Operating Systems 2

Process Address Space

Arkadiusz Chrobot

Department of Computer Science

June 1, 2020

Outline

1. Introduction
2. Process Address Space Organization
3. Memory Descriptor
4. Virtual Memory Areas Management

Introduction

The Linux kernel manages 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 any process cannot read or modify data of other processes, even if it uses the same addresses as them. However, Linux kernel makes it possible for some of the processes to *share* their address spaces. That's how it implements user-space *threads*. Each address space of a process is partitioned into *intervals* of addresses knowns as *memory areas*. Every process can ask the kernel to add a new memory area to its address space, but it cannot reference areas that are not its own 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.

Process Address Space Organization

Memory Areas

There are several types of memory areas:

text section it is a mapping of the part of executable file that contains process code onto the memory,

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

.bss section it is a memory area where the zero page is mapped, possibly multiple times; the area contains uninitialized global variables,

stack this is an area for the process user-space stack; initially the zero page is mapped here (usually multiple times),

memory mapped files files that have been mapped onto memory,

shared memory segments memory areas that enable 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 means "by symbol started" and it is a name used for historical reasons. The data section contains global variables with initial values other than zero. Those values are stored in executable files. That's why they are called "initialized". The .bss section contains global variables that initial value is zero. Those values are not stored in the executable file, hence those variables are called "uninitialized".
The text sections, data sections and .bss sections are also used by *shared libraries* also called *shared objects*. 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 leas two processes which are threads. The exact number of those threads is store 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 applications run on 64-bit hardware platforms. Two fields of the memory descriptor are associated with data structures that store the same information but in different ways. The first one is the `mmap` field which stores an address of list that stores data about all memory areas. The second one is called `mm_rb` and stores an address of the 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 sequential traversal is simpler in case of the list. In the Linux kernel 2.0 series the data about memory areas were store in a list as long as the number of those areas was less than 20. If it had exited 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 got the `CLONE_VM` flat as its argument, then the new process will share the address space with its parent. In other words those processes will be threads. In this case no memory descriptor is allocated for this 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 activities 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 then value of the `mm_count` filed in the memory descriptor. If the value of the 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` fields 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 information about 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 Spectre and Meltdown vulnerabilities (see: https://meltdownattack.com/). The address of last 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 the fields 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 which fields are pointers to functions that perform some operations on the virtual memory ares. In other words those 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 behaviour of pages that are part of the virtual memory area. Among those 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 ares where the input/output registers of a device are mapped, `VM_LOCKED` — specifies memory area which pages are not swapped, `VM_SEQ_READ` — specifies

Virtual Memory Areas Management

a memory area where a file is mapped that offers only sequential read, so the kernel can read some of its data in advance, i.e. before the user-process request 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`. This structure has several members that point to functions performing operations on virtual memory area. Among those 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 risen, and the page exists, but is not present in the memory, `page_mkwrite()` — it is also called when the page fault exception is risen, but when then read-only page changes to writable, `access()` — the function is called when some exceptions

Virtual Memory Areas Management

are risen while the 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 earlier mentioned, 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 finds a memory area, which contains 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 the virtual memory area object. Similarly the `find_vma_prev()` function finds an area that is located before the address that is this function argument. Finally, the `find_vma_intersection()` function returns an address of the object that specifies a virtual memory area that at least partially overlaps the address interval formed by two addressed that are arguments of this function.

Virtual Memory Areas Management

The data about all virtual memory areas of a given process are stored in the `/proc/<pid>/maps` file, where the `<pid>` denotes the PID of the process. The same information can be displayed on the screen in more human readable form with the use of the `pmap` command. The data reveal 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 job is to map a file onto the memory. However, if the `NULL` value is given as one of its arguments, instead of an address of a file object, then the function will perform an anonymous mapping, i.e. a zero page will be mapped in this area. This function is invoked by the `mmap2()` and `mmap()` system calls. The former requires that the offset in the mapped file is specified in the size of page units not in bytes. The virtual memory area can be deleted is `do_munmap()` function, invoked by the `munmap()` system call.

Questions

?

The End

Thank You for Your attention!