

Lecture 11.2

Memory Allocation

malloc, the heap, and the kernel's allocators

Md Mahbub E Noor

Assistant Professor, Department of CSE

University of Barishal

```
./chunk (real)
a(1) 0x55c12ffe02a0 chunk 32
b(24) 0x55c12ffe02c0 chunk 32
c(25) 0x55c12ffe02e0 chunk 48
free(b); malloc(20) = 0x55c12ffe02c0
(the same place as b)
# malloc(1) costs 32 bytes
```

What you will learn today



Explain how malloc gets memory from the kernel: brk, mmap and the threshold.



Describe glibc's chunks, headers and free lists, and write a small allocator.



Recognise heap bugs and find them with glibc's checks and AddressSanitizer.



Explain overcommit, the OOM killer, and the kernel's buddy and slab allocators.

Today's plan (60 min)

0-5

Recap of 11.1

5-15

malloc to the kernel

15-30

Inside malloc

30-38

Heap bugs

38-55

The kernel's side

55-60

Checkpoint

Lab 11.2: mymalloc.c, your own malloc.

Quick recap of Lecture 11.1

Q1

After malloc(100 MiB), how much RAM does the process use?

Almost none. Frames come one page at a time, on the first touch: 25 600 minor faults.

Q2

What does fork copy, and when are the pages copied?

The page tables. A page is copied only when someone writes to it (copy-on-write).

Q3

How many levels of page tables, and how many bits per index?

Four levels, 9 bits each, plus a 12-bit offset into the 4 KiB page.

PART 1

From malloc to the kernel

The kernel sells memory wholesale, in pages. malloc sells it retail, by the byte.

Who hands out memory, in which unit?

your code: malloc(100), free(p)

bytes

malloc (glibc): cuts big pieces into small chunks; keeps free lists

chunks of 32, 48 ... bytes

brk / mmap (system calls): grows the heap, or makes a new mapping (VMA)

whole pages

page fault: gives a frame to a page on first touch

4 KiB frames

Analogy: wholesale and retail. The kernel is the wholesaler: it sells only whole boxes (pages), and only on a form (a system call). malloc is the shop: it buys boxes, opens them, and sells you single pieces. When you give a piece back (free), it goes on the shop's shelf, not back to the wholesaler.

SAME AS C

malloc and free from your first semester

Same functions, same rules. Today we look inside them.

DIFFERENT FROM C

free is not "give back to the system"

Usually it only marks the chunk as free inside your process. RAM goes back only for mmap'ed blocks, or when the heap's top shrinks.

Book: CS:APP §9.9.1–9.9.3 (the malloc package)

Small requests: the heap (brk). Big ones: their own mmap

```
● ● ● strace -e trace=brk,mmap ./malloco (real)
```

```
--- malloc(100)
brk(NULL)                = 0x55ffb33e3000
brk(0x55ffb3404000)      = 0x55ffb3404000
--- malloc(1000)        # no system call
--- malloc(100000)      # no system call
--- malloc(200000)
mmap(NULL, 200704, PROT_READ|PROT_WRITE, MAP_PRIVATE|
      MAP_ANONYMOUS, -1, 0) = 0x7f3c61f82000
--- malloc(1048576)
mmap(NULL, 1052672, ...)   = 0x7f3c61e81000
```

First malloc: the heap grows by $0x21000 = 132$ KiB at once. The next 100 000 bytes fit in it: no system call.

≥ 128 KiB: glibc gives the block its own anonymous mapping (DEFAULT_MMAP_THRESHOLD_MIN, malloc.c 946). free returns it with munmap at once.

Threshold moves: after you free an mmap'ed block, glibc raises the threshold to its size (up to 32 MiB): Homework 3.

```
● ● ● ./malloco (the addresses; real)
```

```
malloc( 100) = 0x55ffb33e32a0   malloc(1000) = 0x55ffb33e4320
malloc(100000) = 0x55ffb33e4710 # all in [heap]
malloc(200000) = 0x7f3c61f82010 # own mapping + 16
```

Why two ways? The heap is one contiguous area: a big free hole in its middle cannot be returned. A separate mapping can be returned whole.

The program break: the end of the heap

```
● ● ● include/linux/mm_types.h, Linux 6.8 (real)
```

```
734 struct mm_struct {  
751     struct maple_tree mm_mt; /* the VMAs */  
765     pgd_t * pgd;             /* page tables */  
853     unsigned long total_vm;  
870     unsigned long start_code, end_code, ...  
871     unsigned long start_brk, brk, start_stack;
```

`brk(addr)`

"let my heap end at addr" (mm/mmap.c 178). The kernel grows or shrinks the [heap] VMA; the pages arrive by page fault.

`sbrk(n)`

the libc helper: move the break by n bytes, return the old end. `sbrk(0)` asks where it is.

Our lab's malloc

calls `sbrk` for every new block: 4 system calls above. glibc asks for 132 KiB at once and cuts it.

```
● ● ● strace -e trace=brk ./mymalloc (Lab 11.2; real)
```

```
brk(NULL)                = 0x55b157303000  
brk(0x55b157303090)      = 0x55b157303090 # a  
brk(0x55b1573034a0)      = 0x55b1573034a0 # b  
brk(0x55b157303500)      = 0x55b157303500 # c  
brk(0x55b1573040e0)      = 0x55b1573040e0 # e
```

Book: CS:APP §9.9.1; Love, LKD Ch. 15 (the memory descriptor)

PART 2

Inside glibc's malloc

Chunks with headers, and shelves of free chunks.

A chunk: 8 bytes of header, then your data

```
glibc 2.39 malloc/malloc.c (real lines)

1136 struct malloc_chunk {
1137     INTERNAL_SIZE_T mchunk_prev_size;
1138     INTERNAL_SIZE_T mchunk_size; /* incl. overhead */
1139     struct malloc_chunk* fd; /* only if free */
1140     struct malloc_chunk* bk;
1141     #define PREV_INUSE 0x1
1142     #define IS_MMAPPED 0x2
1143     #define NON_MAIN_ARENA 0x4
```

```
./chunk (reads p[-1]; real)

a(1)    size field 0x021 = chunk 32, flags 1
b(24)   size field 0x021 = chunk 32, flags 1
c(25)   size field 0x031 = chunk 48, flags 1
d(100)  size field 0x071 = chunk 112, flags 1
b - a = 32, c - b = 32, d - c = 48
big     size field 0x31002 = chunk 200704,
        flags 2 (IS_MMAPPED)
```

size | flags (8 bytes)

your data: malloc returns a pointer HERE (24 usable bytes for malloc(1..24))

one 32-byte chunk: the smallest there is (MINSIZE). Sizes are multiples of 16, so the low 3 bits of the size field are free for flags.

malloc(1) costs 32 bytes; malloc(25) costs 48. Many tiny mallocs waste memory: an array of 1000 structs in one malloc is far cheaper than 1000 mallocs.

Book: CS:APP §9.9.6 (implicit free lists, block format)

free puts the chunk on a shelf; the next malloc looks there first

Shelf (glibc 2.39)	What is on it	Speed
tcache	per thread; 64 sizes (16 ... 1032 bytes); up to 7 chunks each (lines 294, 313)	fastest: no lock
fastbins	small chunks, kept as they are, not merged	fast
unsorted, small and large bins	freed chunks after merging with free neighbours; sorted by size	slower
top chunk	the untouched end of the heap; grown by brk when empty	when nothing fits

1

malloc(n): round n up; look on the tcache shelf for that size → bins → split the top chunk → ask the kernel.

2

free(p): small: onto the tcache shelf (last in, first out). Otherwise merge with free neighbours, into a bin.

```
./chunk (real)
```

```
b(24) at 0x55c12ffe02c0
```

```
$ free(b); malloc(20)
```

```
= 0x55c12ffe02c0
```

```
(the same place as b)
```

C link: a free list is a linked list whose nodes ARE the free chunks: the fd pointer is stored inside the freed memory itself. No extra memory for the list.

Last in, first out: the chunk you just freed comes back first. Good for caches; and the reason a use-after-free sees someone else's data.

Book: CS:APP §9.9.13–9.9.14 (explicit and segregated free lists)

The same ideas in 100 lines: our own malloc at work

Inside a heap: our own malloc and free

MALLOC FREE **SPLIT** MERGE GROW

```
d = my_malloc(200);
```



header: 32 bytes (size, next, free)

in use

free

the heap from its start (+0x000) to the program break (not to scale: small blocks are drawn wider so they stay readable)

`d = malloc(200)`: first fit finds b's free block (1008 bytes), takes 208 of them and splits the rest into a new free block with its own header: $1008 - 208 - 32 = 768$.

Click the video to play or pause. About 1 minute, no sound.

Watch for

1. Every block has a **header**.
2. free only **marks** a block.
3. **First fit + split**.
4. Free neighbours **merge**.
5. Only when nothing fits, the **break** moves.

PART 3

Heap bugs

What happens when the rules of malloc and free are broken.

Three classic heap bugs (bugs.c)

```
● ● ● ./bugs ... (glibc's own checks; real)
```

```
$ ./bugs double      # free(p); free(p);  
free(): double free detected in tcache 2  
Aborted  status 134  
$ ./bugs overflow    # 16 bytes past a 24-byte block  
malloc(): corrupted top size  
Aborted  status 134
```

```
● ● ● ./bugs uaf      # use after free (real)
```

```
p still points to: "new data"  
uaf: the program went on      status 0
```

The silent one is the dangerous one. The use-after-free gave no error: p and q are the same chunk now (last in, first out).

```
● ● ● gcc -fsanitize=address; ./bugs_asan uaf (real)
```

```
==429==ERROR: AddressSanitizer: heap-use-after-free  
on address 0x503000000040  
READ of size 2 at 0x503000000040 thread T0  
#3 ... in main bugs.c:23  
0x503000000040 is located 0 bytes inside of  
32-byte region [0x503000000040,0x503000000060)  
freed by thread T0 here:  
#1 ... in main bugs.c:20  
previously allocated by thread T0 here:  
#1 ... in main bugs.c:18
```

gcc -Wall saw all three at compile time: -Wuse-after-free and -Wstringop-overflow. Read your warnings.

Book: CS:APP §9.11 (common memory-related bugs)

PART 4

The kernel's side

Promises it may not keep, a killer for emergencies, and allocators for itself.

Overcommit: the kernel promises more than it has

```
./overcommit (malloc only, never touch; real)
```

```
# our machine: 8 GiB RAM, no swap
malloc( 1 GiB) succeeded
malloc( 2 GiB) succeeded
malloc( 4 GiB) succeeded
malloc( 8 GiB) FAILED
# QEMU guest, 512 MiB: malloc(1 GiB) FAILED
```

```
mm/util.c, Linux 6.8 (real)
```

```
942 int __vm_enough_memory(..., long pages, ...)
954     if (sysctl_overcommit_memory == OVERCOMMIT_GUESS) {
955         if (pages > totalram_pages() + total_swap_pages)
            goto error;      /* -ENOMEM */
```

/proc/sys/vm/overcommit_memory	Rule
0 (default; both our machines)	refuse one request bigger than RAM + swap; promise the rest
1	always say yes
2	never promise more than CommitLimit (swap + 50 % of RAM)

Why promise? Most programs never use all they malloc (and fork would need double the memory for a moment). Promising lets more programs run. The price comes next slide.

C link: "malloc returned non-NULL, so I have the memory" is not quite true on Linux. The memory arrives on first touch, if it is still there.

When the promises are called in: the OOM killer

```
● ● ● guest, 512 MiB: ./overcommit touch 600
```

```
touched 0 MiB
touched 100 MiB
touched 200 MiB
touched 300 MiB
touched 400 MiB
Killed
status 137          # 128 + 9: SIGKILL
```

```
● ● ● dmesg (the same guest; real)
```

```
[6.269525] overcommit invoked oom-killer:
gfp_mask=0x140dca(GFP_HIGHUSER_MOVABLE|
__GFP_COMP|__GFP_ZERO), order=0
[6.276573] Out of memory: Killed process 98
(overcommit) total-vm:441496kB,
anon-rss:432256kB, file-rss:0kB, ...
```

Where

inside a page fault: the kernel needs a frame, none is free, nothing can be evicted.

Who dies

the process with the highest "badness" (mm/oom_kill.c 201): mostly the one using the most RAM. See /proc/PID/oom_score.

How

SIGKILL (Lecture 10.2): it cannot be caught. The program never gets an error code.

Book: Love, LKD Ch. 15; mm/oom_kill.c

The kernel's own pages: the buddy allocator

order	0	1	2	3	4	5	6	7	8	9	10
block size	4K	8K	16K	32K	64K	128K	256K	512K	1M	2M	4M
free (DMA32)	16	9	7	10	7	7	4	4	1	5	87

cat /proc/buddyinfo in the QEMU guest (real): free blocks of each size in zone DMA32; MAX_PAGE_ORDER = 10 (include/linux/mmzone.h 30)

1 Free memory is kept in blocks of 2^{order} pages, one free list per order.

2 **Allocate** 2^k pages: take a block of order k ; if none, split a bigger one in halves, again and again (expand, page_alloc.c 1386).

3 **Free**: if the block's "buddy" (its other half) is free too, merge them into the next order, again and again (__free_one_page, 764).

Why buddies? A block's buddy is found with one XOR on its page number: merging is fast, and big contiguous blocks come back.

87 blocks of 4 MiB free: a fresh machine has big blocks. After a long time, memory gets fragmented: only small blocks are left, and 2 MiB huge pages become hard to find.

Book: Love, LKD Ch. 12; Bovet & Cesati Ch. 8 (the buddy system)

Small kernel objects: the slab allocator (kmalloc)

```
● ● ● grep ... /proc/slabinfo (QEMU guest; real)

# name          active  total  objsize  per slab  pages/slab
task_struct     63      63    10176      3         8
mm_struct       11      11     1408     11         4
kmalloc-4k        8        8     4096      8         8
kmalloc-1k        8        8     1024      8         2
kmalloc-256       64       64      256     16         1
kmalloc-128       32       32      128     32         1
kmalloc-64        64       64       64     64         1
kmalloc-32       512      512       32    128         1
kmalloc-16       256      256       16   256         1
kmalloc-8        512      512        8   512         1
```

A cache per kind: One cache for task_struct (10 176 bytes, Lecture 8.1): 3 fit in a slab of 8 pages. No header per object, no waste between them.

kmalloc(n): the kernel's malloc: rounds n up to the next size class, 8 ... 8192 bytes, and takes an object from that cache.

Fast reuse: a freed task_struct stays in its cache, ready for the next fork.

SAME AS C

Like malloc's size classes

kmalloc-32 is the kernel's version of glibc's 32-byte tcache bin: same idea, one level down. The slab gets its pages from the buddy allocator.

Book: Love, LKD Ch. 12 (the slab layer)

The whole chain, from malloc(100) to a frame of RAM

Layer	Unit	Where it lives	Lecture
malloc / free	chunks of 32, 48 ... bytes	glibc, in your process	11.2
brk / mmap	VMAs, whole pages	system calls → mm_struct	9, 11.2
page fault	one 4 KiB frame per touched page	handle_mm_fault	11.1
buddy allocator	2^order pages	mm/page_alloc.c	11.2
slab (kmalloc)	kernel objects: task_struct, 8 ... 8192 bytes	mm/slub.c	11.2
OOM killer	a whole process	mm/oom_kill.c	11.2

Each layer is lazy: malloc asks the kernel only when its shelves are empty; the kernel gives frames only on the first touch; the OOM killer acts only when there is truly nothing left. That is why a C program can usually ask for memory quickly.

LAB 11.2

mymalloc.c: your own malloc

First fit, split, free, merge: about 100 lines on top of sbrk.

Lab 11.2 tasks, step by step

1 **TODO 2** first (grow): `sbrk(HDR_ALIGNED + n)`, fill in the header, append it to the list

2 **TODO 1** find_fit: walk the list; return the first free block with size $\geq n$

3 **TODO 3** split: if the block is much bigger than `n`, cut the rest into a new free block

4 **TODO 4** free: mark it free; merge with the next block while it is free and adjacent

5 Compare your heap dumps with the expected ones; then `strace -e trace=brk ./mymalloc`

Why a static stdout buffer? `printf` would call `malloc` for its buffer, and `glibc`'s `malloc` would move the same program break: two allocators fighting over one heap.

   `./mymalloc` (reference; real, shortened)

```
1. a = 100, b = 1000, c = 50 bytes:
   +0x000: 112 used +0x090: 1008 used
   +0x4a0: 64 used  break +0x500
2. free(b): +0x090: 1008 FREE
3. d = malloc(200): first fit + split
   +0x090: 208 used +0x180: 768 FREE
4. free(d): merges: +0x090: 1008 FREE
5. free(a); e = malloc(3000):
   +0x000: 1152 FREE +0x4a0: 64 used
   +0x500: 3008 used break +0x10e0
c still holds "C"
```

Checkpoint: test yourself

Q1 `malloc(300000)`: heap or `mmap`? And `malloc(3000)`? → `mmap` (≥ 128 KiB); the heap

Q2 How many bytes does `malloc(1)` really take? → 32: an 8-byte header plus 24 usable bytes (the minimum chunk)

Q3 You free a chunk and `malloc` the same size again. Where does it come from? → usually the same place: the tcache shelf, last in, first out

Q4 `malloc` succeeded, but the program was killed later. What happened? → overcommit: on touching the pages, RAM ran out; the OOM killer sent SIGKILL

Q5 How does the kernel allocate a `task_struct`? → from its slab cache (`kmem_cache`), whose pages come from the buddy allocator

Key ideas from today



malloc gets pages from the kernel with brk (the heap) or mmap (≥ 128 KiB), and sells bytes.



Each chunk has an 8-byte header; sizes are multiples of 16; malloc(1) costs 32 bytes.



free puts chunks on shelves (tcache, bins); neighbours merge; RAM rarely goes back.



Double free and overflows may abort; use-after-free is silent. -Wall and -fsanitize=address help.



Linux overcommits; when the pages are touched and RAM is gone, the OOM killer sends SIGKILL.



The kernel: buddy allocator for pages, slab caches (kmalloc) for its own small objects.

Homework 11.2 (due before Lecture 12.1)

1. Fewer calls

Change your `grow()` to ask `sbrk` for at least 64 KiB and split off the rest as a free block. How many times does the break move now in the test? (`strace`)

2. Real sizes

Predict `malloc_usable_size` for `malloc(0)`, 1, 24, 25, 40, 41 and 1000; then check with `usable.c`. State the rule.

3. Moving threshold

`threshold.c` does `malloc(200000)` and free three times. Predict the system calls; run `strace -e trace=brk,mmap,munmap`; explain round 2.

4. Your machine

Run `overcommit` on a lab PC. What is the biggest `malloc` that succeeds? Compare with `MemTotal + SwapTotal` and `mm/util.c` line 955.

5. Sanitizer

Build `bugs.c` with `-fsanitize=address` and run `./bugs_asan` overflow. Which line does it blame, and how many bytes past the block?

Next lecture and reading

NEXT: LECTURE 12.1

Files and the VFS

What is behind a file descriptor: the file table, inodes and directories, the virtual file system that makes ext4, /proc and pipes look alike, and the page cache you met today.

Main text

R. E. Bryant & D. R. O'Hallaron, **CS:APP**, 3rd ed., §9.9–9.11 today; Ch. 10 (system-level I/O) for 12.1.

Kernel side

R. Love, **LKD**, Ch. 12 (memory) today; Ch. 13 (the virtual file system) for 12.1.

Source

Linux 6.8 mm/mmap.c, mm/util.c, mm/oom_kill.c, mm/page_alloc.c; glibc 2.39 malloc/malloc.c (Linux v6.8 source online: elixir.bootlin.com/linux/v6.8).

Lab files

`lab11_files.zip`: `mymalloc_starter.c` and the demos.