CS 452 Lab: Week 9
Memory Management

Overview

The main purpose of this lab is to investigate the cooperative role played by the compiler and the operating system in the organization of a process' logical address space. It is designed to improve your understanding of memory allocation protocols. A secondary purpose is to introduce a tool for detecting a common memory error -- a memory bound violation.

Hand-in:

- A word document containing the requested information
- Any requested screenshots (embedded into the word document)
- Program source code (no zip files)
- Annotated memory map of virtual memory organization

Activities

- The first section of the lab introduces a tool to help you test your program for memory leaks.
- The rest of the lab consists of an open-ended investigation into the memory mapping of the current EOS system environment (the GNU **gcc** compiler running under Linux on an Intel processor).

Memory Bound Protection

One of the common problems operating systems face is incorrect memory usage by user programs. This problem can be hard to detect -- most often, systems will "allow" a process to make memory errors, provided they do not corrupt memory belonging to the system or to another process. Basically, a high-performance language like C/C++ will not take the time to check every memory access to ensure it is correct; as that would incur a performance hit. Instead, the operating system will intervene only to protect itself and other users from a misbehaving program.

This section introduces a tool for performing dynamic memory debugging, in order to allow a user to validate a program's use of memory while it is executing. It is most often used together with a debugger.

Buffer Overruns

The scenario induced in sampleProgramOne illustrates a common memory problem: the buffer overrun. The problem will be examined using a tool called Electric Fence. Consider the following sample code:

<u>sampleProgramOne:</u>

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

#define SIZE 16

int main()
{
    char *data1;

    data1 = malloc (SIZE);
    printf ("Please input username: ");
    scanf("%s", data1);
    printf ("You entered: [%s]\n", data1);
    free (data1);
    return 0;
}
```

Although the program appears to run correctly (i.e., it does not abort), it includes a potentially dangerous error that can be detected by using **Electric Fence.**

Study the code carefully. Review the man pages on **Electric Fence** (`man efence`), then perform the following:

- Compile and run the program
  - when prompted, enter the username '`username`'
  - re-run, entering the username '`notARealUsername`'
- Re-compile the program, this time using the debug option and linking in the **Electric Fence** library
  - i.e., use '`-g`' and '`-lefence`'
- Re-run the program
  - when prompted, enter the username '`username`'
  - re-run, entering the username '`notARealUsername`'

Electric Fence places a barrier around a process's allocated memory. This will cause any memory problems in an executing program to suffer a segmentation fault. To find the exact location of the problem, use a debugger to step through the code to find the offending line.

1. Describe the error precisely (nature of problem, location). Fix the problem and submit your corrected source code along with a screenshot of the execution. Develop a robust solution that allows the user to enter any size username (within reason) without massively over allocating memory. (10 points)

Memory Management Overview

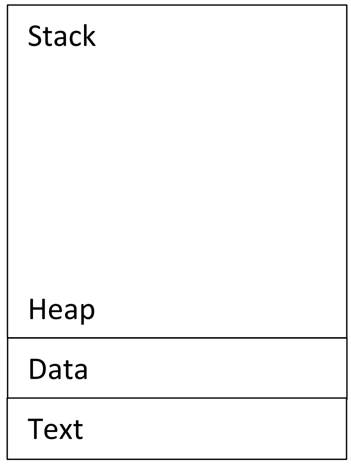
The second part of this lab examines memory management from the system point-of-view. Modern operating systems provide the user with a logical view of memory as an extremely large, contiguous address space. This address space is usually partitioned conceptually into segments: functional partitions of logical memory. The operating system, together with the memory management hardware (the MMU), transparently maps the logical pages of a process into physical memory frames.

Programs consist fundamentally of code and data. However, there are several other distinct regions of user mode logical memory:

- *program text* - this constitutes the machine instructions or program code. It is read-only and of fixed size and initially resides on disk as part of the executable (i.e., the a.out or properly named executable file using the -o compiler option). The size is determined at compile-time and is communicated to the operating system via the header of the executable, so that it can be loaded into the correct amount of memory when run.
- *initialized data* - this data segment holds persistent objects (i.e., globals) that have been initialized with values. Since the data object is to be initialized with a value, the value must be stored as part of the executable.
- *uninitialized data* - this segment holds static (global) objects that have been declared but not initialized. The space for these objects is constructed at run-time by the kernel and initialized to 0 or NULL.
- *run-time data* - this refers to heap space used for dynamic memory allocation. Heap space fluctuates during execution as memory is obtained via **new()** or **malloc()**, and released via **delete()** or **free()**. See the **brk()** and **sbrk()** system calls for more detail.
- *stack* - there is a run-time stack associated with each executing process. It contains stack frames for process context and includes all automatic variables (e.g., non-static data objects such as function parameters and local variables).
- *shared C libraries* - shared libraries loaded during program execution (i.e., dynamically bound code).

The compiler partitions the logical view of your program into these respective regions as it creates the format of the executable. It also places information regarding the sizes of these regions into the program header of your executable. Note that the dynamic regions (stack, heap, uninitialized data) are not actually created until run-time.

These regions each have their own specific locations in virtual memory. As an example, consider Linux memory management. The simplified logical address space of an executing process typically looks like this:



The text and data regions are static in size and are created by the operating system at program load time using the information inserted into the program header by the compiler. The dynamic regions are created and managed at run-time in response to function calls, system calls and process resource requests. Memory management hardware and software cooperate to implement the mapping. For example, using a page table, page# 2 of the program code (in the text region) might be mapped into frame# 0x400006 of physical memory.

Reference Appendix A for some Q&A with ChatGPT. Also feel free to reference other articles online on this topic:
https://medium.com/@shoheiyokoyama/understanding-memory-layout-4ef452c2e709
https://www.geeksforgeeks.org/memory-layout-of-c-program/
https://www.geeksforgeeks.org/stack-vs-heap-memory-allocation/

Memory Mapping Exercise

Perform the following operations:

Create an annotated memory map of GNU/Linux/Intel virtual memory organization:

- Include all of the segments described above
- Determine the direction of growth of the dynamic segments
- Specify the approximate location of each of the segments for a sample program you have written

  Hints:

  - Think about, and create, the type of information each segment stores
  - It is possible to obtain the logical address of any data object using the "address of" operator (**&**).

        - Note: the "%p" format modifier can be used to print addresses (in hexadecimal)
    - Creating multiple objects will indicate their direction of growth. You may need to use a recursive function for the growth to go in the direction you are “expecting”.
    - Do *not* compile your program with Electric Fence included; it changes the location of heap variables

Alternative mapping options:

- Access system variables that contain pointers to specific memory areas
- Some utilities (e.g. **readelf**) interpret binary headers (i.e. they parse an executable)
- Some utilities (e.g. **ldd**) give information about libraries and executables
- The **pmap** command and the **valgrind** utility can be used to determine process memory usage

Synthesize all of this information to help create your annotated diagram.

In addition, perform the following and answer the question:

Run your sample program multiple times. Notice that the dynamic segments appear to "hop around" or re-locate in memory.

2. What's going on here, and why? (5 points)

    - Hint: google "address obfuscation"

3. Submit the source code you used along with the memory map diagram mapping out all of the aforementioned regions. Submit a screenshot of your program execution. (45 points)

Appendix A Chat GPT Q & A

DE

What is the program text of a process running in memory

The program text of a process running in memory refers to the part of the process's memory that contains the executable code of the program. It is also sometimes referred to as the "text segment" or "code segment".

The program text is typically located in a read-only section of the process's memory and contains the compiled machine instructions that make up the program's logic. This code is loaded into memory by the operating system when the process is started and remains there until the process is terminated.

To view the program text of a running process on Linux or Unix systems, you can use the **`pmap`** command-line utility followed by the process ID (PID) of the target process. The program text will be displayed as a range of memory addresses along with the permissions of the memory region (e.g. read-only).

In C, you can access the program text using the **`__text`** symbol, which is defined in the linker script. However, it is generally not recommended to modify the program text segment at runtime, as it can cause unexpected behavior and security vulnerabilities.

DE

What is the initialized data segment

The initialized data segment is a part of a process's memory that contains data values that have been explicitly initialized by the program's source code.

When a program is compiled, any global or static variables that are explicitly initialized are placed into the initialized data segment. This includes variables that are assigned a constant value or initialized with an explicit initializer in the source code.

The initialized data segment is typically located in a read-write section of the process's memory and is allocated when the process is loaded into memory. Unlike the program text segment, which is read-only, the initialized data segment can be modified at runtime.

Some examples of variables that might be stored in the initialized data segment include global variables, static variables, and string literals.

In C, you can declare a variable in the initialized data segment by assigning it an explicit initializer. For example:

c Copy code

```c
int x = 42;         // variable in initialized data segment
char* str = "hello"; // string literal in initialized data segment
```

Note that variables declared without an explicit initializer (e.g. `` `int y;` ``) will be placed in the uninitialized data segment (also known as the BSS segment), which is zero-initialized at program startup.

DE

What is the uninitialized data segment?

The uninitialized data segment, also known as the BSS (Block Started by Symbol) segment, is a part of a process's memory that contains uninitialized or zero-initialized data.

When a program is compiled, any global or static variables that are not explicitly initialized with a value in the source code are placed into the uninitialized data segment. This includes variables that are declared without an explicit initializer, as well as static variables that are initialized to zero.

The uninitialized data segment is typically located in a read-write section of the process's memory and is allocated when the process is loaded into memory. Unlike the initialized data segment, the uninitialized data segment is zero-initialized at program startup, either by the operating system or by the program's startup code.

Some examples of variables that might be stored in the uninitialized data segment include uninitialized global or static variables, or arrays that are declared without an explicit initializer.

In C, you can declare a variable in the uninitialized data segment by declaring it without an explicit initializer. For example:

c Copy code

```c
int x;   // variable in uninitialized data segment
static int y;  // static variable in uninitialized data segment
int arr[100];  // array in uninitialized data segment
```

It's worth noting that while the uninitialized data segment is zero-initialized, it is still good practice to explicitly initialize all variables in your program to avoid potential bugs or security vulnerabilities.

DE Please write a program in c that will map the memory used

To map the memory used by a program in C, we can use the `pmap` command-line utility. However, if you want to do this programmatically within the C program, you can use the `getrusage` function from the `sys/resource.h` header file. Here is an example program:

```c
#include <stdio.h>
#include <sys/resource.h>

int main() {
    struct rusage usage;
    getrusage(RUSAGE_SELF, &usage);

    printf("Memory used: %ld kilobytes\n", usage.ru_maxrss);

    return 0;
}
```

In this program, we declare a `struct rusage` variable named `usage` and pass it to the `getrusage` function along with the `RUSAGE_SELF` constant, which tells the function to return resource usage statistics for the calling process. The `usage.ru_maxrss` field of the `struct rusage` variable contains the maximum resident set size (i.e., the maximum amount of memory used by the process) in kilobytes.

We then print the memory usage in kilobytes using `printf`.

Dr. Bobeldyk Notes: It didn't do the homework for you, plus the information it provided here may not be super useful.

DE

In a running process what is the area of memory known as the 'stack'

In a running process, the area of memory known as the "stack" is a region of memory that is used for storing local variables and function call frames.

Each time a function is called, a new stack frame is created on the stack to store the function's local variables, parameters, and return address. When the function returns, its stack frame is removed from the stack, and execution continues from the return address stored in the previous stack frame.

The stack is typically located at the top of a process's memory address space and grows downward as new stack frames are added. It is a fixed-size memory region that is set by the operating system when the process is started.

The stack is an important part of the process's memory management system, as it allows for efficient allocation and deallocation of local variables and function call frames. However, it is important to be careful with stack usage, as it is limited in size and can lead to stack overflow errors if too much memory is used.

In C, you can declare local variables on the stack by simply declaring them within a function's scope. For example:

c Copy code

```c
void foo() {
    int x = 42;    // local variable on the stack
    // ...
}
```

It's also worth noting that the stack is separate from the heap, which is another area of memory used for dynamic memory allocation. The heap is typically located at the bottom of the process's memory address space and grows upward as new memory is allocated.

DE In a running process what is the area of memory known as the 'heap'

In a running process, the area of memory known as the "heap" is a region of memory that is used for dynamic memory allocation.

The heap is typically located at the bottom of a process's memory address space and grows upward as new memory is allocated. Unlike the stack, the size of the heap is not fixed, and it can grow or shrink dynamically during the execution of the program.

In C, memory can be allocated on the heap using functions like `malloc()`, `calloc()`, and `realloc()`. These functions request a block of memory from the operating system and return a pointer to the allocated memory. The programmer is responsible for managing this memory, which typically involves deallocating it when it is no longer needed using the `free()` function.

The heap is an important part of the process's memory management system, as it allows for dynamic allocation of memory for data structures that may not be known at compile time. However, it is important to be careful with heap usage, as it can lead to memory leaks or heap fragmentation if memory is not properly managed.

Some examples of data structures that might be allocated on the heap include dynamically-sized arrays, linked lists, and trees.

In C, you can allocate memory on the heap using the `malloc()` function, which takes a single argument representing the size of the memory block to be allocated in bytes. For example:

c Copy code

```c
int* arr = malloc(sizeof(int) * 10);   // allocate an array of 10 integers c
```

It's important to remember to deallocate the memory when it is no longer needed to avoid memory leaks. This can be done using the `free()` function, like so:

c Copy code

```c
free(arr);   // deallocate the array when it is no longer needed
```

Dr. Bobeldyk Notes: Don’t you dare use ‘arr’ as a variable name!!!