Dynamic Memory Allocation

The malloc family

- The glibc library provides the well known functions:
  - malloc
  - calloc
  - realloc
  - free

- We will see this is not everything needed – under the hood we have:
  - sbrk
  - mmap

malloc

- So exactly what does malloc do?
  - If we ask for 10 bytes, does it give us that 10 bytes, then ask for 8 more and we have two pointers pointing to a total of 18 bytes?
    - Yes and No (more on the no in a minute)

- So first, malloc is a dynamic (meaning at run-time) memory allocation, this is as opposed to static (compile time) allocation
  - Compile Time:
    - char buf [4096];
  - Run Time:
    - char * bufptr = malloc (4096);

malloc

- The signature for malloc is:
  - `void *malloc(size_t size);`

- `size_t` (on a 64 bit processor) is 8 byte unsigned integer

- So we can allocate 16 Exabytes… not exactly
- `warning: argument 1 value ‘18446744073709551615’ exceeds maximum object size 9223372036854775807 [-Walloc-size-largerthan=]`

- So we are limited to *only* 8 exabytes by the compiler

malloc

- So as large as 8 Exabytes, can I allocate 0 bytes?
  - Implementation dependent
    - Some will return a ptr (but to no memory)
    - Others will return Null
  - The first is more common, so you can pass the ptr to free, or to realloc, but that is about it.

Summary of malloc

- Anything special about the malloc allocation? Yes
  - Always aligned on an 8 byte boundary
  - ***You must check for NULL return value!***
  - Memory is random uninitialized values, you must initialize it
  - ***You must `free` every malloc you make!***

free

- For every malloc there shall be a free!

- Free is very straight forward:
  - `void free(void *ptr);`

  - It takes a pointer (one returned from malloc), frees it and returns with no return value.

  - Once freed…

free

- Once freed,

- Do not write on that memory again.

free

- What's wrong with this code??

```
void *ptr = NULL;
[...]
while(<some-condition-is-true>) {
    if (!ptr)
        ptr = malloc(n);

    [...
    <use 'ptr' here>
    ...]

    free(ptr);
}
```

free

- It will crash…

- Even though it looks like each time through the while look it is allocating; using; then freeing memory - that is not the case.
- Ptr is set the first time through the loop, free does NOT set the ptr to NULL, so the second time through the loop it is using the ptr from the first time that has been freed.

- GOOD PRACTICE: set ptr to NULL after a call to free

```
void *ptr = NULL;
[...]
while(<some-condition-is-true>) {
    if (!ptr)
        ptr = malloc(n);

    [...
    <use 'ptr' here>
    ...]

    free(ptr);
}
```

free summary

- The parameter passed to free() must be a value returned by one of the malloc() family APIs (malloc, calloc, or realloc).
- free has no return value.
- Calling free(ptr) does not set ptr to NULL (So, you do so!).
- Once freed, do not attempt to use the freed memory.
- Do not attempt to free the same memory chunk more than once (it's a bug).
- For now, we will assume that freed memory goes back to the system.
- **Do not forget to free memory that was dynamically allocated earlier.** The forgotten memory is said to have leaked out and that's a really hard bug to catch!

calloc, realloc

calloc

- calloc is almost identical to malloc with just two differences
  - It initializes memory allocated to zero values (binary zero).
  - It take two parameters not just one, count and size.

- The calloc(3) function signature is as follows:
  - `void *calloc(size_t nmemb, size_t size);`

- So if you wanted to allocate 500 integers you would:
  - int * ptr;
  - ptr = calloc (500, sizeof(int));

- Allocates `count * size` bytes of memory.

realloc

- realloc allows you to resize a dynamically allocated block of memory.
  - void * realloc(void * ptr, size_t size);
  - ptr must be a valid pointer returned from malloc, calloc, or a previous realloc.
  - size is the new size this block of memory should be (larger or smaller).
  - The return value is NULL if it fails (in which case the original ptr and memory are left unchanged); or the ptr to the reallocated block which may or may not be the same as the original ptr

That's it, right?

- No –
- Now let's look under the hood of malloc.
- malloc is a library function, NOT a system call.
- Let's start by asking, where does malloc get the memory it gives us?
  - For that we need a few new calls and a slight detour.

The Program Break
sbrk

The Program Break

- Nominally when we want memory we call malloc to allocated memory within our heap, but heap is a dynamic segment, it can grow.

- If it is dynamic, there must be a way to track it, i.e. determine the largest address valid in the heap. That point that is the last legally addressable location. This is known as the program break.

- At the onset of your program the program break is just above the uninitialized memory segment.

sbrk

- sbrk() changes the location of the *program break*, which defines the end of the process's data segment (i.e., the program break is the first location after the end of the uninitialized data segment). Increasing the program break has the effect of allocating memory to the process; decreasing the break deallocates memory.

- On success, **sbrk**() returns the previous program break. (If the break was increased, then this value is a pointer to the start of the newly allocated memory). On error, *(void *) -1* is returned, and *errno* is set to **ENOMEM**.

sbrk

- sbrk(0); has the effect of returning the current program break.

```
$ ./show_curbrk
Current program break: 0x1bb4000
$ ./show_curbrk
Current program break: 0x1e93000
$ ./show_curbrk
Current program break: 0x1677000
```

- Why are they different, in modern systems Linux randomizes the layout of the VAS, this is known as Address Space Layout Randomization (ASLR)

malloc under the hood

- If we look at sbrk(0) in relation to mallocs we can see the following:

```
./show_curbrk 1024
Original program break: 0x1488000 ; malloc(1024) = 0x1488670 ;
curr break = 0x14a9000
```

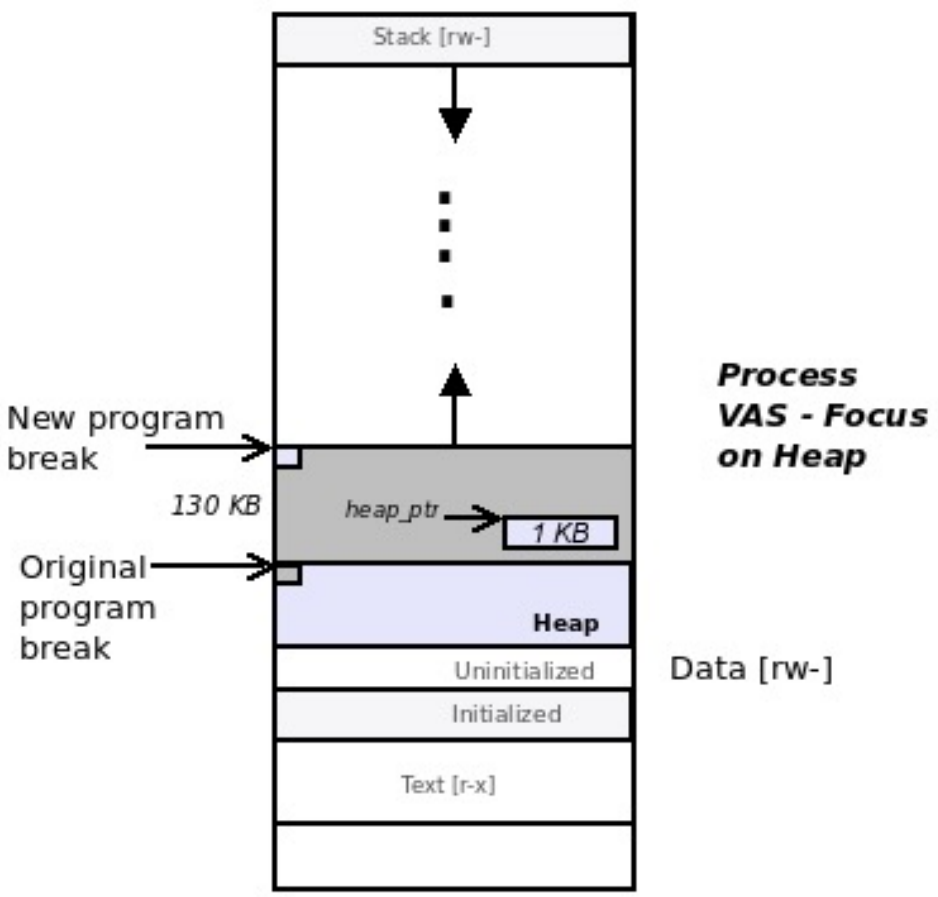
- With an allocation of 1,024 bytes, the heap pointer that's returned to the start of that memory chunk is 0x1488670; that's 0x1488670 - 0x1488000 = 0x670 = 1648 bytes from the original break.

- Also, the new break value is 0x14a9000, which is (0x14a9000 - 0x1488670 = 133520), approximately 130 KB from the freshly allocated block. Why did the heap grow by so much for a mere 1 KB allocation?

malloc / sbrk interaction

- With respect to the diagram:

```
Original program break = 0x14880
heap_ptr = 0x1488670
New program break = 0x14a9000
```



malloc, the real story

- Generally malloc allocates memory from the heap, but that is not always true. malloc is a glibc library function, and it uses subtle strategies to make optimal use of memory
- The library uses a predefined variable called `MMAP_THRESHOLD`, by default this value is 128KB.

- If the allocation is < MMAP_THRESHOLD, then malloc uses the heap segment to allocate the memory
- But, if the allocation is > MMAP_THRESHOLD, and n bytes are not available on the heap's free list, then malloc uses an arbitrary free region of virtual address space to satisfy the allocation

But I want so much more mmap

In comes mmap

- But, how does malloc allocate an arbitrary region.
    - malloc calls the system call mmap
- The mmap system call is very versatile. In this case, it is made to reserve a free region of n bytes of the calling process's virtual address space.
- mmap-ed memory can always be freed up (released back to the system) in an independent fashion when required. free() can not do that.

mmap

- mmap allocations can be expensive.
  - Allocations are page-aligned (potential waste)
  - The kernel zeros out the memory region (expensive on large blocks)

Piecing it together

- Note , after the first change in sbrk(0), it does not change for the subsequent allocations.

```
$ ./malloc_brk_test
                                init_brk =        0x1c97000
 #: malloc(        n) =        heap_ptr           cur_brk   delta
                                                      [cur_brk-
init_brk]
 0: malloc(        8) =        0x1c97670          0x1cb8000 [135168]
 1: malloc(     4083) =        0x1c97690          0x1cb8000 [135168]
 2: malloc(        3) =        0x1c98690          0x1cb8000 [135168]
$
```

In More Detail

- By using the malloc_stats() API, we can see even more detailed information.

```
$ ./malloc_brk_test 1
                                  init_brk =   0x184e000
 #: malloc(        n) =          heap_ptr     cur_brk   delta
                                                [cur_brk-init_brk]
 0: malloc(        8) =          0x184e670    0x186f000 [135168]
Arena 0:
system bytes     =     135168
in use bytes     =       1664
Total (incl. mmap):
system bytes     =     135168
in use bytes     =       1664
max mmap regions =          0
max mmap bytes   =          0
```

In More Detail

```
 1: malloc(    4083) =         0x184e690    0x186f000 [135168]
Arena 0:
system bytes     =     135168
in use bytes     =       5760
Total (incl. mmap):
system bytes     =     135168
in use bytes     =       5760
max mmap regions =          0
max mmap bytes   =          0
```

In More Detail

```
 2: malloc(        3) =         0x184f690    0x186f000 [135168]
Arena 0:
system bytes     =     135168
in use bytes     =       5792
Total (incl. mmap):
system bytes     =     135168
in use bytes     =       5792
max mmap regions =          0
max mmap bytes   =          0
```

In More Detail

```
$ ./malloc_brk_test 2
                                    init_brk =         0x2209000
 #: malloc(        n) =             heap_ptr            cur_brk   delta
                                                              [cur_brk-
init_brk]
[...]

 3: malloc(  136168) =    0x7f57288cd010          0x222a000 [135168]
Arena 0:
system bytes     =     135168
in use bytes     =       5792
Total (incl. mmap):
system bytes     =     274432
in use bytes     =     145056
max mmap regions =          1
max mmap bytes   =     139264
```

In More Detail

```
 4: malloc( 1048576) =   0x7f57287c7010          0x222a000 [135168]
Arena 0:
system bytes     =     135168
in use bytes     =       5792
Total (incl. mmap):
system bytes     =    1327104
in use bytes     =    1197728
max mmap regions =          2
max mmap bytes   =    1191936
```

Summary of malloc's allocations

- Now, we allocate 132 KB (point 3 in the preceding output); some thing to take note of are as follows:
  - The allocations (#3 and #4) are for 132 KB and 1 MB – both above the MMAP_THRESHOLD (value of 128 KB)
  - The (arena 0) heap in-use bytes (5,792) has not changed at all across these two allocations, indicating that heap memory has not been used
  - The max mmap regions and max mmap bytes numbers have changed to positive values (from zero), indicating the use of mmap-ed memory

Where does freed memory go?

- free(), is a library routine – so it stands to reason that when we free up memory, previously allocated by one of the dynamic allocation routines, it does not get freed back to the system, but rather to the process heap.

- However, there are at least two cases where this may not occur:

  - If the allocation was satisfied internally via mmap rather than via the heap segment, it gets immediately freed back to the system

  - On modern glibc, if the amount of heap memory being freed is very large, this triggers the return of at least some of the memory chunks back to the OS.

Demand Paging

Demand Paging

- Question: Does `ptr = malloc (128 * 1024);` actually allocate 128KB of memory?

Demand Paging

- Question: Does `ptr = malloc (128 * 1024);` actually allocate 128KB of memory?

- Of course, asking this question – the answer is no.

Demand Paging

- Question: Does `ptr = malloc (128 * 1024);` actually allocate 128KB of memory?
- Of course, asking this question – the answer is no.
- What happens is space is *reserved* in the process virtual address space
- So when is memory actually allocated?

Demand Paging

- Remember that the Virtual Memory Paging size 4KB, so we requested 32 – 4K pages.

- When we try to read/write/or execute to an address in that space that particular page will be allocated in physical memory. Each of the 32 pages are allocated independently, meaning that you can read from the last byte and write to the first byte and only 2 of the 32 pages will be allocated.

- This is demand paging - paging memory only when actually needed.

- To force the paging – do a memset() across the entire range.

Down side?

- From the man page on malloc
  - By default, Linux follows an optimistic memory allocation strategy. This means that when malloc() returns non-NULL there is no guarantee that the memory really is available. In case it turns out that the system is out of memory, one or more processes will be killed by the OOM killer.

- OOM killer?
  - Out-Of-Memory Killer – used when too many processes try to claim to much physical memory.

Demand Paging

- So how do we know if memory is actually allocated/assigned to physical memory?
  - mincore () – memory in core
    - `int mincore(void *addr, size_t length, unsigned char *vec);`
    - Pass in the base address of the virtual address, and the size
    - Returns a vector array to indicate if each page is allocated or not.

Locking Memory

Locking Memory

- As a normal process, our allocated memory may at any point be virtual, in physical memory, or in swap.

- Page faults (accessing memory when virtual or in swap) is what services to move memory into physical space. Normally this has no effect on an application.

- But, there are times that having memory paged is not desired.
  - Realtime applications
  - Hardware associated memory
  - Cryptography/Security applications

Locking Memory

- Why Cryptography
  - If the security application keeps a “secret” in memory, it may not want that written to swap because it can remain on disk even after the application terminates
  - This is *information leakage*

- So how to we prevent the swap
  - mlock

mlock

- `int mlock(const void *addr, size_t len);`
- If you use `mlock()`, be sure to use `munlock()`
- There are limits RLIMIT_MEMLOCK normally only 64KB
- Privileged processes (root or those with CAP_IPC_LOCK) can lock without the RLIMIT_MEMLOCK limitations.

mlock issues

- Remember when dealing with frames and pages in VM they must be on a page (4K) boundry.
  - That is, (addr % pgsz) == 0. You can use the `posix_memalign()` API to guarantee this

- Sometimes (realtime apps) we don't now which memory to lock down, in this case we can utilize mlockall()

Memory Protection
&
Issues

Memory Protection

- When discussing VM, we talked about the privileges, we can set these utilizing mprotect()
  - `int mprotect(void *addr, size_t len, int prot);`

- Remember that the VM operates on pages, so the address must be page aligned

Protection bit	**Meaning of memory protection**
`PROT_NONE`	No access allowed on the page
`PROT_READ`	Reads allowed on the page
`PROT_WRITE`	Writes allowed on the page
`PROT_EXEC`	Execute access allowed on the page

Memory Issues

- Incorrect memory accesses
  - Using uninitialized variables
  - Out-of-bounds memory accesses (read/write underflow/overflow bugs)
  - Use-after-free/use-after-return (out-of-scope) bugs
  - Double-free
- Leakage
- Undefined behavior (UB)
- Data Races
- Fragmentation (internal implementation) issues
  - Internal
  - External