HCMC University Of Technology
Faculty of Computer Science & Engineering

– CO2018 Operating Systems – Spring 2018 –

Lab 4
Process (Part 2/2)

September 21, 2018

Goal: Review the memory layout of a process.

Content In detail, this lab requires student identify the memory regions of process's data segment, including:

- Data segment
- BSS segment
- Stack
- Heap

Furthermore, the lab emphasizes the important of dynamic memory allocation in OS. Students need to consider the data alignment when using dynamic allocation.

Result After completing this lab, students are expected to understand the mechanism of distributing memory region to allocate the data segment for specific processes.

1 Introduction

Traditionally, a Unix process is divided into segments. The standard segments are code segment, data segment, BSS (block started by symbol), and stack segment.

The code segment contains the binary code of the program which is running as the process (a "process" is a program in execution). The data segment contains the initialized global variables and data structures. The BSS segment contains the uninitialized global data structures and finally, the stack segment contains the local variables, return addresses, etc. for the particular process.

Under Linux, a process can execute in two modes - user mode and kernel mode. A process usually executes in user mode, but can switch to kernel mode by making system calls. When a process makes a system call, the kernel takes control and does the requested service on behalf of the process. The process is said to be running in kernel mode during this time. When a process is running in user mode, it is said to be "in userland" and when it is running in kernel mode it is said to be "in kernel space". We will first have a look at how the process segments are dealt with in userland and then take a look at the book keeping on process segments done in kernel space.

In Figure 1.1, blue regions represent virtual addresses that are mapped to physical memory, whereas white regions are unmapped. The distinct bands in the address space correspond to memory segments like the heap, stack, and so on.

Userland's view of the segments

- The Code segment consists of the code - the actual executable program. The code of all the functions we write in the program resides in this segment. The addresses of the functions will give us an idea where the code segment is. If we have a function func() and let p be the address of func() (p = &func;). We know that p will point within the code segment.
- The Data segment consists of the initialized global variables of a program. The Operating system needs to know what values are used to initialize the global variables. The initialized variables are kept in the data segment. To get the address of the data segment we declare a global variable and then print out its address. This address must be inside the data segment.
- The BSS consists of the uninitialized global variables of a process. To get an address which occurs inside the BSS, we declare an uninitialized global variable, then print its address.
- The automatic variables (or local variables) will be allocated on the stack, so printing out the addresses of local variables will provide us with the addresses within the stack segment.
- A process may also include a heap,which is memory that is dynamically allocated during process run time.

2 Practice

2.1 Examining a process

Looking at the following C program with basic statements:

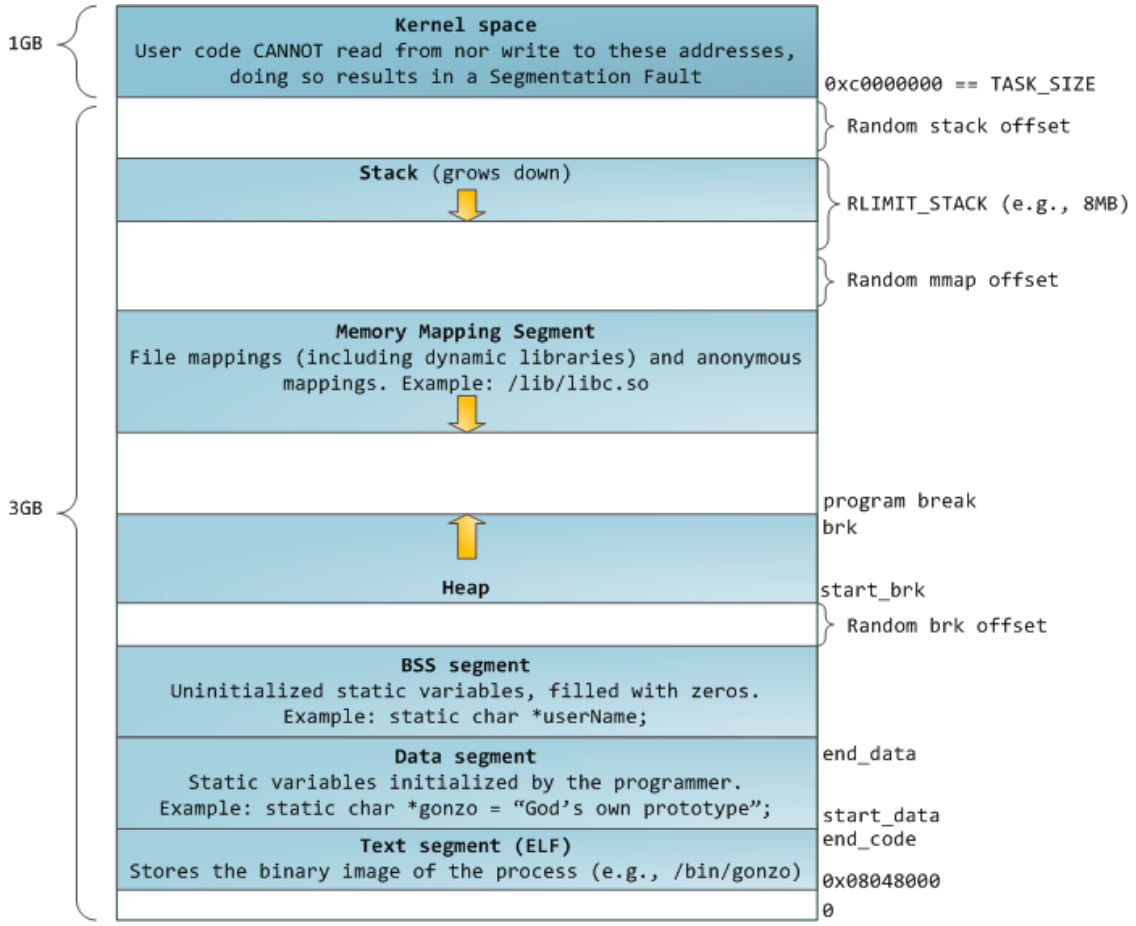

Figure 1.1: Layout of memory segments with process.

```c
#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>

int glo_init_data = 99;
int glo_noninit_data;

void print_func(){
   int local_data = 9;
   printf("Process ID = %d\n", getpid());
   printf("Addresses of the process:\n");
   printf("1. glo_init_data = %p\n", &glo_init_data); Data segment
   printf("2. glo_noninit_data = %p\n", &glo_noninit_data); BSS
   printf("3. print_func() = %p\n", &print_func); Code segment
   printf("4. local_data = %p\n", &local_data); Stack
}
```

```
int main(int argc, char **argv) {
20    print_func();
    return 0;
}
```

Let's run this program many times and give a discussion about the segments of a process. Where is the data segment, BSS segment, stack and code segment?

2.2 Dynamic allocation on Linux/Unix system

2.2.1 malloc

`malloc()` is a Standard C Library function that allocates (i.e. reserves) memory chunks. It compiles with the following rules:

- malloc allocates at least the number of bytes requested
- The pointer returned by malloc points to an allocated space (i.e. a space where the program can read or write successfully)
- No other call to malloc will allocate this space or any portion of it, unless the pointer has been freed before.
- malloc should be tractable: malloc must terminate in as soon as possible.
- malloc should also provide resizing and freeing.

2.2.2 Process's memory and heap

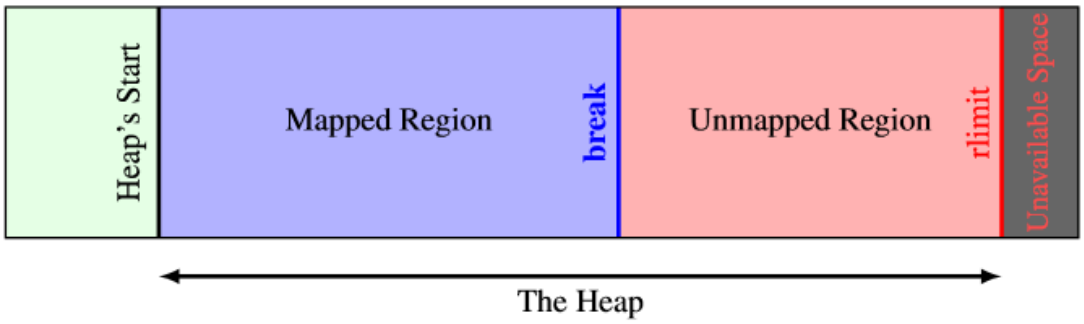

Figure 2.1: Heap region.

The heap is a continuous (in term of virtual addresses) space of memory with three bounds: a starting point, a maximum limit (managed through sys/ressource.h's functions getrlimit(2) and setrlimit(2)) and an end point called the break. The break marks the end of the mapped memory space, that is, the part of the virtual address space that has correspondence into real memory. Figure 2.1 sketches the memory organization.

Write a simple program to check the allocation of memory using malloc(). The heap is as large as the addressable virtual memory of the computer architecture. The following program checks the maximum usable memory per process.

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

int main(int argc,char* argv[]){
   size_t MB = 1024*1024; // # of bytes for allocating
   size_t maxMB = 0;
   void *ptr = NULL;

   do{
      if(ptr != NULL){
         printf("Bytes of memory checked = %zi\n",maxMB);
         memset(ptr,0,maxMB); // fill the allocated region
      }
      maxMB += MB;
      ptr = malloc(maxMB);
   }while(ptr != NULL);

   return 0;
}
```

In order to code a malloc(), we need to know where the heap begins and the break position, and of course we need to be able to move the break. This is the purpose of the two system calls brk() and sbrk().

2.2.3 brk(2) and sbrk(2)

We can find the description of these syscalls in their manual pages:

```
int brk(const void *addr);
void* sbrk(intptr_t incr);
```

brk(2) places the break at the given address *addr* and return 0 if successful, -1 otherwise. The global *errno* symbol indicates the nature of the error.

sbrk(2) moves the break by the given increment (in bytes.) Depending on the system implementation, it returns the previous or the new break address. On failure, it returns (void *)-1 and set *errno*. On some system sbrk() accepts negative values (in order to free some mapped memory.)

The following program implements a simple malloc() function with sbrk(). The mechanism is simple, each time malloc is called, we move the break by the amount of space required and return the previous address of the break. This malloc wastes a lot of space in obsolete memory chunks. Therefore this code is just for a reference of poor practice.

```
#include <sys/types.h>
#include <unistd.h>

void *simple_malloc(size_t size)
{
   void *p;
   p = sbrk (0);
   /* If sbrk fails , we return NULL */
   if (sbrk(size) == (void*)-1)
      return NULL;
   return p;
}
```

3 Exercises

Problem 1 Implement the following function

```
void * aligned_malloc(unsigned int size, unsigned int align);
```

This function is similar to the standard $malloc$ function except that the address of the allocated memory is a multiple of $align$. $align$ must be a power of two and greater than zero. If the $size$ is zero or the function cannot allocate a new memory region, it returns a NULL. For example:

```
aligned_malloc(16, 64)
```

requires us to allocate a block of 16 bytes in memory and the address of the first byte of this block must be divisible by 64. This means if the function returns a pointer to $0x7e1010$ then it is incorrectly implemented because $0x7e1010$ (8261648 in decimal format) is not divisible by 64. However, $0x7e1000$ is a valid pointer since it is divisible by 64.

Associated with aligned_malloc() function, that is free() function to deallocate the memory region allocated in aligned_malloc() function. Along with the implementation of aligned_malloc() function, you have to implement aligned_free() below:

```
void * aligned_free(void *ptr);
```

Given pointer ptr, this function must deallocate the memory region pointed by this pointer.

Problem 2 Write a short essay which summarizing the knowledge of process's data segment to answer for these questions:

- In which cases we should use $alined\_malloc()$ instead of standard $malloc$?
- How can we increase the size of heap in a running process?

Note: You must put the definition of those functions ($aligned\_malloc()$ and $aligned\_free()$) in a file named $ex1.h$ and their implementation in another file named $ex1.c$. DO NOT write $main$ function in those files. You must write the test part of your function in another file. Your answer for problem 2 should be limited within one A4 page. Please put the answer in a PDF file named $ex2.pdf$. Move all of your files ($ex1.h$, $ex1.c$, and $ex2.pdf$) to a single directory <studentID>.zip and submit to Sakai.

A Memory-related data structures in the kernel

In the Linux kernel, every process has an associated struct task_struct. The definition of this struct is in the header file include /linux/sched.h.

```
struct task_struct {
    volatile long state;
    /* -1 unrunnable, 0 runnable, >0 stopped */
    struct thread_info *thread_info;
    atomic_t usage;
    ...
    struct mm_struct *mm, *active_mm;
    ...
    pid_t pid;
    ...
    char comm[16];
    ...
};
```

Three members of the data structure are relevant to us:

- pid contains the Process ID of the process.
- comm holds the name of the process.
- The mm_struct within the task_struct is the key to all memory management activities related to the process.

The mm_struct is defined in include/linux/sched.h as:

```
struct mm_struct {
        struct vm_area_struct * mmap; /* list of VMAs */
        struct rb_root mm_rb;
        struct vm_area_struct * mmap_cache; /* list of VMAs */
        ...
        unsigned long start_code, end_code, start_data, end_data;
        unsigned long start_brk, brk, start_stack;
        ...
}
```

Here the first member of importance is the mmap. The mmap contains the pointer to the list of VMAs (Virtual Memory Areas) related to this process. Full usage of the process address space occurs very rarely. The sparse regions used are denoted by VMAs. The VMAs are stored in struct vm_area_struct defined in linux/mm.h:

```
struct vm_area_struct {
    struct mm_struct * vm_mm; /*The address space we belong to.*/
    unsigned long vm_start; /*Our start address within vm_mm.*/
    unsigned long vm_end; /*The first byte after our end
                            address within vm_mm.*/
    ....
    /* linked list of VM areas per task, sorted by address */
    struct vm_area_struct *vm_next;
    ....
}
```

Kernel's view of the segments

The kernel keeps track of the segments which have been allocated to a particular process using the above structures. For each segment, the kernel allocates a VMA. It keeps track of these segments in the mm_struct structures. The kernel tracks the data segment using two variables: start_data and end_data. The code segment boundaries are in the start_code and end_code variables. The stack segment is covered by the single variable start_stack. There is no special variable to keep track of the BSS segment - the VMA corresponding to the BSS accounts for it.