Programming Assignment #2: Memory Layout Printer

Due: Check My Courses

Memory Layout Of A Process

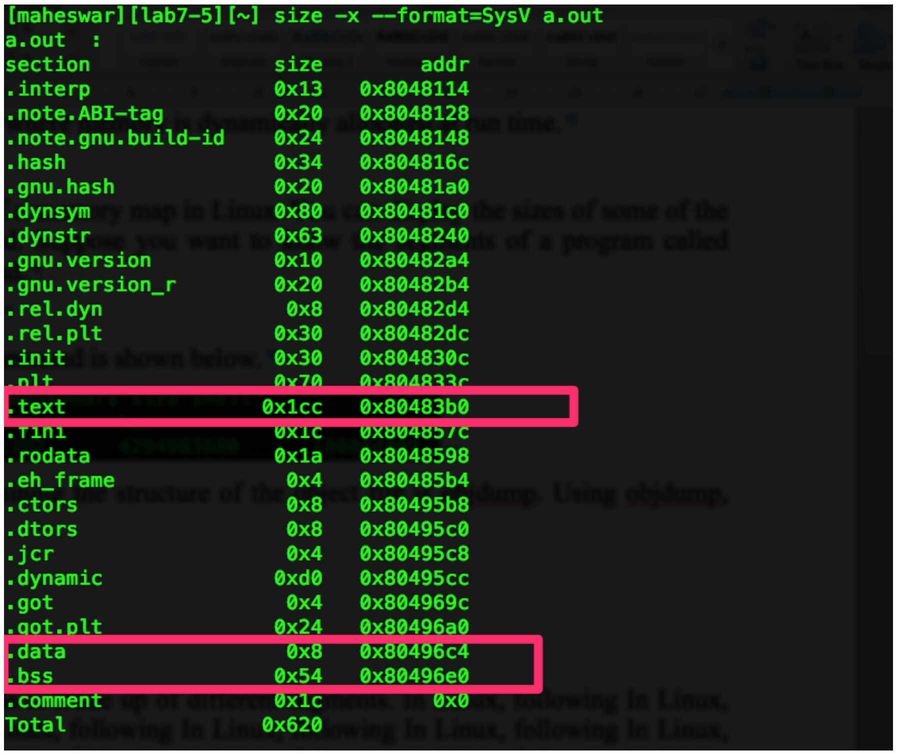
Memory allocated for a process is made up of different segments. In Linux, following are **some of the important** segments provided for a process.

- The text segment contains the machine-language instructions of the program run by the process. The text segment is made read-only so that the process doesn't accidentally modify its own instructions. Also, the text segment is shared between all instances of processes running the same program.
- Initialized data segment (data) contains global and static variables that are explicitly initialized. The initial values are read from the executable file.
- Uninitialized data segment (BSS) contains global and static variables that are not initialized. Before starting the program initializes the whole segment to 0. It is not necessary to allocate space on disk for uninitialized variables so they are separated from the initialized variables.
- The stack is a dynamically growing and shrinking segment containing stack frames. One stack frame is allocated for each currently called function. A frame stores the function's local variables, arguments, and return value.
- The heap is a segment from where memory is dynamically allocated at run time.

Figure 1 shows the key elements of a memory map in Linux. You can display the sizes of some of the segments using the **`size`** command. Suppose you want to know the segments of a program called **`a.out`**. Type the following command.

```
size a.out
```

An example of running the **`size`** command is shown below.



```
[maheswar][lab7-5][~] size -x --format=SysV a.out
a.out  :
section              size      addr
.interp              0x13   0x8048114
.note.ABI-tag        0x20   0x8048128
.note.gnu.build-id   0x24   0x8048148
.hash                0x34   0x804816c
.gnu.hash            0x20   0x80481a0
.dynsym              0x80   0x80481c0
.dynstr              0x63   0x8048240
.gnu.version         0x10   0x80482a4
.gnu.version_r       0x20   0x80482b4
.rel.dyn              0x8   0x80482d4
.rel.plt             0x30   0x80482dc
.init                0x30   0x804830c
.plt                 0x70   0x804833c
.text               0x1cc   0x80483b0
.fini                0x1c   0x804857c
.rodata              0x1a   0x8048598
.eh_frame             0x4   0x80485b4
.ctors                0x8   0x80495b8
.dtors                0x8   0x80495c0
.jcr                  0x4   0x80495c8
.dynamic             0xd0   0x80495cc
.got                  0x4   0x804969c
.got.plt             0x24   0x80496a0
.data                 0x8   0x80496c4
.bss                 0x54   0x80496e0
.comment             0x1c   0x0
Total               0x620
```

There are other tools that provide similar functionality. For example, **`objdump`**, **`readelf`**, and **`nm`** can be used to examine object files. In this programming assignment, you can use these tools to gather relevant information to complete the assignment. Use the man pages to learn how to use these tools to examine the structure of the object files.

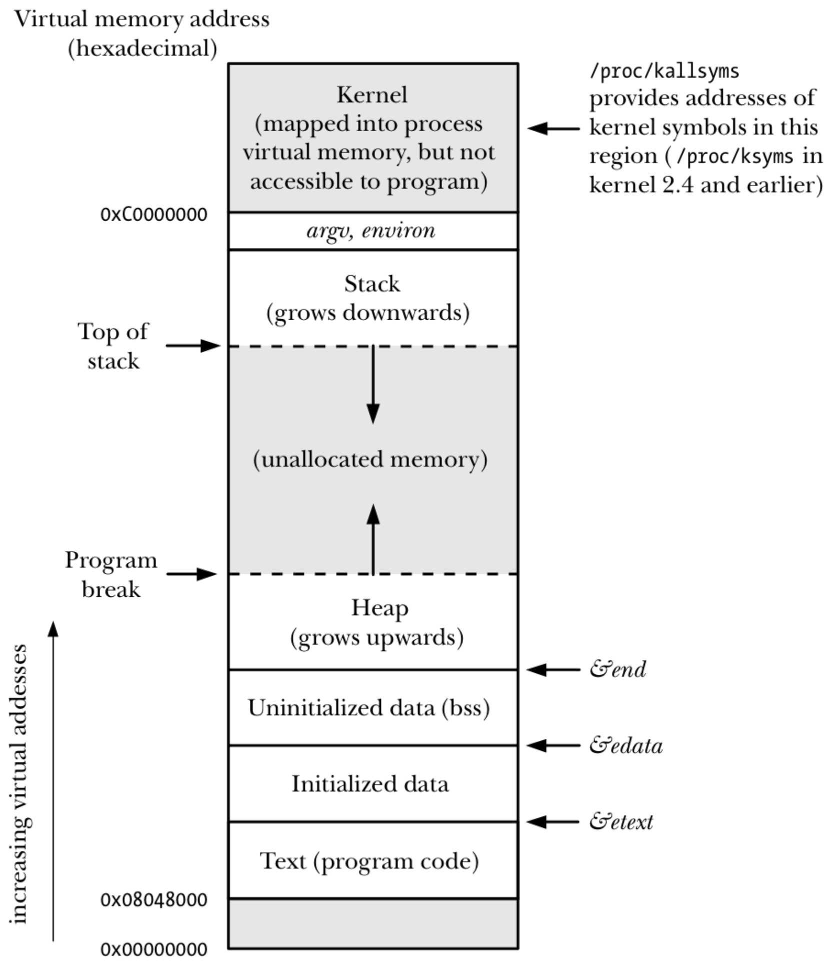

Virtual memory address (hexadecimal)
Kernel (mapped into process virtual memory, but not accessible to program)
/proc/kallsyms provides addresses of kernel symbols in this region (/proc/ksyms in kernel 2.4 and earlier)
0xC0000000
argv, environ
Stack (grows downwards)
Top of stack
(unallocated memory)
Program break
Heap (grows upwards)
&end
Uninitialized data (bss)
&edata
Initialized data
&etext
Text (program code)
0x08048000
0x00000000
increasing virtual addesses

Figure 1: Memory map in Linux.

More On Memory Mapping

Actual memory mapping is little complicated than what was shown in Figure 1. It looks more like what is shown in Figure 2.

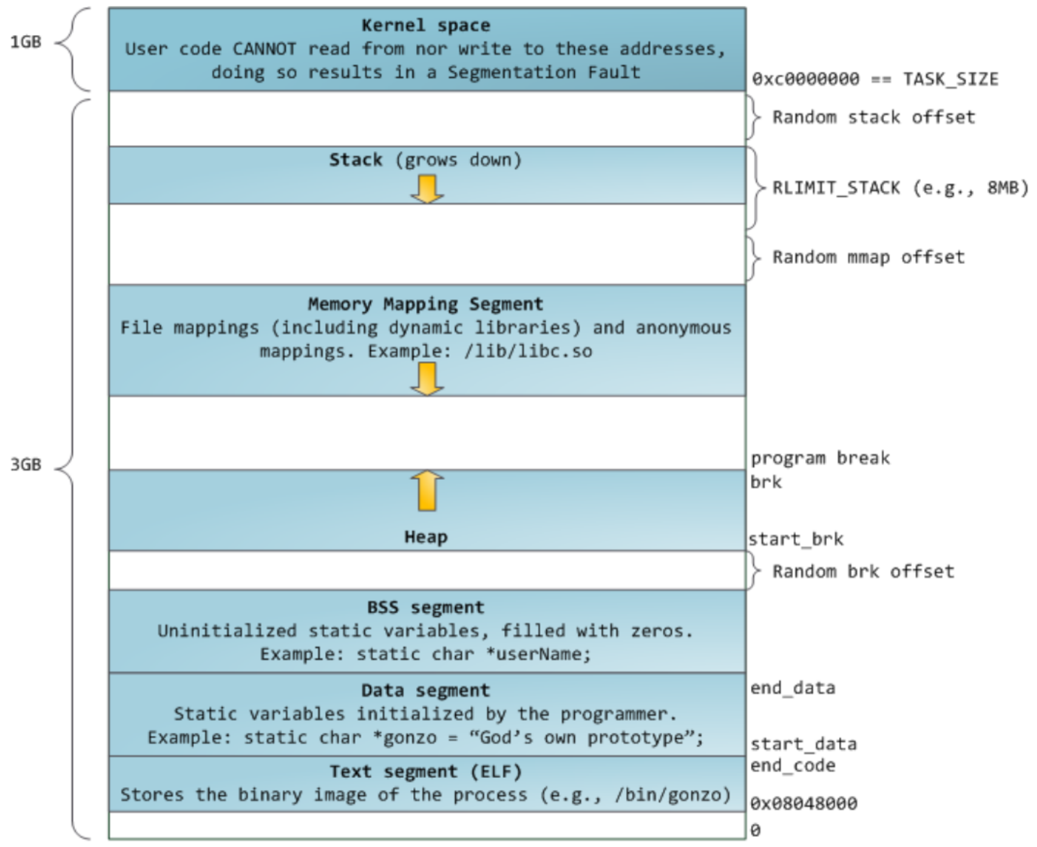

1GB
Kernel space
User code CANNOT read from nor write to these addresses, doing so results in a Segmentation Fault
0xc0000000 == TASK_SIZE
Random stack offset
Stack (grows down)
RLIMIT_STACK (e.g., 8MB)
Random mmap offset
Memory Mapping Segment
File mappings (including dynamic libraries) and anonymous mappings. Example: /lib/libc.so
3GB
program break
brk
Heap
start_brk
Random brk offset
BSS segment
Uninitialized static variables, filled with zeros.
Example: static char *userName;
end_data
Data segment
Static variables initialized by the programmer.
Example: static char *gonzo = "God's own prototype";
start_data
end_code
Text segment (ELF)
Stores the binary image of the process (e.g., /bin/gonzo)
0x08048000
0

Figure 2: Memory map with address space randomization and memory mapped segment.

Modern Linux distributions by default turn on address space randomization. Compare the above memory map with the one given in the beginning of the handout. The memory map given in the beginning of the handout does not have address space randomization or the memory mapped segment. Address space randomization is used to randomly perturb the starting positions of the stack, heap, and memory map segments. This random perturbation is performed to make it harder for the buffer overflow attacks to succeed. You can turn off address space randomization by executing the program in the following manner.

```
setarch `arch' -R your-program
```

You can print the location of the variables, functions, and labels as shown in the sample program below.

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

extern char etext, edata, end;   /* The symbols must have some type,
                                    or "gcc -Wall" complains */

int main(int argc, char *argv[])
{
      int x;
   label:
      printf("First address past:\n");
      printf("    program text (etext)      %10p\n", &etext);
      printf("    initialized data (edata)  %10p\n", &edata);
      printf("    uninitialized data (end)  %10p\n", &end);
      printf("Address of the main func      %10p\n", main);
      printf("Location of variable x        %10p\n", &x);
      printf("Location of a label           %10p\n", &&label);

      exit(EXIT_SUCCESS);
}
```

It is quite easy to obtain the sizes of the text, data, and bss segments. They are fixed for a given program. The stack, heap, and memory map segments vary in size as the program runs. For these segments, you need to find the maximum possible dimensions. You can turn off the address space randomization and find their sizes.

Finding Memory Mapped Segment Sizes

You need to use an "experimental approach" to map the memory mapped segment. The basic idea is to repeat mapping memory segments until the OS fails to allocate the requested size of segment. There are different ideas to try out here. For example, segment sizes could be kept constant or they could increase.

The program below is an implementation the `cat` command in Unix using a memory-mapped segment. In this example, we open the file that needs to be displayed on the screen. We use `fstat` to find the size of the file and use it to map the file to a memory segment. Once the memory segment is successfully created, the file is available as a memory array. You could print the file by simply going through the memory array and printing the characters to the display.

Consult the man page of the `mmap()` system call and carefully study the different options and their meaning. It might be necessary to use a different set of options that what is shown below in your experiments.

```
  #include <stdio.h>
  #include <stdlib.h>
  #include <sys/mman.h>
  #include <sys/types.h>
  #include <sys/stat.h>
  #include <fcntl.h>

  int main(int argc, char *argv[])
  {

        if (argc != 2) {
              printf("\nUsage: %s file-to-cat\n", argv[0]);
```

```
            exit(EXIT_FAILURE);
        }

        // Open the file...
        int fd = open(argv[1], O_RDONLY);
        if (fd < 0) {
            perror(“Opening file”);
            exit(EXIT_FAILURE);
        }

        // Get the size of the file
        struct stat filestat;
        if (fstat(fd, &filestat) == -1)
            perror("Unable to get file size");

        // Do a memory map of the file.
        char *ptr = (char *)mmap(NULL, filestat.st_size, PROT_READ,
                        MAP_PRIVATE, fd, 0);

        // Print the file..
        int i;
        for(i =0; i < filestat.st_size; i++)
            putchar(ptr[i]);

        // Unmap the memory region.
        munmap(ptr, filestat.st_size);
        exit(EXIT_SUCCESS);
    }
```

Finding the Stack Segment Size

The stack segment grows as information is pushed onto the stack. Normally, the program is responsible allocating memory on the stack. However, UNIX based operating systems provide a library function called `alloca()` to allocate storage on the stack. You will use an idea similar to the previous one to progressively increase the allocation on the stack and see at which point the allocations fail. The failure point gives you the maximum allocated size of the stack.

Finding the Heap Segment Size

The heap segment grows as memory is allocated on the heap. This is normally performed by `malloc()` and `realloc()` functions. You can use idea similar to that used with the stack segment to find the heap segment size.

What Do You Need To Do?

You are required to create a map of the virtual address space used by a typical program in Linux and compare it to the ones given in this handout. Because you are observing the address values from inside a C program, your boundaries may not precisely agree with the given ones. It is expected.

Your mapping should take care of the following segments:

1. *Stack segment*. This segment is used by local variables in C. When a function is entered, the C runtime creates a stack frame and allocates the local variables on the stack frame. The memory allocated for the local variables is automatically reclaimed when the function returns. You can allocate an arbitrary amount of memory off the stack using the alloca() system call. This memory is auto-released when the function returns. Your program should be able to estimate the stack dimensions by finding out the addresses of the variables that go onto the stack. A more precise dimensioning is possible using inline assembly code (this is NOT necessary for this assignment).
2. *Heap (dynamic) segment*. This segment is used by dynamically allocated variables. One way of taking memory out of this segment is using malloc() and other variations of this library routine. Again by observing the addresses of the variables that are allocated onto the dynamic segment, you can create a map for this segment.
3. *BSS and Data segments*. Suppose you declare a global array of integers and do not initialize the values it goes into BSS (at least in Linux/gcc). If you initialize the global array with initial values then it goes into the data segment.
4. *Memory mapping segment*. This segment is used for memory mapping a file into memory. You can use this technique to load a file into memory. See the sample program in Piazza that implements ``cat'' using memory mapping. Operating systems load libraries using memory mapping. You can detect the memory address region used for memory mapping by observing the addresses of the regions using this scheme.
5. *Text segment*. This is the segment that contains the program. You can use function addresses and label address values to detect where the text (program) is loaded.

It is important that observing the addresses returned by observing the program variables, functions, and labels is approximate. It gives an easy way of mapping the address regions and the accuracy is sufficient for this assignment.

You will also notice that because of address space randomization that stack, memory mapped, and heap regions start at different locations at different program runs. You can disable the address space randomization to get the same mapping values.

Submit the code you write for find the segment sizes. Also, you need to submit a small document containing the values you found for these segments using your program.