

initial value is 0. In practice, most global variables are not initialized explicitly, and are thus 0. This could be implemented by simply having a section of the executable binary file exactly equal to the number of bytes of data, and initializing all of them, including the ones that have defaulted to 0.

However, to save space in the executable file, this is not done. Instead, the file contains all the explicitly initialized variables following the program text. The uninitialized variables are all gathered together after the initialized ones, so all the compiler has to do is put a word in the header telling how many bytes to allocate.

To make this point more explicit, consider Fig. 10-12(a) again. Here the program text is 8 KB and the initialized data is also 8 KB. The uninitialized data (BSS) is 4 KB. The executable file is only 16 KB (text + initialized data), plus a short header that tells the system to allocate another 4 KB after the initialized data and zero it before starting the program. This trick avoids storing 4 KB of zeros in the executable file.

In order to avoid allocating a physical page frame full of zeros, during initialization Linux allocates a static *zero page*, a write-protected page full of zeros. When a process is loaded, its uninitialized data region is set to point to the zero page. Whenever a process actually attempts to write in this area, the copy-on-write mechanism kicks in, and an actual page frame is allocated to the process.

Unlike the text segment, which cannot change, the data segment can change. Programs modify their variables all the time. Furthermore, many programs need to allocate space dynamically, during execution. Linux handles this by permitting the data segment to grow and shrink as memory is allocated and deallocated. A system call, `brk`, is available to allow a program to set the size of its data segment. Thus to allocate more memory, a program can increase the size of its data segment. The C library procedure *malloc*, commonly used to allocate memory, makes heavy use of it. The process address-space descriptor contains information on the range of dynamically allocated memory areas in the process, typically called the **heap**.

The third segment is the stack segment. On most machines, it starts at or near the top of the virtual address space and grows down toward 0. For instance, on 32bit x86 platforms, the stack starts at address 0xC0000000, which is the 3-GB virtual address limit visible to the process in user mode. If the stack grows below the bottom of the stack segment, a hardware fault occurs and the operating system lowers the bottom of the stack segment by one page. Programs do not explicitly manage the size of the stack segment.

When a program starts up, its stack is not empty. Instead, it contains all the environment (shell) variables as well as the command line typed to the shell to invoke it. In this way, a program can discover its arguments. For example, when

```
cp src dest
```

is typed, the *cp* program is run with the string “cp src dest” on the stack, so it can find out the names of the source and destination files. The string is represented as an array of pointers to the symbols in the string, to make parsing easier.

When two users are running the same program, such as the editor, it would be possible, but inefficient, to keep two copies of the editor's program text in memory at once. Instead, Linux systems support **shared text segments**. In Fig. 10-12(a) and Fig. 10-12(c) we see two processes, *A* and *B*, that have the same text segment. In Fig. 10-12(b) we see a possible layout of physical memory, in which both processes share the same piece of text. The mapping is done by the virtual-memory hardware.

Data and stack segments are never shared except after a fork, and then only those pages that are not modified. If either one needs to grow and there is no room adjacent to it to grow into, there is no problem since adjacent virtual pages do not have to map onto adjacent physical pages.

On some computers, the hardware supports separate address spaces for instructions and data. When this feature is available, Linux can use it. For example, on a computer with 32-bit addresses, if this feature is available, there would be 2^{32} bits of address space for instructions and an additional 2^{32} bits of address space for the data and stack segments to share. A jump or branch to 0 goes to address 0 of text space, whereas a move from 0 uses address 0 in data space. This feature doubles the address space available.

In addition to dynamically allocating more memory, processes in Linux can access file data through **memory-mapped files**. This feature makes it possible to map a file onto a portion of a process' address space so that the file can be read and written as if it were a byte array in memory. Mapping a file in makes random access to it much easier than using I/O system calls such as `read` and `write`. Shared libraries are accessed by mapping them in using this mechanism. In Fig. 10-13 we see a file that is mapped into two processes at the same time, at different virtual addresses.

An additional advantage of mapping a file in is that two or more processes can map in the same file at the same time. Writes to the file by any one of them are then instantly visible to the others. In fact, by mapping in a scratch file (which will be discarded after all the processes exit), this mechanism provides a high-bandwidth way for multiple processes to share memory. In the most extreme case, two (or more) processes could map in a file that covers the entire address space, giving a form of sharing that is partway between separate processes and threads. Here the address space is shared (like threads), but each process maintains its own open files and signals, for example, which is not like threads. In practice, however, making two address spaces exactly correspond is never done.

10.4.2 Memory Management System Calls in Linux

POSIX does not specify any system calls for memory management. This topic was considered too machine dependent for standardization. Instead, the problem was swept under the rug by saying that programs needing dynamic memory management can use the *malloc* library procedure (defined by the ANSI C standard).

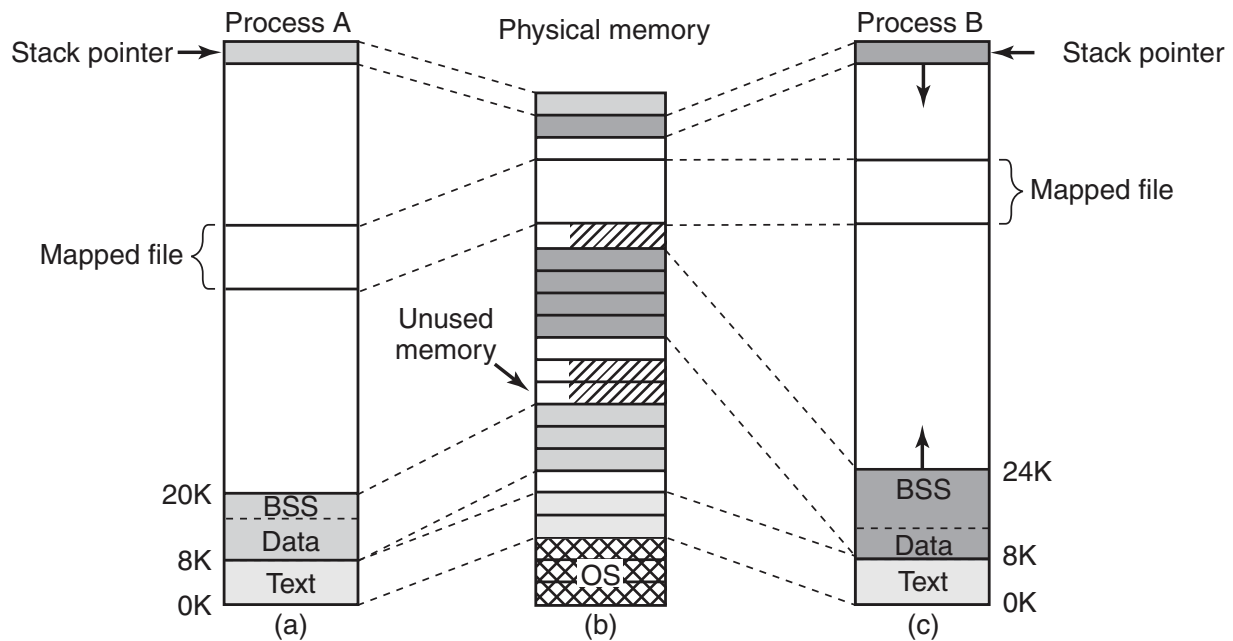


Figure 10-13. Two processes can share a mapped file.

How *malloc* is implemented is thus moved outside the scope of the POSIX standard. In some circles, this approach is known as passing the buck.

In practice, most Linux systems have system calls for managing memory. The most common ones are listed in Fig. 10-14. *Brk* specifies the size of the data segment by giving the address of the first byte beyond it. If the new value is greater than the old one, the data segment becomes larger; otherwise it shrinks.

System call	Description
<code>s = brk(addr)</code>	Change data segment size
<code>a = mmap(addr, len, prot, flags, fd, offset)</code>	Map a file in
<code>s = unmmap(addr, len)</code>	Unmap a file

Figure 10-14. Some system calls relating to memory management. The return code *s* is `-1` if an error has occurred; *a* and *addr* are memory addresses, *len* is a length, *prot* controls protection, *flags* are miscellaneous bits, *fd* is a file descriptor, and *offset* is a file offset.

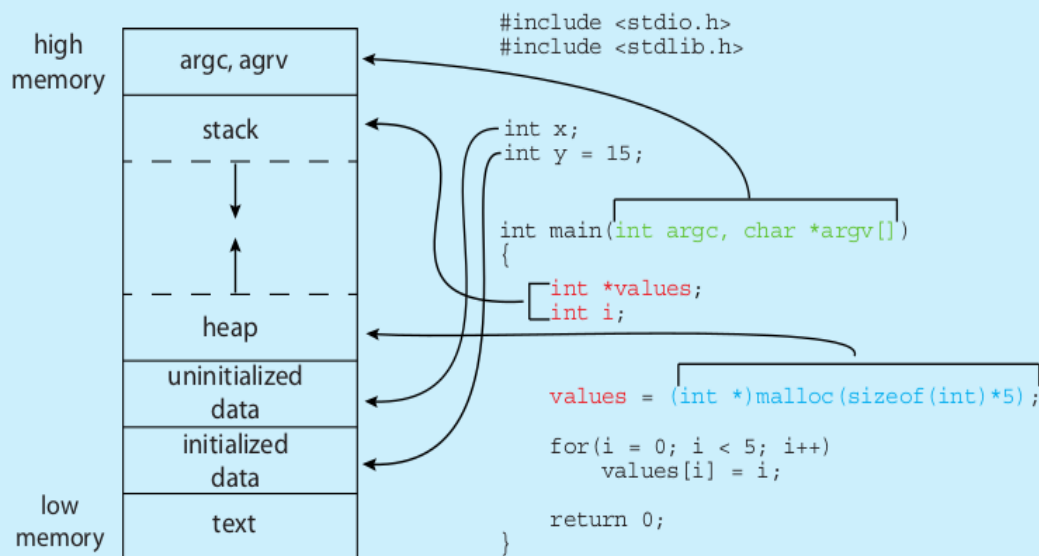
The *mmap* and *munmap* system calls control memory-mapped files. The first parameter to *mmap*, *addr*, determines the address at which the file (or portion thereof) is mapped. It must be a multiple of the page size. If this parameter is 0, the system determines the address itself and returns it in *a*. The second parameter, *len*, tells how many bytes to map. It, too, must be a multiple of the page size. The third parameter, *prot*, determines the protection for the mapped file. It can be marked readable, writable, executable, or some combination of these. The fourth parameter, *flags*, controls whether the file is private or sharable, and whether *addr* is a requirement or merely a hint. The fifth parameter, *fd*, is the file descriptor for

the file to be mapped. Only open files can be mapped, so to map a file in, it must first be opened. Finally, *offset* tells where in the file to begin the mapping. It is not necessary to start the mapping at byte 0; any page boundary will do.

The other call, *unmap*, removes a mapped file. If only a portion of the file is unmapped, the rest remains mapped.

The figure shown below illustrates the layout of a C program in memory, highlighting how the different sections of a process relate to an actual C program. This figure is similar to the general concept of a process in memory as shown in Figure 3.1, with a few differences:

- The global data section is divided into different sections for (a) initialized data and (b) uninitialized data.
- A separate section is provided for the *argc* and *argv* parameters passed to the *main()* function.



The GNU *size* command can be used to determine the size (in bytes) of some of these sections. Assuming the name of the executable file of the above C program is *memory*, the following is the output generated by entering the command *size memory*:

text	data	bss	dec	hex	filename
1158	284	8	1450	5aa	memory

The *data* field refers to uninitialized data, and *bss* refers to initialized data. (*bss* is a historical term referring to *block started by symbol*.) The *dec* and *hex* values are the sum of the three sections represented in decimal and hexadecimal, respectively.