Deep Dive into Memory with Yash

Part -I

On my journey to understand memory allocation techniques and how memory works behind the user space, I am getting started with an understanding of malloc. **Malloc()**, **Calloc()**, **Realloc()** and Free() are the Dynamic memory allocation functions present in **stdlib.h** .

Throughout the years, various revisions have been implemented on malloc and only one among them stands out. In the late **1980**, **Doug Lea** developed an improved version of **malloc()** known as **dlmalloc()**. It became the basis for the malloc() used in the GNU C library. This version introduced several optimizations and became widely adopted due to its efficiency.

Apart from **dlmalloc()**, there are other versions of malloc; malloc() that was developed for FreeBSD, **tcmalloc()** which is a part of googles performance tools, **ptmalloc()** that is an adaption of **dlmalloc()** to support multiple threads without locks which was later incorporated into glibc. The latest versions of malloc runs on the implementation of **ptmalloc2()**.

Whenever the user calls **malloc()** or any dynamic memory related functions, behind the scenes of them, the system calls **brk()/sbrk()** and **mmap()** are called in order to execute the required action; allocate memory in this case.

The main actions that are performed on a top level are:

- **Determine Available Memory**: Before allocating memory, the system checks if there's enough free memory to fulfill the request. If not, either the request is denied, or some memory management magic is performed (like using swap space).
- **Allocate Memory**: When a program or a part of a program needs memory, it requests a specific amount. The system then finds a contiguous block of memory of the requested size and marks it as in use. This reserved memory is then handed over to the requester.
- **Deallocate Memory**: Once the program is done using the allocated memory, it's essential to return it to the system. This process releases the previously reserved memory, making it available for future allocations either by the same program or by other programs.
- .

brk(): **brk()** is used to change the location of the program break, which defines the end of the process's data segment (i.e., the program's heap). Changing the program break to a higher value increases the amount of heap memory available, while setting it to a lower value decreases it.

int brk(void *end_data_segment);

end_data_segment: The new end of the data segment (i.e., the new program break).

Returns: On success, zero is returned. On error, **-1** is returned, and **errno** is set to **ENOMEM**.

sbrk(): changes the program break by the value specified. It's a more convenient way to allocate or deallocate memory in the heap. Positive values increase the heap size, while negative values decrease it.

void *sbrk(intptr_t increment);

increment: The number of bytes to change the data segment by. Positive values increase the heap, and negative values decrease it.

mmap(): maps either a file or a block of memory into the address space of a process. It's used for memory-mapped files and also for allocating memory, especially large blocks.

void *mmap(void *addr, size_t length, int prot, int flags, int fd, off_t offset);

addr: Suggested starting address for the mapping. Usually specified as **NULL** to let the kernel choose the starting address.

length: The length of the mapping in bytes.

prot: Desired memory protection (e.g., **PROT_READ**, **PROT_WRITE**).

flags: Flags that determine how the mapping is applied. Common flags include **MAP_PRIVATE**, **MAP_SHARED**, and **MAP_ANONYMOUS**.

fd: File descriptor of the file to be mapped. For anonymous memory mappings (i.e., not backed by a file), **-1** is used.

offset: Offset in the file where the mapping starts. For anonymous mappings, this is ignored.

Returns: On success, **mmap()** returns a pointer to the starting address of the mapping. On failure, it returns **MAP_FAILED**, and **errno** is set appropriately.

More programmatical representation in further articles.

Follow the tag #memorywithyash to get updates regarding upcoming articles on Understanding memory in depth in C.

~~~~~~
Do follow me to receive updates regarding Embedded Systems, Space and Technology concepts.
~~~~~~

Happy learning.
Learn together, Grow together.

#earlycareer #embeddedsystems #learning #embeddedc #learnwithyash #linuxfun #memoryallocators