Linux kernel and C programming

BLOCK 1: Introduction to Memory

Gergely Korcsák

Óbuda University

korcsak.gergely@nik.uni-obuda.hu

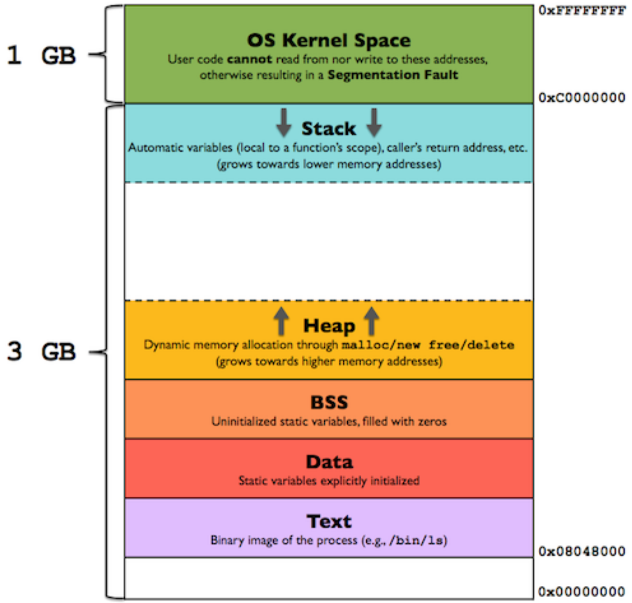
February 25, 2025

Presentation Overview

Memory

Memory

Memory Types


1 GB
3 GB
0xFFFFFFFF
0xC0000000
0x08048000
0x00000000
OS Kernel Space
User code **cannot** read from nor write to these addresses, otherwise resulting in a **Segmentation Fault**
Stack
Automatic variables (local to a function's scope), caller's return address, etc. (grows towards lower memory addresses)
Heap
Dynamic memory allocation through malloc/new free/delete (grows towards higher memory addresses)
BSS
Uninitialized static variables, filled with zeros
Data
Static variables explicitly initialized
Text
Binary image of the process (e.g., /bin/ls)

Program

Program memory

Stores the program (`text`) data. Which is the executable software.

Data

Runtime

The state of a program when it starts executing the main function.

Data memory

Stores the constant and static variables, that are initialized before runtime. These are usually default configs, and parameters, like memory addresses of components (eg: uart driver address map).

BSS memory

Stores zeroed variables, which have a specific size, and initialized before runtime.

Heap

Heap

The heap stores every variable which has been allocated during runtime.

Alert

In the user-space every application memory will be freed, after the program execution, but in kernel-space, the memory will only be freed by ether using the `free` operator or restarting the system.

Stack

Stack

The stack is a constantly changing memory, where every new function variable will be allocated. The growth direction of the stack is dependent on the system, but usually it grows downwards, by every new function call. If a function is executed, and returned, the stack location is freed.

Stack overflow?

Stack overflow is when we fill up the stack and start to use memory areas that are not meant to be used as function variable area. Therefore it can cause execution errors.
Like not returning from a recursive function, before reaching the stack size.

Stack protection

Be aware

As the stack is for functions, it also contains the return addresses. To where the current function will return after its execution is finished. If the stack becomes corrupted by some unchecked memory fill. An ”attacker” can modify the return address, To point to their own code.

Stack canary

There are several stack and return protection features in modern hardware and software, but by default it is usually turned off. We can use stack canaries, to add an extra word of memory into the end of a functions allocated stack, so if we overflow it, it will cause an error.

Memory allocation

Memory allocation

User-space malloc

In user-space we can allocate memory (**from the Heap**) by calling the `malloc()` function.
It will only need a memory size in bytes, and will return a void pointer, to the newly allocated memory. If it fails, it will return 0.

```c
// Will allocate 12 bytes of memory
int* array1 = (int*)malloc(12);
// Will allocate 4x12 bytes of memory
int* array2 = (int*)malloc(sizeof(int) * 12);

struct person {
    char* name;
    int age;
};
// Will allocate 12 bytes of memory
struct person* myperson =
    (struct person*)malloc(sizeof(struct person));
```

User-space free

After we allocate an area, it will not be reallocated to another variables and programs, until it is freed. We can free up an allocated memory area by calling the `free()` function.

```c
// Will allocate 40 bytes of memory
int* array = (int*)malloc(sizeof(int) * 10);

// some array manipulation here..

// Will free up 40 bytes of memory
free(array);
```

User after free

If we would use the previously allocated then freed variable again, it would be ”use after free”, which can cause undefined behavior.

Kernel-space

We can also allocate and free memory in the kernel when programming modules, but using kmalloc and kfree:

```c
// in linux/slab.h
void* kmalloc(size_t size, int flags);
void kfree(void* obj)
```

The flags will be used to decide the type of memory to be allocated.

Flag	Description
GFP_USER	Allocate memory on behalf of user. (may sleep)
GFP_KERNEL	Allocate normal kernel memory. (may sleep)
GFP_ATOMIC	Allocation will not sleep. (use in interrupts)
GFP_NOWAIT	Allocation will not sleep.
...	...

Strings

Strings

String functions

Some string functions are basically `memcpy()`, but will use the `'\0'` character as size.

Function	**Description**
`strlen(str)`	Will return the string length, excluding the `'\0'` character.
`strcpy(dest, src)`	Will copy `src` to `dest`.
`strncpy(dest, src, n)`	Will copy n characters from `src` to `dest`.
`strcat(dest, src)`	Concatenate `src` to the end of `dest`.
`strncat(dest, src, n)`	Concatenate n characters from `src` to the end of `dest`.
`strcmp(dest, src)`	Compares `src` to `dest`.
`strncmp(dest, src, n)`	Compares n characters from `src` to `dest`.

String functions

Some string functions are basically `memcpy()`, but will use the `'\0'` character as size.

Function	Description
`strchar(str, c)`	Find the first occurrence of 'c' or return NULL.
`strrchar(str, c)`	Find the last occurrence of 'c' or return NULL.
`strstr(str, sub)`	Find the first occurrence of the sub-string or return NULL.

We can also use `sprintf()` to print into an allocated memory area, instead of the standard output.

```c
char* fname = "Balu";
char* lname = "Hasu";
char name[100] = { 0 };
sprintf(name, "Name is: %s %s", fname, lname);
printf("name: %s\n", name);
```

”Safe” functions

”Safe” functions

Exercises

Exercises

Exercises

Declare the following functions:

```c
#ifndef __MALLOC_H__
#define __MALLOC_H__

#include <stdint.h>

#define MALLOC_BUF_SIZE 0x1000U
#define BEEF            0xBEEFBEEF
#define NULL            0x0U

struct allocator {
    uint32_t beef;
    uint32_t size;
};

void*   my_malloc(unsigned long long size);
int     my_free(void* obj);

#endif /* __MALLOC_H__ */
```

The End

Questions? Comments?