library, relocating them on the fly as they are created, but this scheme defeats the purpose of sharing the library, of course.

A better solution is to compile shared libraries with a special compiler flag telling the compiler not to produce any instructions that use absolute addresses. Instead only instructions using relative addresses are used. For example, there is almost always an instruction that says jump forward (or backward) by *n* bytes (as opposed to an instruction that gives a specific address to jump to). This instruction works correctly no matter where the shared library is placed in the virtual address space. By avoiding absolute addresses, the problem can be solved. Code that uses only relative offsets is called **position-independent code**.

3.5.7 Mapped Files

Shared libraries are really a special case of a more general facility called **memory-mapped files**. The idea here is that a process can issue a system call to map a file onto a portion of its virtual address space. In most implementations, no pages are brought in at the time of the mapping, but as pages are touched, they are demand paged in one page at a time, using the disk file as the backing store. When