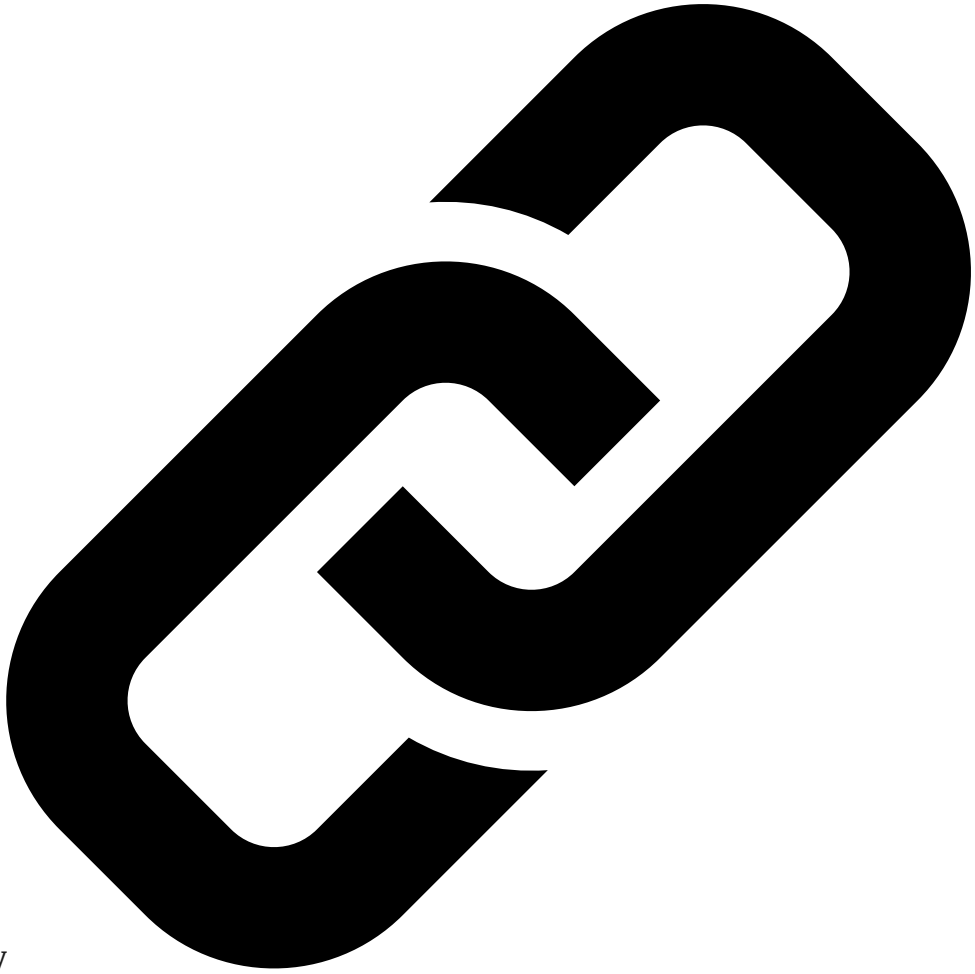
Understanding the Linux /proc/id/maps File | Baeldung on Linux

By baeldung

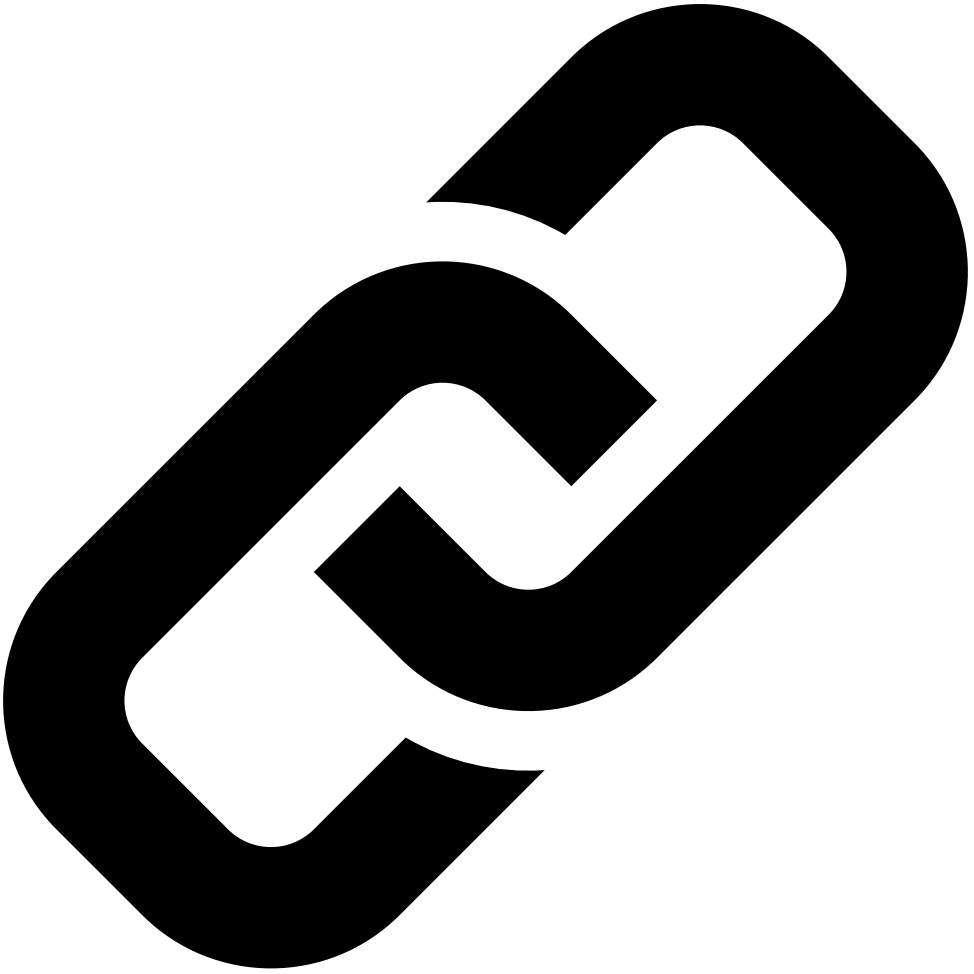
Published: 2022-03-14 · Archived: 2026-04-05 18:11:59 UTC



1. Overview

In this tutorial, we'll see how to profile the memory usage of a Linux process by **reading the output of the *<u>/proc/id/maps</u>* file**. We'll start by **explaining the concept of virtual memory**. Next, we'll **describe the virtual address space of a process**, **its structure, and the permissions around it**. Finally, we'll go over how to interpret the output of */proc/id/maps* to view the virtual address space of a particular process.

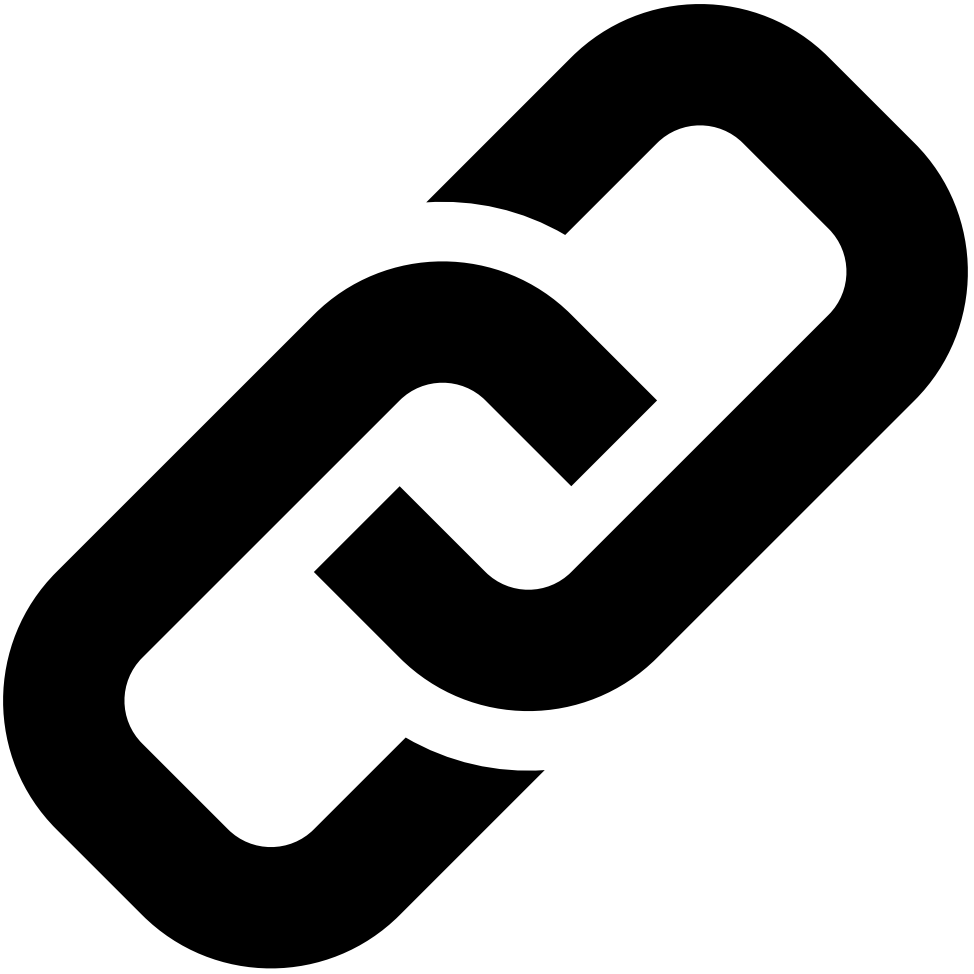
2. What Is Virtual Memory?



Virtual memory allows each process to claim all of the available physical memory that is present on the machine. In other words, each process acts as if it is the only process running in the operating system. This approach has many benefits. For instance, it offers a much better developer experience, as it greatly simplifies the new memory requests. Moreover, it improves security, as it isolates processes so they can't interfere with each other. Lastly, it increases the performance because when an error occurs, it only affects a single process without adding overhead to the rest of the processes.

However, **special care needs to be taken if the total amount of virtual memory exceeds the actual memory that's available** (including the swap space). To solve that problem, Linux has introduced the out-of-memory killer.

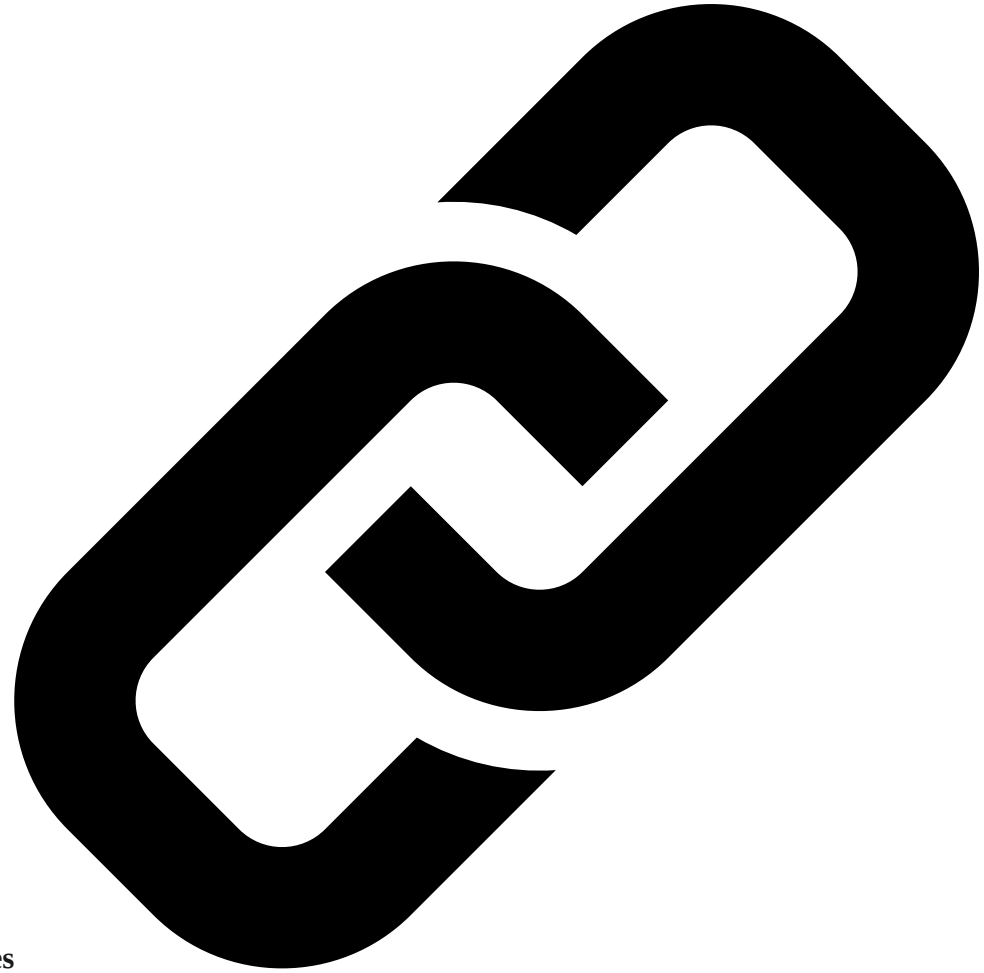
3. Virtual Address Space (VAS) of a Process



The **virtual address space (VAS) of a process consists of all the available memory addresses that this process can refer to**. This is the virtual memory, which is approximately equal to the total physical memory that's installed on the machine.

The VAS is split into two regions:

- Kernel space virtual addresses are referred to by processes that switch from user mode to kernel mode
- User space virtual addresses contain different types of segments that contain the processes' code, data, and dependencies

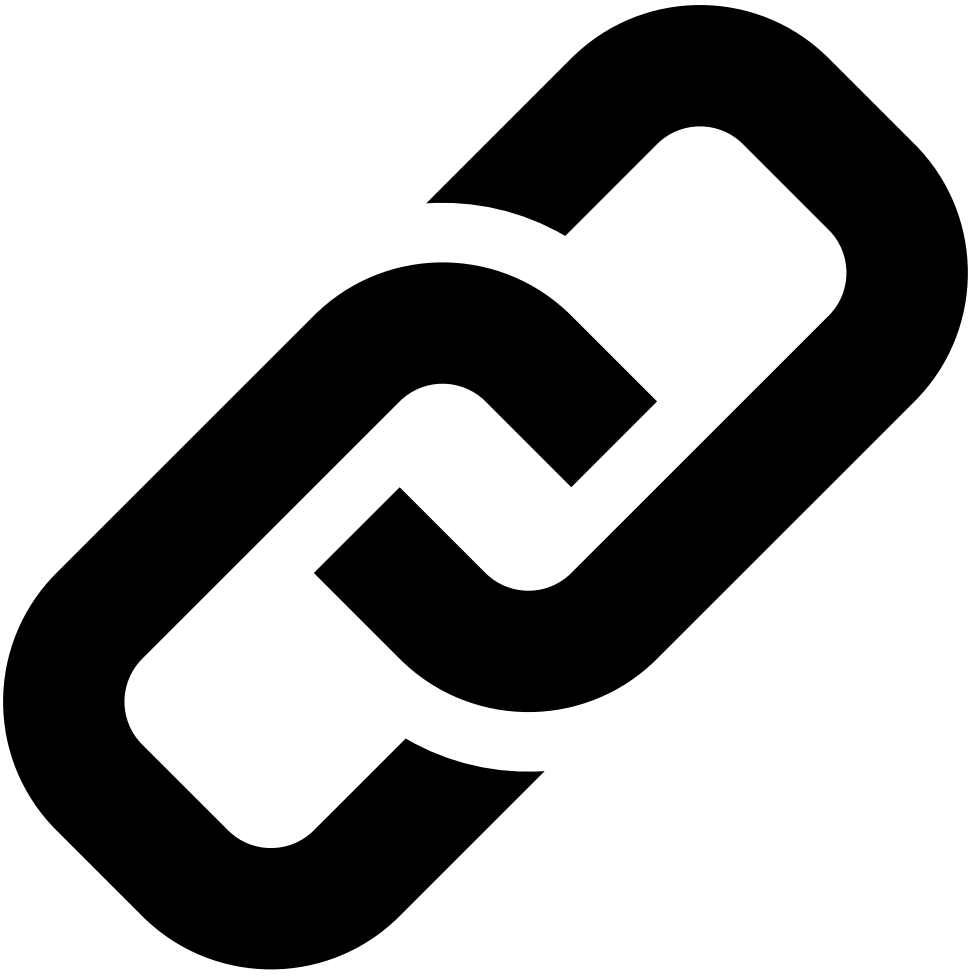


3.1. Segment Types

The user **VAS segments, also known as mappings, are contiguous blocks of memory** whose content depends on the segment type. Let’s look at the different types:

- Code segments (or text segment), which contain executable code
- Data segments that contain process data such as variables and constants. **They break further down into initialized data segments, uninitialized data segments**, **and heap segments**
- Stack segments can be expanded dynamically and contain the function parameters and local function variables
- Shared libraries segments, which contain linked (shared) libraries that the process uses

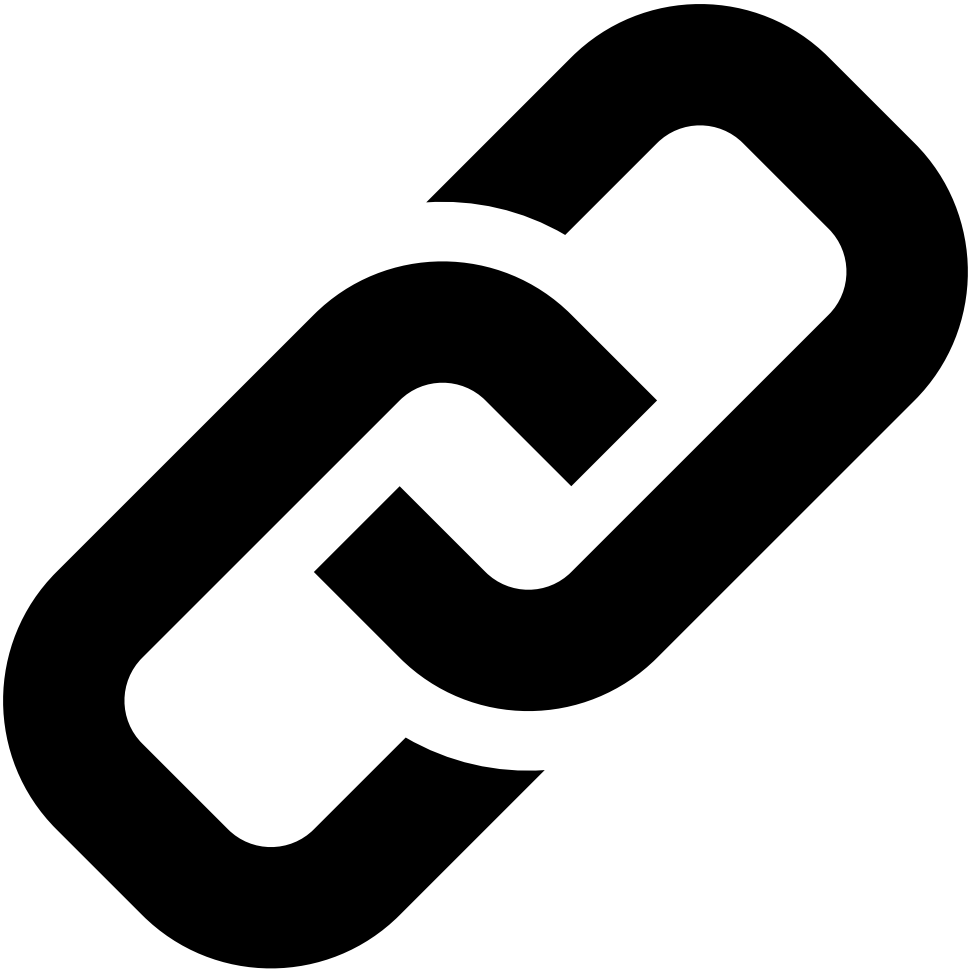
3.2. Segment Permissions



Those **segments are mapped with a set of permissions that control which actions are allowed on them**. These permissions, which are often referred to as modes, are the following:

- read-only (r) means that the segment is readable, hence all segments usually have that mode
- read-write (w) means that the segment is readable and writeable to allow for data modification
- execute (x) means that the segment contains executable code
- private (p), which means that the segment is private, thus only visible from that process
- shared (s), which means that multiple (at least 2) processes share that segment

4. Examining the VAS of a Process With */proc/id/maps*

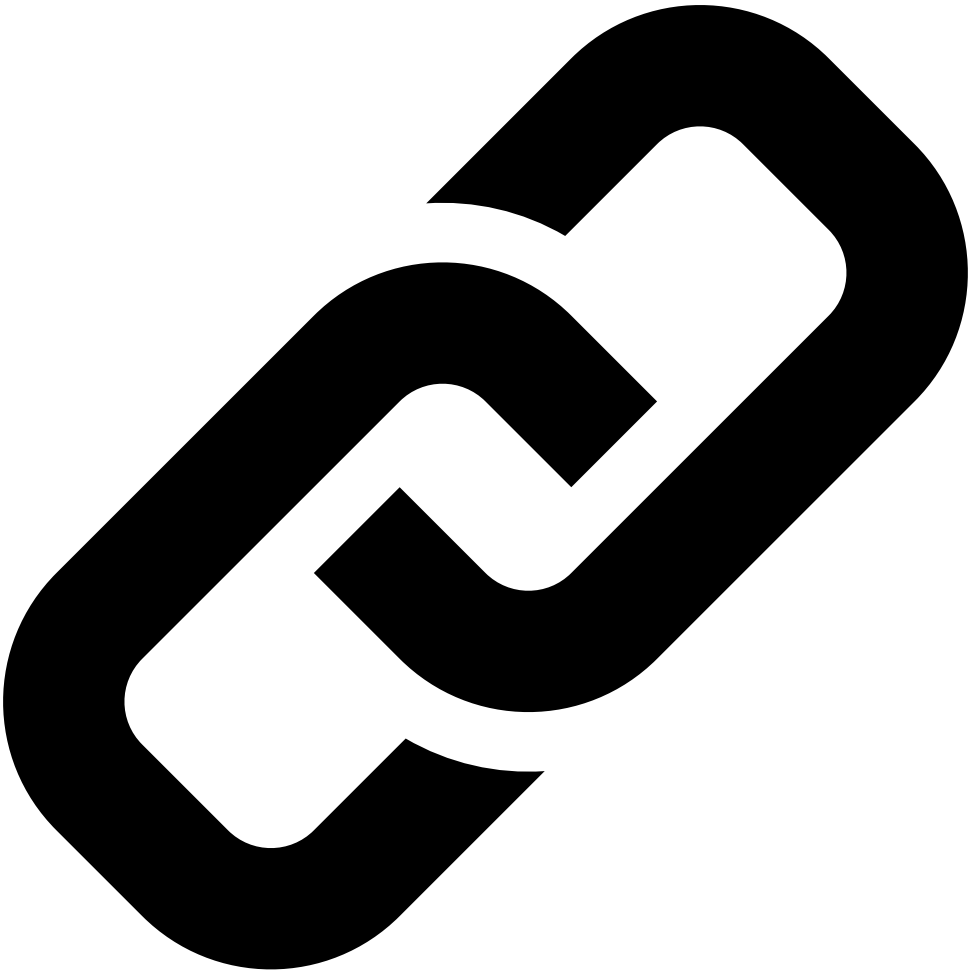


Let's now apply what we learned in the previous sections to understand the VAS of a process. Assuming that we know the PID of the process, we can leverage procfs to see what the VAS looks like. To do that, **we can read from the */proc/<pid>/maps* file**. Here's an example output, using the *cat* command:

```
 $ cat /proc/self/maps
559b8c418000-559b8c41a000 r--p 00000000 08:30 1708                       /usr/bin/cat
559b8c41a000-559b8c41f000 r-xp 00002000 08:30 1708                       /usr/bin/cat
559b8c41f000-559b8c422000 r--p 00007000 08:30 1708                       /usr/bin/cat
559b8c422000-559b8c423000 r--p 00009000 08:30 1708                       /usr/bin/cat
559b8c423000-559b8c424000 rw-p 0000a000 08:30 1708                       /usr/bin/cat
559b8c5d1000-559b8c5f2000 rw-p 00000000 00:00 0                          [heap]
7faa72001000-7faa72023000 rw-p 00000000 00:00 0
7faa72023000-7faa72055000 r--p 00000000 08:30 3023                       /usr/lib/locale/C.UTF-8/LC_CTYPE
7faa72055000-7faa72056000 r--p 00000000 08:30 3030                       /usr/lib/locale/C.UTF-8/LC_NUMERIC
7faa72056000-7faa72057000 r--p 00000000 08:30 3033                       /usr/lib/locale/C.UTF-8/LC_TIME
7faa72057000-7faa721ca000 r--p 00000000 08:30 3022                       /usr/lib/locale/C.UTF-8/LC_COLLATE
7faa721ca000-7faa721cb000 r--p 00000000 08:30 3028                       /usr/lib/locale/C.UTF-8/LC_MONETARY
7faa721cb000-7faa721cc000 r--p 00000000 08:30 3027                       /usr/lib/locale/C.UTF-8/LC_MESSAGES/SYS
7faa721cc000-7faa721cd000 r--p 00000000 08:30 3031                       /usr/lib/locale/C.UTF-8/LC_PAPER
7faa721cd000-7faa721ce000 r--p 00000000 08:30 3029                       /usr/lib/locale/C.UTF-8/LC_NAME
```

```
7faa721ce000-7faa721cf000 r--p 00000000 08:30 3021                       /usr/lib/locale/C.UTF-8/LC_ADDRESS
7faa721cf000-7faa724b5000 r--p 00000000 08:30 3034                       /usr/lib/locale/locale-archive
7faa724b5000-7faa724da000 r--p 00000000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa724da000-7faa72652000 r-xp 00025000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa72652000-7faa7269c000 r--p 0019d000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa7269c000-7faa7269d000 ---p 001e7000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa7269d000-7faa726a0000 r--p 001e7000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa726a0000-7faa726a3000 rw-p 001ea000 08:30 11971                      /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa726a3000-7faa726a9000 rw-p 00000000 00:00 0
7faa726a9000-7faa726aa000 r--p 00000000 08:30 3032                       /usr/lib/locale/C.UTF-8/LC_TELEPHONE
7faa726aa000-7faa726ab000 r--p 00000000 08:30 3025                       /usr/lib/locale/C.UTF-8/LC_MEASUREMENT
7faa726ab000-7faa726b2000 r--s 00000000 08:30 11818                      /usr/lib/x86_64-linux-gnu/gconv/gconv-m
7faa726b2000-7faa726b3000 r--p 00000000 08:30 11854                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7faa726b3000-7faa726d6000 r-xp 00001000 08:30 11854                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7faa726d6000-7faa726de000 r--p 00024000 08:30 11854                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7faa726de000-7faa726df000 r--p 00000000 08:30 3024                       /usr/lib/locale/C.UTF-8/LC_IDENTIFICATI
7faa726df000-7faa726e0000 r--p 0002c000 08:30 11854                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7faa726e0000-7faa726e1000 rw-p 0002d000 08:30 11854                      /usr/lib/x86_64-linux-gnu/ld-2.31.so
7faa726e1000-7faa726e2000 rw-p 00000000 00:00 0
7ffeb2f53000-7ffeb2f74000 rw-p 00000000 00:00 0                          [stack]
7ffeb2f99000-7ffeb2f9d000 r--p 00000000 00:00 0                          [vvar]
7ffeb2f9d000-7ffeb2f9e000 r-xp 00000000 00:00 0                          [vdso]
```

4.1. Understanding the Output



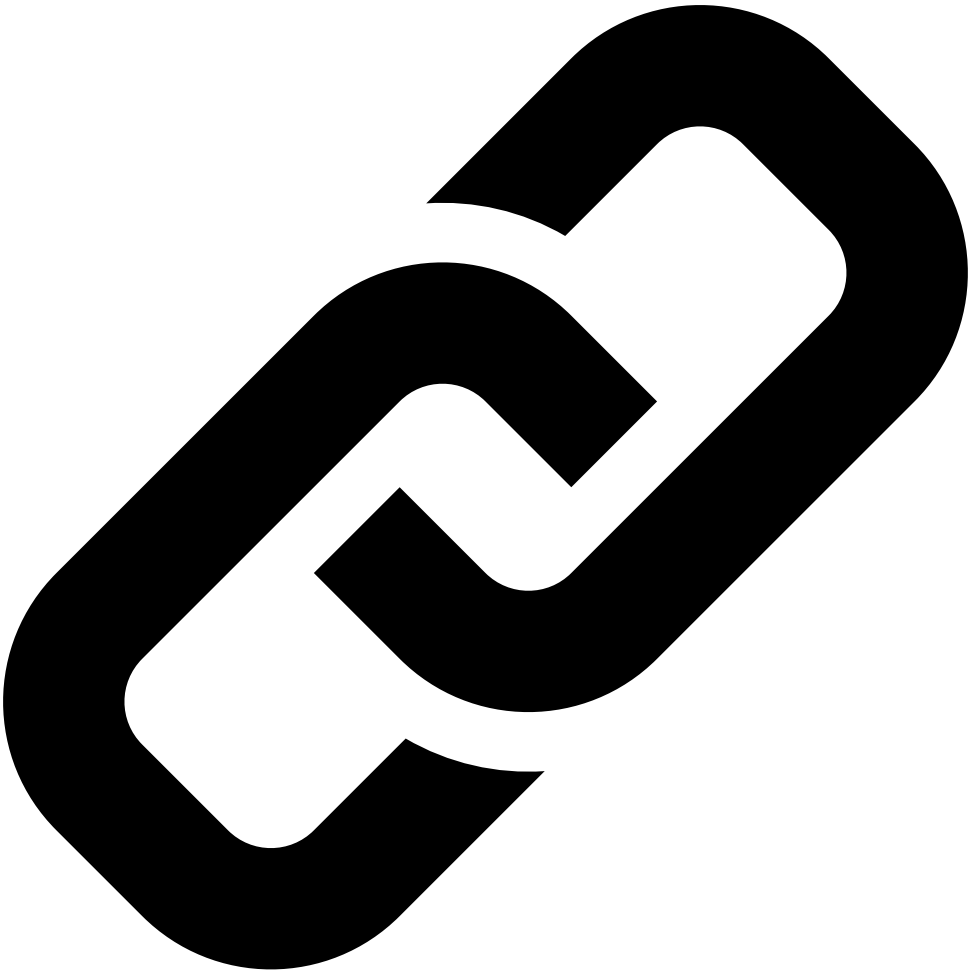
In the above output, we can see the user VAS of the command line utility *cat*. We can see that there are six columns in total. Let's look at the first line and describe what each column is about:

```
<address start>-<address end>  <mode>  <offset>   <major id:minor id>   <inode id>   <file path>
559b8c418000-559b8c41a000      r--p    00000000          08:30               1708     /usr/bin/cat
```

- **address start – address end is the start and end address of that mapping**. Note that **the whole output is sorted based on those addresses**, from low to high.
- **mode (permissions) specifies which actions are available on this mapping** and if it's private or shared.
- **offset is the start offset in bytes within the file that is mapped**. This **only makes sense for file mappings**. For instance, stack or heap mappings are examples of mappings that aren't files, and in those cases, the offset is 0. In the above example, the mapping is of the */usr/bin/cat* file, and the offset is 0.
- **major:minor ids represent the device that the mapped file lives** in the form of a major and minor id. In the above example, 08:30 represents the major and minor id of the hard drive that has the root filesystem. For non-file mappings, this column shows 00:00.
- **inode id of the mapped file** (again, that's only valid for file mappings). Inodes **are data structures that contain the core filesystem-related metadata**. When it comes to non-file mappings, this field is set to 0. In our example, this id is 1708.

- The **file path of the file for that mapping**. In the event that this is not a file mapping, that field is empty.

4.2. Examining the Output Line by Line



We're now ready to examine the output line by line. To keep this short, we are only going to describe a representative example for every different type of segment that we see in the output. Let's jump right in:

Firstly, **we see a read-only, private mapping for the *`/usr/bin/cat`* file**:

```
559b8c418000-559b8c41a000 r--p 00000000 08:30 1708 /usr/bin/cat
```

This is a private data segment, which most probably contains global variables or constants and is therefore read-only.

Secondly, we see a readable and executable mapping, again for */usr/bin/cat:*

```
559b8c41a000-559b8c41f000 r-xp 00002000 08:30 1708 /usr/bin/cat
```

Given that it is executable, this segment is the code segment.

Thirdly, **we see a readable and writable mapping**:

```
559b8c423000-559b8c424000 rw-p 0000a000 08:30 1708 /usr/bin/cat
```

Again, this is a data segment, most probably for the process to store and update variables.

Further, **we can see a mapping that corresponds to the heap segment**:

```
559b8c5d1000-559b8c5f2000 rw-p 00000000 00:00 0 [heap]
```

Note that this is not a file mapping, so the major, minor, and inode ids are 0.

Next, **we see a read-only private mapping**:

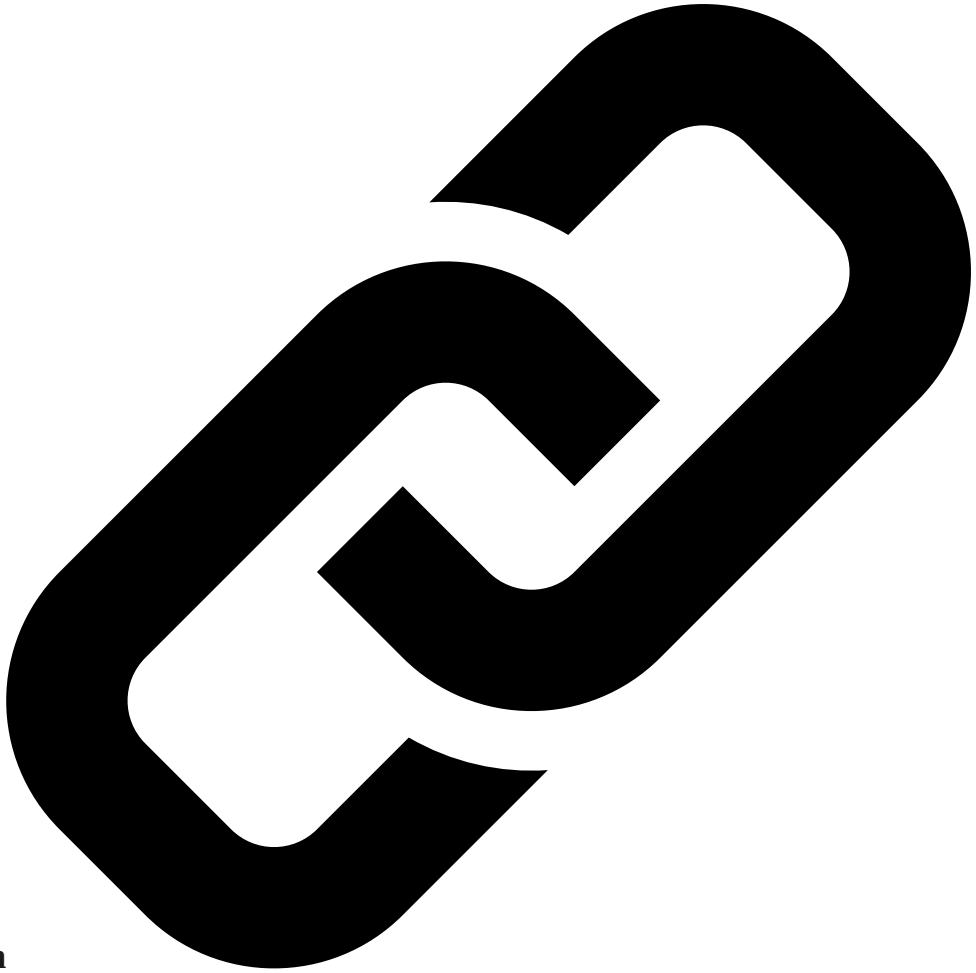
```
7faa72023000-7faa72055000 r--p 00000000 08:30 3023 /usr/lib/locale/C.UTF-8/LC_CTYPE
```

This is a (read-only) data segment that contains locale-related constants.

Lastly, **we can see read-only data segments and code segments for libc**:

```
 7faa724b5000-7faa724da000 r--p 00000000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa724da000-7faa72652000 r-xp 00025000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa72652000-7faa7269c000 r--p 0019d000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa7269c000-7faa7269d000 ---p 001e7000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa7269d000-7faa726a0000 r--p 001e7000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
7faa726a0000-7faa726a3000 rw-p 001ea000 08:30 11971 /usr/lib/x86_64-linux-gnu/libc-2.31.so
```

This is a shared library used by the cat process.



5. Conclusion

In this article, we saw how to examine the user virtual address space (VAS) of a Linux process. We started by briefly explaining what virtual memory is. Next, we described the user VAS of a process and its segments. Then, we explained how to dive deep into the VAS by reading the */proc/id/maps* file. Finally, we saw a line-by-line example of the output from that file for the *cat* command-line utility.
