Operating Systems 2
Process Address Space

Arkadiusz Chrobot

Department of Information Systems

June 11, 2024

Outline

Introduction

The Linux kernel is responsible for managing its own *address space*, as well as the address spaces of user-space processes. Each of them is given a *flat* (*linear*) address space which by default is separated from the address spaces of other processes. It means that a process cannot read or modify data of other processes, even if it uses the same virtual addresses as them. However, Linux kernel makes it possible for the processes to *share* their address spaces, if it is requested. That's how it implements user-space *threads*. Each address space of a process is partitioned into *intervals* of addresses known as *memory areas* (see Figure 1). Every process can ask the kernel to add a new memory area to its address space, but it should not reference non-existent areas or violate the permissions (write, read or execute) of its own memory areas. Otherwise, the kernel will abort such a process and its user will see the "Segmentation Fault" message on the screen.
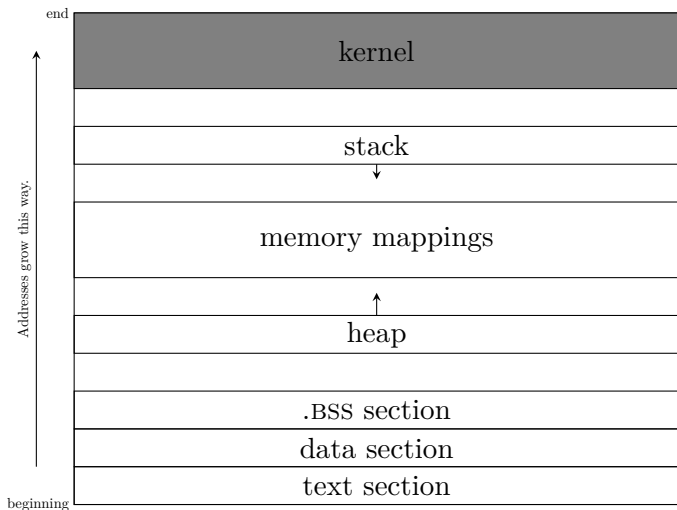
Introduction

Figure 1: Conceptual Model of User Process ▸ Address Space in Linux

Process Address Space Organization
Memory Areas

There are several types of memory areas:

text section is a memory area where the part of an executable file that contains process code is mapped,

data section is a mapping of the part of an executable file that contains process initialized global variables,

.BSS section is a memory area where the zero page is mapped; this area contains uninitialized global variables,

stack is an area for the process user-space stack; initially the zero page is mapped here,

memory mapped files files that have been mapped onto memory,

shared memory segments memory areas that implement shared memory,

anonymous memory mappings memory areas allocated for example with the use of the `malloc()` function.

Process Address Space
Memory Areas

The .BSS abbreviation stands for "block started by symbol" and it is a name used for historical reasons. The data section contains global variables with initial values other than zero. These values are stored in an executable file. That's why they are called "initialized". The .BSS section contains global variables whose initial value is zero. These values are not stored in the executable file, hence these variables are called "uninitialized".

The text sections, data sections and .BSS sections are also used by *shared libraries*, often called *shared objects* or *dynamically loaded libraries*. With each memory area is associated a separate set of permissions. Memory areas don't overlap.

Memory Descriptor

Information about an address space of a single process is stored in its *memory descriptor*. It is a structure of the struct mm_struct type, which is defined in the same file as the process descriptor type. The memory descriptor has many members. Among them are fields that store the start and end addresses of the text section, the data section, the stack, the memory area that stores command line arguments and the memory area that stores environmental variables. If the value of the mm_count field is 1 then the address space specified by the memory descriptor is shared by at least two processes which are threads. The exact number of these threads is stored in the mm_users field. The task_size field defines the address space size. It has been added to the kernel to allow the 32-bit processes to run in 64-bit systems. Two fields of the memory descriptor are associated with data structures that store the same data but in different ways. The first one is the mmap field which stores an address of a list that stores data about all memory areas. The second one is called mm_rb and stores an address of a red-black tree root.

Memory Descriptor

The tree stores the same data as the list, but searching in such a tree is quicker than searching the list. On the other hand the list is a simpler data structure for traversing it sequentially. In the 2.0 Linux kernel series the data about memory areas were stored in the list, as long as the number of these areas was less than 20. If it had exceeded this limit then the data would have been reorganized into AVL tree.

The kernel links all memory descriptors into a doubly linked list, starting with the memory descriptor of the init (or its equivalent) process. Also, the address of a memory descriptor is stored in the mm field of the descriptor of the process, that owns the address space specified by the memory descriptor. When a process forks, then a new memory descriptor is allocated to its child with the use of the allocate_mm macro and then the content of its memory descriptor is copied to the memory descriptor of the child with the help of the copy_mm() function. The allocation of the memory descriptor is performed by the slab allocator.

Memory Descriptor

If the `clone()` system call gets the `CLONE_VM` flag as one of its arguments, then the new process will share the address space with its parent. In other words these processes will be threads. In this case no memory descriptor is allocated for the new process. Both of them will share the same memory descriptor.

When a process or a thread exits then the `exit_mm()` function is invoked that updates some statistics, performs some cleanup operations and calls the `mmput()` function that decrements the value of the `mm_users` field in the memory descriptor. If the value reaches zero, then the `mmdrop()` function is called, which decrements the value of the `mm_count` field in the memory descriptor. If the value of the latter field reaches zero too, then the memory descriptor is deallocated with the use of the `free_mm()` function.

Memory Descriptor

The kernel-space threads or simply kernel threads do not have their own address space, they share it with the kernel. Therefore they also do not have memory descriptors. The value of their process descriptor mm field is NULL. However, kernel threads have to access memory to run, so they use the memory descriptors of user-space processes that were using the CPU before them. The memory descriptor of each process stores data about the kernel address space for the needs of system calls. This information is the same for all processes, but since the release of the 4.15 kernel version different page tables are used in the kernel mode and in the user mode. This change has been introduced by the KPTI patch to mitigate the Meltdown vulnerability (see: https://meltdownattack.com/). The address of the recently scheduled user-space process memory descriptor is stored in the active_mm field of the kernel thread process descriptor. In case of regular user processes the kernel uses this field when the process begins to run a different program — a different code loaded from the executable file.

Virtual Memory Areas Management

The subsystem that manages memory areas or more precisely *virtual memory areas* (VMA) has been developed with the use of object-oriented techniques. Each virtual memory area is represented by an object which is a structure of the `vm_area_struct` type. Aside from "regular" fields this structure has a member which is a pointer to a structure whose fields are pointers to functions that perform some operations on the virtual memory ares. In other words these functions are methods and the latter structure is a method table. The `vma_start` and `vma_end` fields of the virtual memory area object store the start and end address of the memory area. The `vm_flags` field stores flags that specify the properties and purpose of pages that form the virtual memory area. Among these flags are: `VM_READ`, `VM_WRITE`, `VM_EXEC` — specify memory areas that can be read, written or executed, `VM_SHARED` — denotes a memory area that is shared, `VM_IO` — indicates a memory area where the input/output registers of a device are mapped, `VM_LOCKED` — specifies memory area which pages cannot be swapped, `VM_SEQ_READ` — specifies a memory

Virtual Memory Areas Management

area where a file is mapped that offers only sequential reads, so the kernel can read some of its data in advance, i.e. before a user-process requests the data, to increase the efficiency of the file read operation, `VM_RAND_READ` — denotes a memory area where a file is mapped that offers both sequential and random access, so reading its data in advance doesn't bring any benefits. The object method table is a structure of the `vm_operations_struct` type. This structure has several members that point to functions performing operations on a virtual memory area. Among these functions are: `open()` — the function is invoked when a new virtual memory area is added to the process address space, `close()` — the function is invoked when a virtual memory area is removed form the process address space, `fault()` — this function is called when the page fault exception is raised, and the page exists, but is not present in the RAM, `page_mkwrite()` — it is called when the page fault exception is raised and a read-only page changes to writable, `access()` — the function is called when some exceptions are raised while an

Virtual Memory Areas Management

address space of a specific process is being accessed. In earlier kernel versions the `populate()` function was available that was latter removed. The `fault()` function replaced the `nopages()` function. As it was mentioned earlier, virtual memory area objects are linked into a list and a red-black tree. The tree is used by the `find_vma()` kernel function, that either finds a memory area, which contains an address given to the function as its argument, or an area that starts with a greater address. If it fails to find such an area it returns NULL, otherwise it returns the address of an object associated with the virtual memory area. Similarly, the `find_vma_prev()` function finds an area that is located before the address that is an argument of this function. Finally, the `find_vma_intersection()` function returns an address of the object that specifies a virtual memory area which at least partially overlaps an address interval formed by two addressed that are arguments of this function.

Virtual Memory Areas Management

The information about all virtual memory areas of a given process is stored in the /proc/<pid>/maps file, where the <pid> is the PID of the process. The same data can be displayed on the screen in a more human-readable form with the use of the pmap command. The information reveals that text sections and read-only data sections can be shared by processes as well as shared libraries.

A virtual memory area can be expanded or a new virtual memory area can be created with the help of the do_mmap() function. Its primary purpose is to map a file onto the memory. It takes, as one of its arguments, an address of a file object. However, if a NULL value is given instead of that address, then the function will perform an anonymous mapping, i.e. a zero page will be mapped onto a specified memory area. This function is invoked by the mmap2() and mmap() system calls. The former requires that the offset in the mapped file is expressed in pages, not in bytes. The virtual memory area can be deleted with the help of the do_munmap() function, invoked by the munmap() system call.

Questions

?

The End

Thank You for Your attention!