CHAPTER 10

Memory Management

Learning Objectives

1. Students understand the concepts of virtual memory in Linux.
2. Students able to programmatically allocate memory to their process.

Introduction

To accommodate many processes in main memory, operating systems use virtual memory to address the limitation of physical memory size. This limitation is addressed by using a disk as an extension of physical memory such that more processes can enter the physical memory to be executed. Using virtual memory, a process can have a portion of its memory needs in the physical memory and another portion in secondary storage. In Linux, the part of the memory located in the secondary storage is called the *swap space*. Linux operating systems allow users to define the swap space as either a dedicated storage partition or a file within the file system.

In Linux operating systems, each process is allocated its own virtual address space (VAS). This VAS contains all available memory addresses to which a process can refer. The VAS is split into kernel space and user space virtual addresses. Virtual addresses in the kernel space are referred to by processes during the switch between the user mode and the kernel mode. The virtual addresses of the user space contain several types of segments containing the code (or text) and data of the processes. Data segments can expand depending on the execution of the process. These data segments consists of heap and stack segments. The stack data segment is used for local variables that are defined without any allocation specification. The heap data segment is used for data that is dynamically allocated by the program. Figure 1 shows the virtual address space of a process.

In this lab work, we will look at virtual memory usage in Linux operating systems. Then, we will see the virtual address space of a process. Finally, we will write some programs to dynamically allocate memory for a variable in the programs.

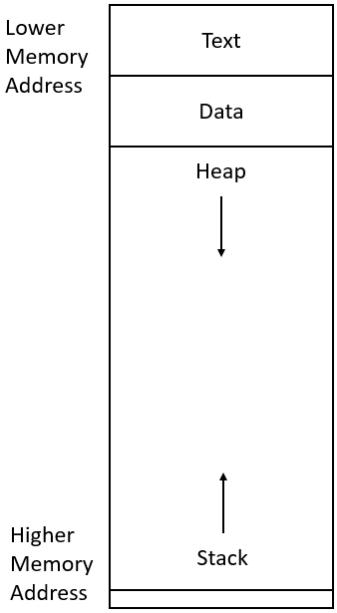

Figure 1 Virtual Address Space of a Process

10.1 Displaying Virtual Memory Usage

In Linux operating systems, the current usage of virtual memory can be displayed using the “top” command.

```
$ top
```

This command shows all processes running in the operating system, including their virtual memory usage.

```
top - 11:10:09 up  1:03,  1 user,  load average: 0.06, 0.17, 0.09
Tasks: 101 total,   2 running,  99 sleeping,   0 stopped,   0 zombie
%Cpu(s):  0.7 us,  0.7 sy,  0.0 ni, 98.6 id,  0.0 wa,  0.0 hi,  0.0 si,  0.0
st
MiB Mem :    976.9 total,     80.6 free,    502.3 used,    394.0 buff/cache
MiB Swap:      0.0 total,      0.0 free,      0.0 used.    324.1 avail Mem

    PID USER      PR  NI    VIRT    RES    SHR S  %CPU  %MEM     TIME+
COMMAND
    740 mysql     20   0 1338996 372808  13920 S   1.7  37.3   0:47.43 mysqld
    905 ubuntu    20   0   13928   4056   2592 S   0.3   0.4   0:06.84 sshd
   1475 ubuntu    20   0   11012   3740   3192 R   0.3   0.4   0:00.75 top
```

The above result shows that some processes are currently running in the operating system. Let us focus on the three columns that represent the virtual memory status of each process, namely VIRT, RES, and SHR.

1. **VIRT** denotes the virtual size of a process comprising the size of memory it is currently using, the memory it has mapped to itself (e.g., video card for the OS window manager), files on the disk that have been mapped to it (e.g., shared library), and the memory shared with other processes.
2. **RES** denotes the resident set size that represents the actual size of physical memory a process is currently using. This column corresponds to the %MEM column. The value in this column will virtually be less than the VIRT.
3. **SHR** denotes the size of sharable memory or libraries of a process. For libraries, some parts of the library that are currently used will be resident and counted towards RES, while the whole library will be counted in VIRT and SHR.

In the above result, the "**mysqld**" process has a virtual memory size of 1.3 GB in which around 370 MB of the process reside in physical memory. This usage occupies approximately 37.3% of the physical memory that has a size of 976 MB.

Another way to see the memory status of a Linux operating system is by using the "**free**" command. This command displays the amount of free and used memory in the system.

```
$ free -h -t
              total        used        free      shared  buff/cache
available
Mem:          976Mi       498Mi        79Mi       3.0Mi       397Mi
327Mi
Swap:            0B          0B          0B
Total:        976Mi       498Mi        79Mi
```

The above result shows the result of the invocation of the "free" command. We use the flag "-h" to display the size in the closest unit, in this case, Megabytes. The flag "-t" provides the total amount at the end of the line. The above result indicates that the system's physical memory size is 976 MB and 498 MB among them is being used.

10.2 Displaying Virtual Address Space of a Process

To illustrate the virtual memory allocated by a Linux operating system to a process, we will use the "**sleep**" command and see the virtual memory space for the process. First, let us execute the "sleep" command and pass a considerable amount of sleep time as a background process.

```
$ sleep 1000 &
[1] 2608
$ ps
    PID TTY          TIME CMD
```

```
    907 pts/0    00:00:00 bash
   2608 pts/0    00:00:00 sleep
   2609 pts/0    00:00:00 ps
```

Using the ps command as shown above, we will obtain the process id (PID) of the sleep process, namely 2608. Then, let us see the memory mapping file of this process. This file is located in the **/proc/<pid>/maps**.

```
$ cat /proc/2608/maps
55fb884a1000-55fb884a3000 r--p 00000000 08:01 1642        /usr/bin/sleep
55fb884a3000-55fb884a7000 r-xp 00002000 08:01 1642        /usr/bin/sleep
55fb884a7000-55fb884a9000 r--p 00006000 08:01 1642        /usr/bin/sleep
55fb884aa000-55fb884ab000 r--p 00008000 08:01 1642        /usr/bin/sleep
55fb884ab000-55fb884ac000 rw-p 00009000 08:01 1642        /usr/bin/sleep
55fb88fb5000-55fb88fd6000 rw-p 00000000 00:00 0           [heap]
7fe7c5682000-7fe7c56b4000 r--p 00000000 08:01 6081        /usr/lib/locale/C.UTF-8/LC_CTYPE
7fe7c56b4000-7fe7c56b5000 r--p 00000000 08:01 6086        /usr/lib/locale/C.UTF-8/LC_NUMERIC
7fe7c56b5000-7fe7c56b6000 r--p 00000000 08:01 6089        /usr/lib/locale/C.UTF-8/LC_TIME
7fe7c56b6000-7fe7c5829000 r--p 00000000 08:01 6080        /usr/lib/locale/C.UTF-8/LC_COLLATE
7fe7c5829000-7fe7c582a000 r--p 00000000 08:01 6084        /usr/lib/locale/C.UTF-8/LC_MONETARY
7fe7c582a000-7fe7c582b000 r--p 00000000 08:01 6078        /usr/lib/locale/C.UTF-
8/LC_MESSAGES/SYS_LC_MESSAGES
7fe7c582b000-7fe7c582c000 r--p 00000000 08:01 6087        /usr/lib/locale/C.UTF-8/LC_PAPER
7fe7c582c000-7fe7c582d000 r--p 00000000 08:01 6085        /usr/lib/locale/C.UTF-8/LC_NAME
7fe7c582d000-7fe7c582e000 r--p 00000000 08:01 6079        /usr/lib/locale/C.UTF-8/LC_ADDRESS
7fe7c582e000-7fe7c582f000 r--p 00000000 08:01 6088        /usr/lib/locale/C.UTF-8/LC_TELEPHONE
7fe7c582f000-7fe7c5b15000 r--p 00000000 08:01 6075        /usr/lib/locale/locale-archive
7fe7c5b15000-7fe7c5b37000 r--p 00000000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
7fe7c5b37000-7fe7c5caf000 r-xp 00022000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
7fe7c5caf000-7fe7c5cfd000 r--p 0019a000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
7fe7c5cfd000-7fe7c5d01000 r--p 001e7000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
7fe7c5d01000-7fe7c5d03000 rw-p 001eb000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
7fe7c5d03000-7fe7c5d09000 rw-p 00000000 00:00 0
7fe7c5d09000-7fe7c5d0a000 r--p 00000000 08:01 6083        /usr/lib/locale/C.UTF-
8/LC_MEASUREMENT
7fe7c5d0a000-7fe7c5d11000 r--s 00000000 08:01 3729        /usr/lib/x86_64-linux-
gnu/gconv/gconv-modules.cache
7fe7c5d11000-7fe7c5d12000 r--p 00000000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d12000-7fe7c5d35000 r-xp 00001000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d35000-7fe7c5d3d000 r--p 00024000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d3d000-7fe7c5d3e000 r--p 00000000 08:01 6082        /usr/lib/locale/C.UTF-
8/LC_IDENTIFICATION
7fe7c5d3e000-7fe7c5d3f000 r--p 0002c000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d3f000-7fe7c5d40000 rw-p 0002d000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d40000-7fe7c5d41000 rw-p 00000000 00:00 0
7ffdaa668000-7ffdaa689000 rw-p 00000000 00:00 0           [stack]
7ffdaa6b5000-7ffdaa6b8000 r--p 00000000 00:00 0           [vvar]
7ffdaa6b8000-7ffdaa6b9000 r-xp 00000000 00:00 0           [vdso]
ffffffffff600000-ffffffffff601000 --xp 00000000 00:00 0 [vsyscall]
```

The above result shows six columns in the maps file that represent the following:

1. **Address start – address end** represents the starting and ending address for the mapping. The whole output of the map file is ordered based on these addresses.
2. **Mode** represents the permission or action available on the mapping and whether it is private or shared.
3. **Offset** represents the starting offset in bytes of the file that is mapped. This column is only used for file mappings. For instance, heap and stack mappings are some examples of non-file mappings, thus their offset are 0.
4. **Major:minor ids** represent the device on which the mapped file is located. In the above result, 08:01 represents the major and minor ids of the device where the files are located. For non-file mapping, these ids show 0:0.
5. **Inode id represents** the id of the data structures that contain the core file system-related metadata. This column is also only valid for file mappings.
6. **The file path** represents the path of the file for the mapping.

Let us have a look at some examples of this mapping. The first line shows the mapping of the /usr/bin/sleep file.

```
55fb884a1000-55fb884a3000 r--p 00000000 08:01 1642                      /usr/bin/sleep
```

This file is mapped to the area of memory address 55fb884a1000-55fb884a3000, from the beginning (offset = 0). It has read and private permissions. Most likely, this line represents a private data segment containing global variables and constants, and thus it has read-only access.

```
55fb884a3000-55fb884a7000 r-xp 00002000 08:01 1642                      /usr/bin/sleep
```

The second line shows a different file permission, read, execute, and private. Because this mapping is executable, this segment is the code segment that contains the instructions for the process. Now, let us see the line that is not a file mapping.

```
55fb88fb5000-55fb88fd6000 rw-p 00000000 00:00 0                         [heap]
```

The above line shows the addresses for the heap data structures commonly used to store variables. Because this is not a file mapping, the major, minor, and inode ids are 0. The following lines provide a different range of addresses.

```
7fe7c5682000-7fe7c56b4000 r--p 00000000 08:01 6081        /usr/lib/locale/C.UTF-8/LC_CTYPE
7fe7c56b4000-7fe7c56b5000 r--p 00000000 08:01 6086        /usr/lib/locale/C.UTF-8/LC_NUMERIC
...
7fe7c582f000-7fe7c5b15000 r--p 00000000 08:01 6075        /usr/lib/locale/locale-archive
7fe7c5b15000-7fe7c5b37000 r--p 00000000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
```

```
7fe7c5b37000-7fe7c5caf000 r-xp 00022000 08:01 4694        /usr/lib/x86_64-linux-gnu/libc-
2.31.so
...
7fe7c5d3e000-7fe7c5d3f000 r--p 0002c000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
7fe7c5d3f000-7fe7c5d40000 rw-p 0002d000 08:01 4690        /usr/lib/x86_64-linux-gnu/ld-2.31.so
```

The above lines represent the read-only data segments and code segments of the libraries used by the process.

10.3 Programmatically Allocating Memory in a Process

In C, the stdlib library provides dynamic memory allocation. Using this library, we can programmatically allocate memory for our process. The function provided in this library for dynamic memory allocation is "**malloc()**". This function will allocate a block of memory with specified size and returns a pointer of type void that can be casted into a pointer of any form. The syntax for using the malloc function is defined below.

```
ptr = (cast type*) malloc (byte size)
```

In this syntax, ptr is a pointer of a specified cast type returned by the malloc function referring to a memory with the specified byte size. To show this function in action, let us create a C file using the command "**nano malloc.c**".

```
#include <stdlib.h>
#include <stdio.h>

int main(){

   int *ptr;
   ptr = (int *) malloc (5 * sizeof(int));

   if(ptr != NULL){
      printf("Memory has been successfully allocated.\n");
      printf("Starting address: %p\n", ptr);
      printf("End address: %p\n", ptr+4);

      for(int i=0;i<5;i++){
            ptr[i] = i+1;
      }

      printf("The elements of the array are:\n");
      for(int i=0;i<5;i++){
            printf("%d\n",ptr[i]);

      }
   }
}
```

In the above code, we first declare a pointer variable. Then, we assign this pointer with the address of the memory allocated for the program with five times the size of an integer. This is similar to creating an array to store five integer values. Then, we check if the memory is successfully allocated. If so, we print the start and end addresses allocated. Afterwards, we assign some values to be stored in the allocated memory and print them. The compiling and running of the program will have a result similar to the following.

```
Memory has been successfully allocated.
Starting address: 0x55c43f5cf2a0
End address: 0x55c43f5cf2b0
The elements of the array are:
1
2
3
4
5
```

In the above result, the operating system allocated virtual memory with starting address of 0x55c43f5cf2a0 to the pointer. The size of the allocated memory, by calculating the difference between these two addresses, is 10 bytes. This is equal to the size of an integer (2 bytes) multiplied by five. Then, we print the result.

Activity 10.1

Modify the above program by adding the function "sleep(10000)". You may need to import the unistd.h library. Then compile the program and run it in the background. Identify the process id of this program and check the value of /proc/<pid>/maps file.

1. In what part of the memory is the pointer pointed to?
2. What does it mean in terms of memory allocation?

10.4 Reallocate Memory

We can also change the memory allocated before. For example, the memory size allocated using the malloc function before may no longer be sufficient. Using the "**realloc()**" function, we can dynamically reallocate memory but still preserve the present values. To demonstrate this, let us create a copy of malloc.c using the command "**cp malloc.c realloc.c**". Then, we modify the realloc.c file as follows.

```
#include <stdlib.h>
#include <stdio.h>
```

```
#include <unistd.h>

int main(){

   int *ptr;
   ptr = (int *) malloc (5 * sizeof(int));

   if(ptr != NULL){
      printf("Memory has been successfully allocated.\n");
      printf("Starting address: %p\n", ptr);
      printf("End address: %p\n", ptr+4);

      for(int i=0;i<5;i++){
            ptr[i] = i+1;
      }

      ptr = realloc(ptr, 10 * sizeof(int));
      printf("Successfully realocated the pointer\n");

      for(int i=5;i<10;i++){
            ptr[i] = i+1;
      }

      printf("The elements of the array are:\n");
      for(int i=0;i<10;i++){
            printf("%d\n",ptr[i]);
      }
   }
}
```

After storing some values on the pointer, we then reallocate the pointer to a larger size, which is 10. Then, we assign more values to the pointer. To see the results, let us compile the file and run the executable.

```
Memory has been successfully allocated.
Starting address: 0x556a4a2d42a0
End address: 0x556a4a2d42b0
Successfully realocated the pointer
The elements of the array are as follows.
1
2
3
4
5
6
7
8
9
10
```

In the result, we can see the values that we assigned to the pointer variable.

Activity 10.2

Modify the realloc.c by commenting on the realloc function and its corresponding message printing. Then compile and run the realloc.c again.

1. What is the result?
2. Why does that happen?

10.5 Deallocate Memory

The functions to allocate memory provided by the stdlib library do not de-allocate on their own. To free up a memory allocation, we can use the "**free()**" function. This function only needs one parameter, namely the pointer variable that we want to de-allocate. To illustrate this, create a copy of malloc.c using the command "**cp malloc.c free.c**" and modify the free.c file as follows.

```
#include <stdlib.h>
#include <stdio.h>

int main(){

   int *ptr;
   ptr = (int *) malloc (5 * sizeof(int));

   if(ptr != NULL){
      printf("Memory has been successfully allocated.\n");
      printf("Starting address: %p\n", ptr);
      printf("End address: %p\n", ptr+4);

      free(ptr);
      for(int i=0;i<5;i++){
            ptr[i] = i+1;
      }

      printf("The elements of the array are:\n");
      for(int i=0;i<5;i++){
            printf("%d\n",ptr[i]);
      }
   }
}
```

Activity 10.3

Compile the changes that you made and run the program.

1. What is the result?
2. Can the pointer still be used to store and print values? Why?

References

1. https://web.archive.org/web/20210130190302/http://mugurel.sumanariu.ro/linux/the-difference-among-virt-res-and-shr-in-top-output/
2. https://www.baeldung.com/linux/proc-id-maps
3. https://tldp.org/LDP/sag/html/vm-intro.html
4. https://courses.engr.illinois.edu/cs225/fa2022/resources/stack-heap/