CS 347M (Operating Systems Minor)

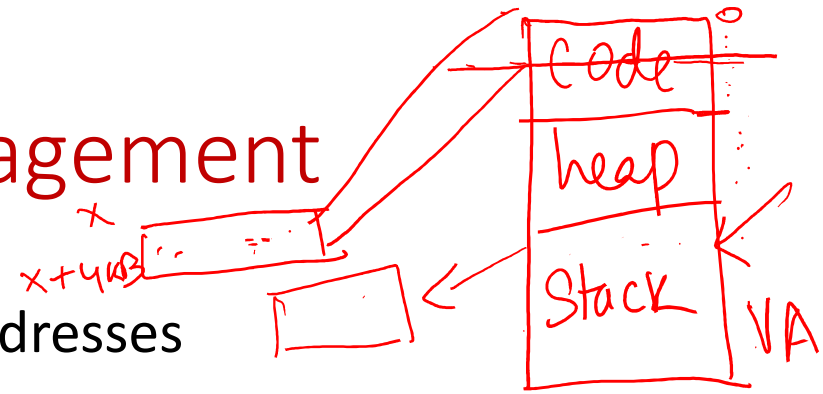
Spring 2022

Lecture 10: Memory allocation in user programs

Mythili Vutukuru

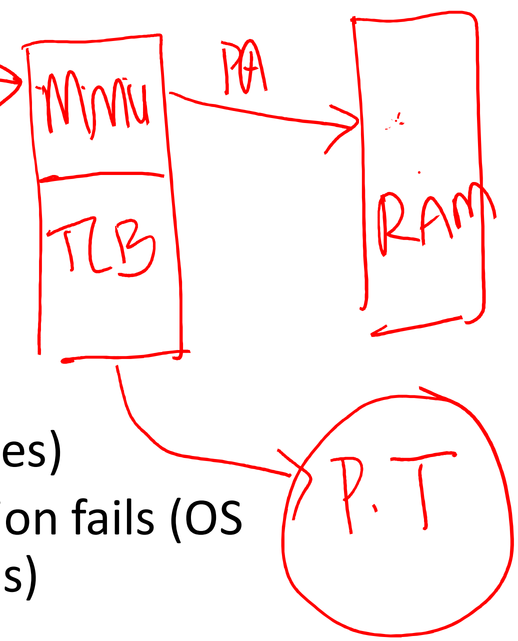
CSE, IIT Bombay

Recap: Virtual memory management

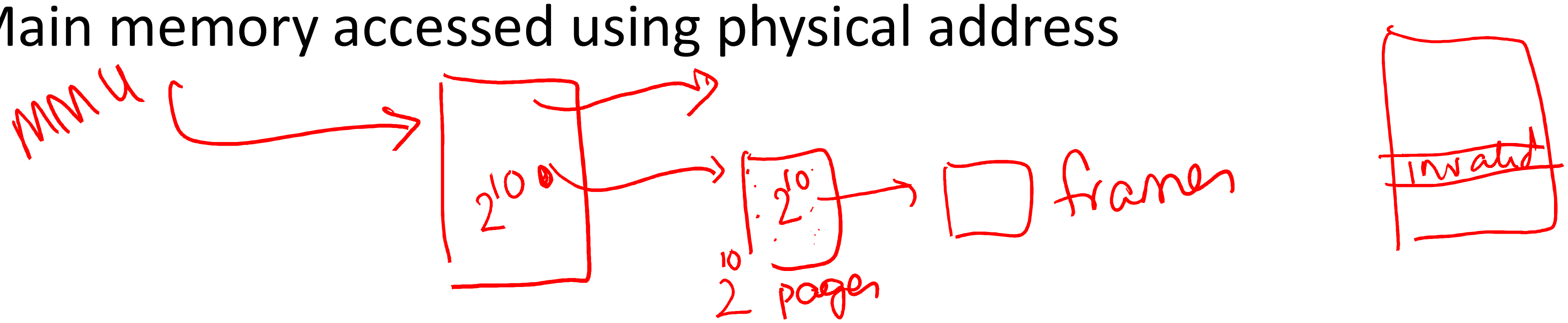


- Code+data of a process assigned virtual addresses
- Fixed size chunks of virtual addresses (pages) mapped to chunks of physical addresss (frames) in main memory
- Page table of process maps page# to frame#
  - Used by MMU for address translation
  - Page table mappings cached in TLB
- Not all valid pages (virtual addresses) assigned frames (physical addresses) by OS always
  - Some pages may be mapped to frames only when used (demand paging)
  - MMU traps to OS when it cannot translate address (page fault)

Recap: What happens on a memory access



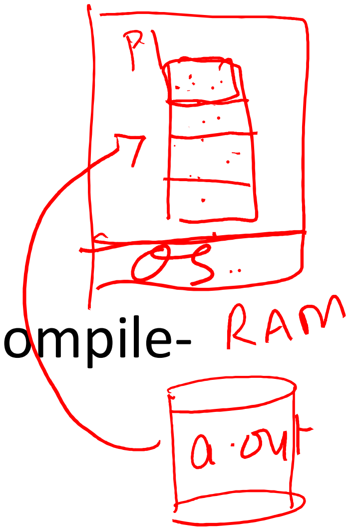
- CPU accesses code/data at a certain virtual address
- Check CPU cache, else fetch from main memory
- MMU translates virtual address to physical address
  - If TLB hit, physical address directly available
  - If TLB miss, MMU walks page table (multiple memory accesses)
  - During page table walk, MMU traps to OS if address translation fails (OS handles page fault, updates page table entry, restarts process)
- Main memory accessed using physical address



Running a program

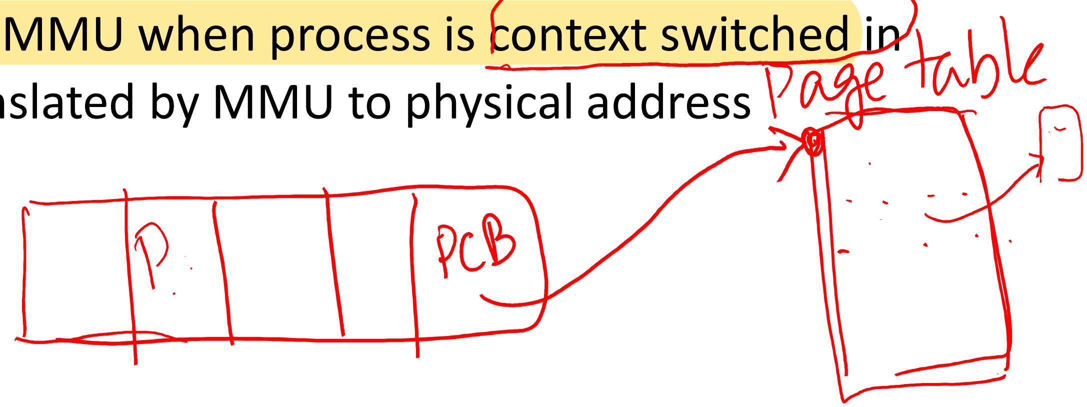
P → C

P.T created

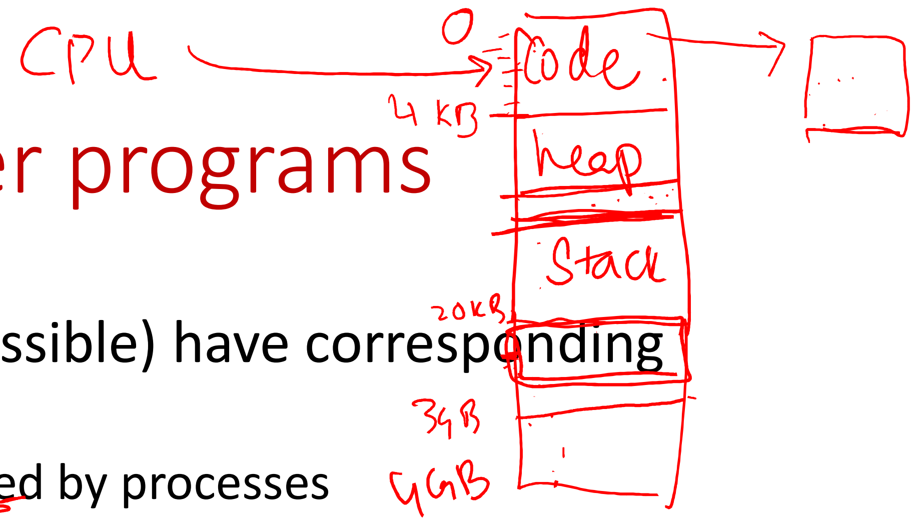


- User program is compiled into executable = program code + compile-time data (global/static variables)
- When program is run, OS creates process
  - Allocates some physical frames to store executable code/data, stack, heap, ..
  - Allocates page table, adds mapping from virtual addresses (page numbers) to physical addresses (frame numbers) in page table
  - Pointer to page table provided to MMU when process is context switched in
  - CPU accesses virtual address, translated by MMU to physical address

OS memory



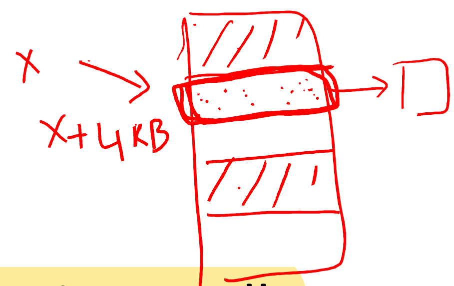
Memory allocation to user programs



- Not all virtual addresses (0 to MAX possible) have corresponding physical addresses

invalid

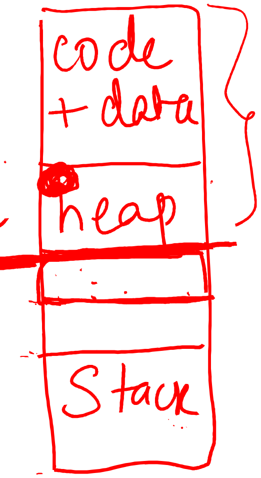
  - Not all virtual addresses available are used by processes
  - Not all used virtual addresses are assigned physical addresses by OS
- How can process ask for physical memory to be allocated for its valid virtual addresses?
  - Access virtual address which has no memory allocated
  - OS handles page fault and assigns physical memory
  - Keep using memory actively, so that OS doesn't reclaim it



- How can process expand its virtual address space? Ask OS via syscalls

System calls for virtual memory allocation

program break malloc



- Default virtual address space: code+static data, stack, heap, …
  - Most programs are happy with this, no need to use syscalls to expand
  - Heap expansion taken care of by programming language libraries

malloc(8)

- System calls to expand virtual address space
  - The syscalls brk/sbrk are used to allocate page-sized chunks at the end of the data segment of executable (called program break)
  - mmap syscall used to obtain one or more page sized chunks at any unallocated range of virtual addresses

heap

malloc

mmap

- Syscalls sbrk/brk/mmap used by heap managers to expand heap, can also be used by user programs (instead of malloc) to obtain larger chunks of memory

P 10 → heap

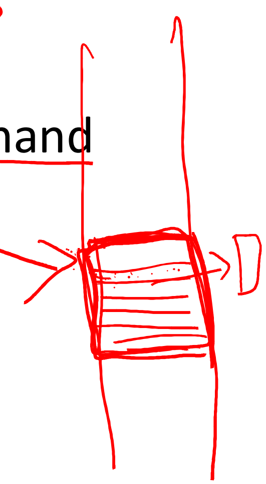
Memory mapping

4KB OS

(int *) int[0] int[1]

```
char *buf = mmap(address, size, ..)
buf[0] = ...
```

- mmap system call for mapping new memory into address space
  - Used to expand virtual address space
  - Takes starting virtual address, size of memory required as arguments (must be multiple of page size)
  - OS allocates unused virtual addresses, marks page table entries as valid
  - Syscall returns the starting virtual address of the allocation
  - Physical memory corresponding to the virtual addresses allocated on demand
- How is memory-mapped page used? malloc(5) mmap
  - Allocated memory can be split into smaller chunks by heap manager
  - Allocated memory can be directly used in user programs to store data



char * ptr = malloc (10)
free (ptr)

test.c
malloc

C library

syscall
OS

0
int

Heap management

- Heap: one or more pages of memory used for dynamic memory allocation via malloc, managed by programming language libraries
- Heap manager: code within libraries to manage heap
  - Exposes API to allocate/deallocate memory in variable-sized chunks at run time (e.g., malloc, free)
  - Gets page-sized chunks from OS using sbrk/brk/mmap
  - Splits pages into smaller chunks and gives out during malloc
  - Malloc returns the starting virtual address of the free chunk of requested size
  - Freed up memory in heap is reused for future allocations
  - Some language libraries automatically clean unused chunks, some do not (malloc-ed memory must be explicitly freed up by user, else memory leak)

12 KB
16 KB

int a
&a

C / C++
malloc
free

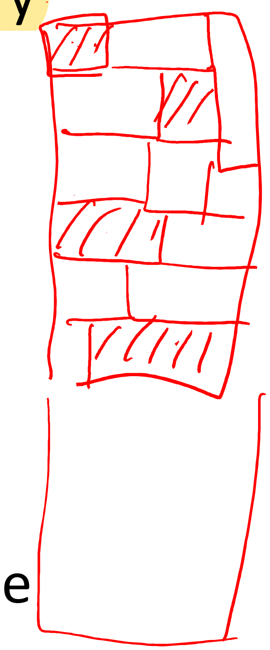
malloc {

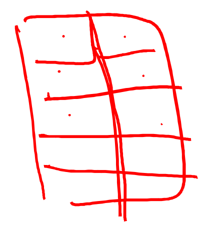
}

implementation

Types of heap managers

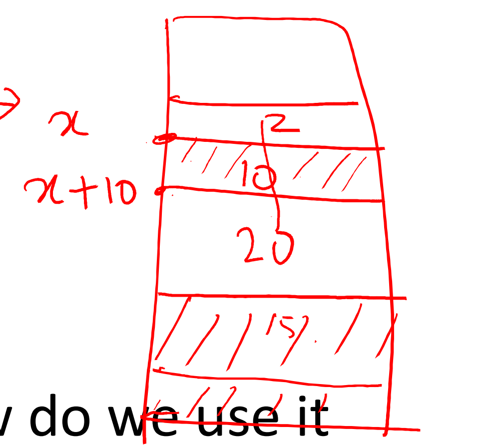
- General-purpose heap managers support variable sized memory allocation using malloc
  - Complex data structures to keep track of variable sized free chunks
  - Frequent calls to malloc/free can slow down user programs
- Some heap managers optimized for fixed size allocation: slab allocators
  - Useful for user applications that allocate memory in fixed sizes
  - Heap memory is divided into fixed size chunks for allocation
  - More efficient than general-purpose variable sized allocation
- Usually managed by programming language libraries, but can be configured/changed by users



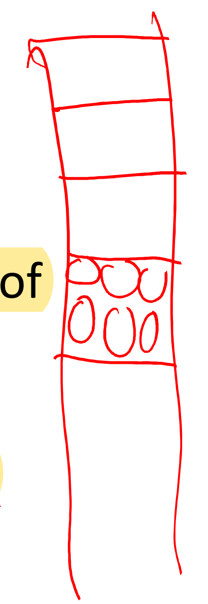


free (x)   malloc

Variable sized memory allocation



- How do we design a general purpose heap manager?
- Given a large block of memory (one or more pages), how do we use it to satisfy variable-sized memory allocation requests?
- Why is this hard?
  - Need to track information about variable sized chunks: what sized chunk is allocated where in memory
  - External fragmentation: after memory is allocated and freed-up, many gaps of different sizes throughout heap
  - Need to keep track of various free chunks to reuse for later allocations
  - What if chunk of desired size not found? May need to merge or split chunks



Variable sized allocation: headers

- Consider a simple implementation of `malloc`
- Every allocated chunk has a header with info like size of chunk
  - Why store size? We should know how much to free when `free()` is called
- Header also has some random number ("magic")
  - Detects if previous chunk has overflown into this chunk

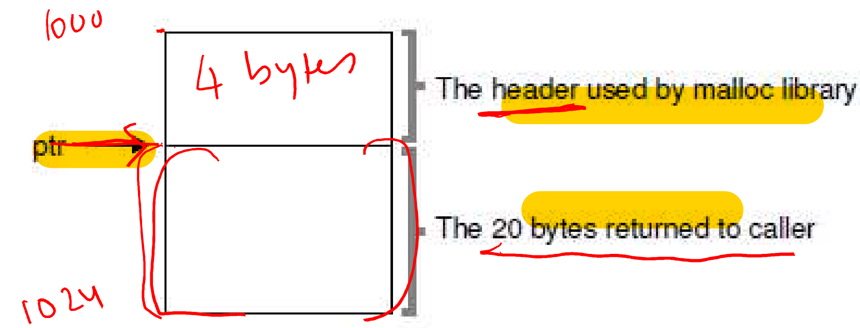
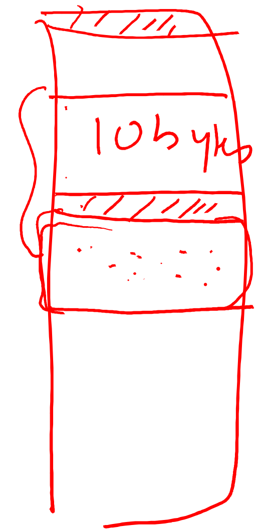


Figure 17.1: An Allocated Region Plus Header



free (1004)

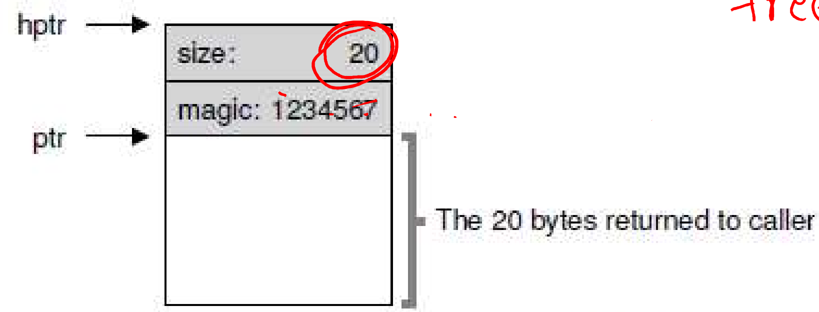


Figure 17.2: Specific Contents Of The Header

Variable sized allocation: free list

- How to store information about which chunks are free?
- Free space can be managed as a linked list of free chunks
  - Pointer to the next free chunk is embedded within the free chunk
- Need to only remember head of list within heap manager
  - Allocations happen from the head

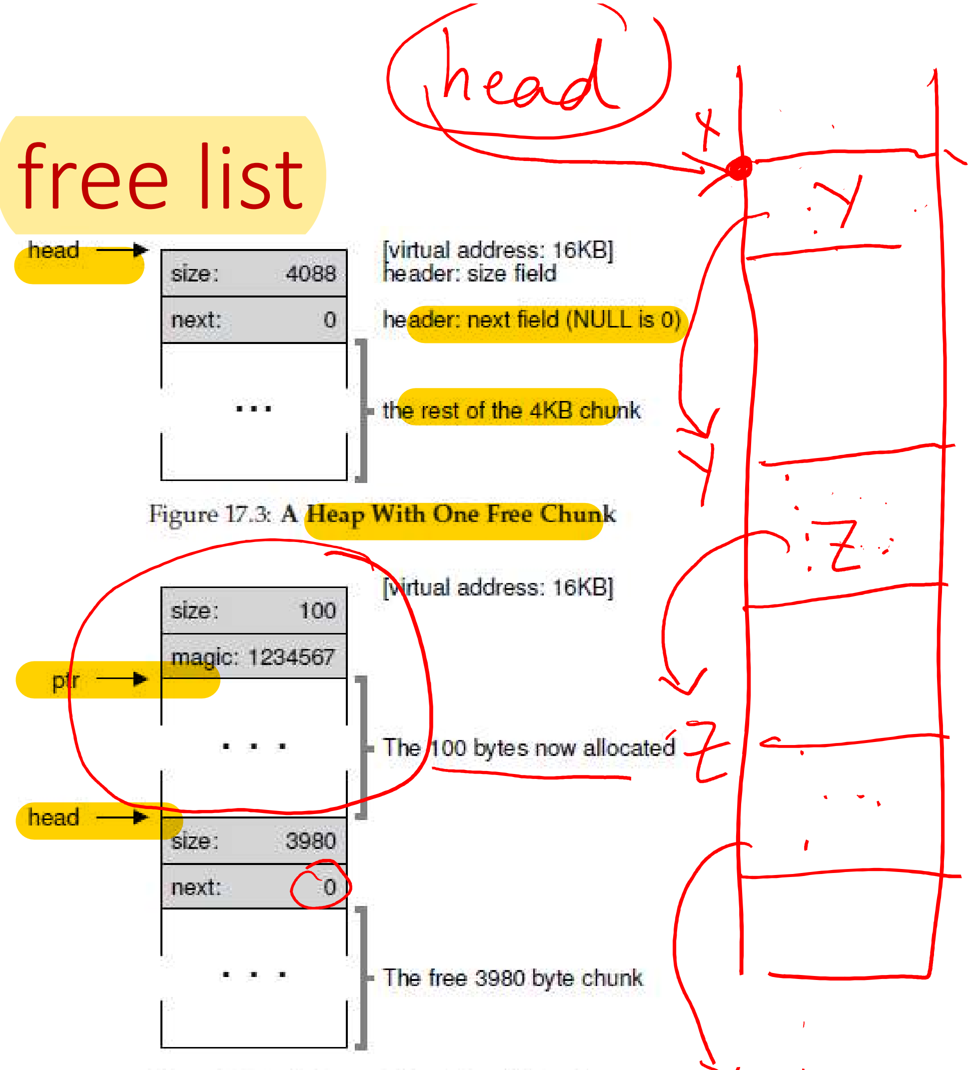
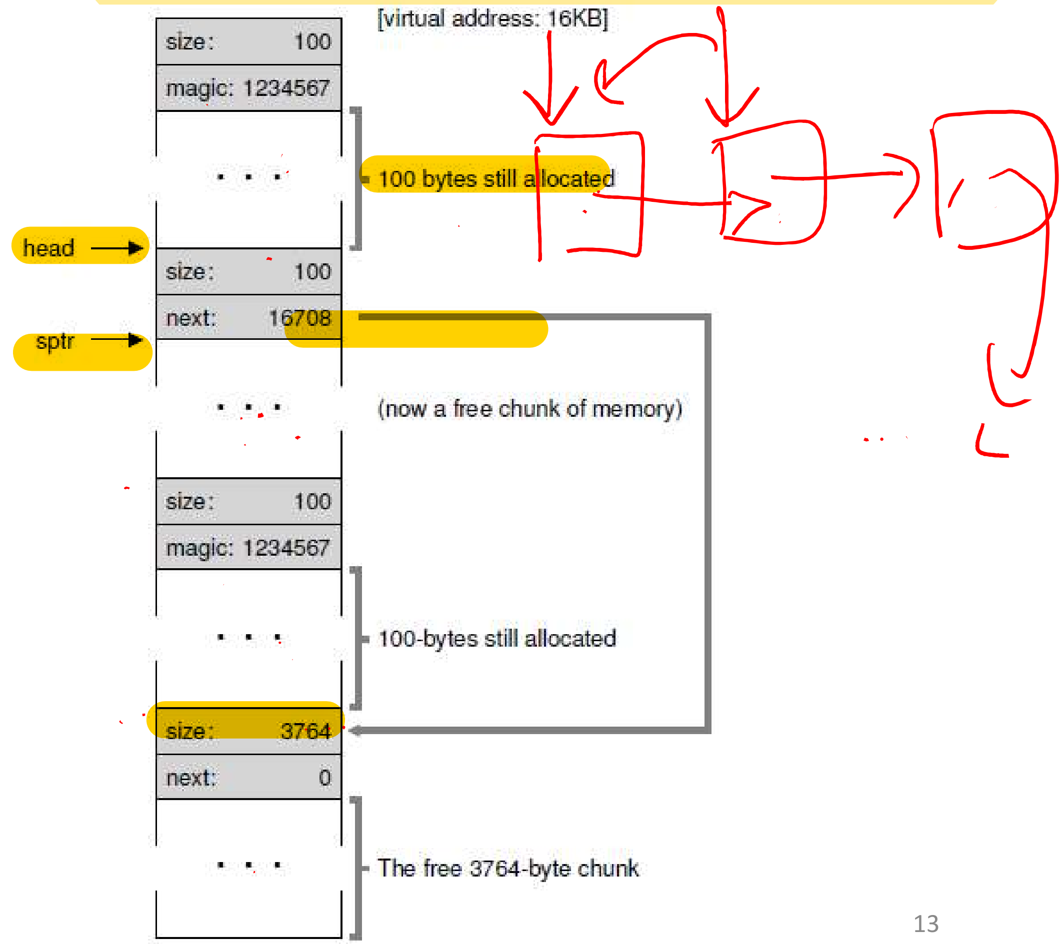


Figure 17.3: A Heap With One Free Chunk

Figure 17.4: A Heap: After One Allocation

Variable sized allocation: external fragmentation

- Suppose 3 allocations of size 100 bytes each happen. Then, the middle chunk pointed to by `sptr` is freed
- What is the free list?
  - It now has two non-contiguous elements
- Free space may be scattered around due to fragmentation
  - Cannot satisfy a request for 3800 bytes even though we have the free space



Variable sized allocation: splitting and coalescing

- Suppose all the three chunks are freed
- The list now has a bunch of free chunks that are adjacent
- A smart algorithm would merge them all into a bigger free chunk
- Must split and coalesce free chunks to satisfy variable sized requests

head

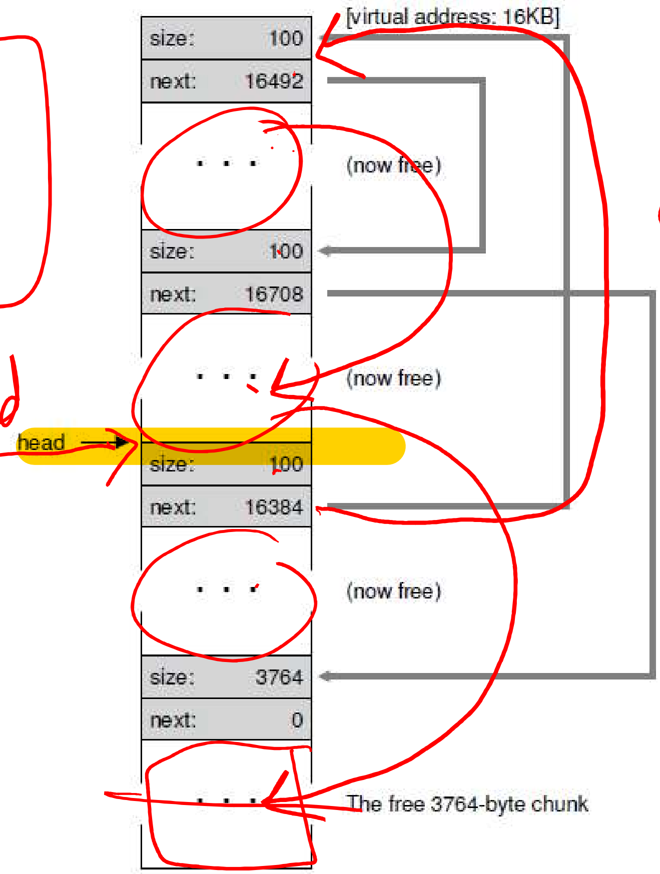
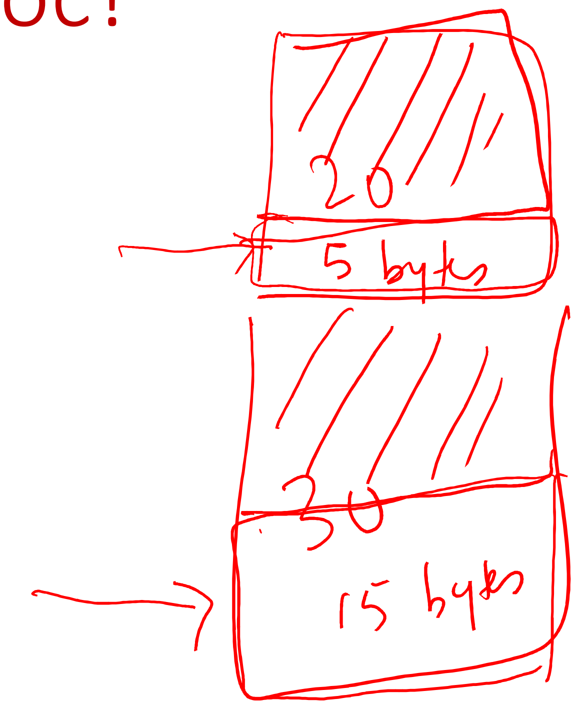


Figure 17.7: **A Non-Coalesced Free List**

Linked list

Which free chunk to pick in malloc?

- First fit: allocate first free chunk that is sufficient
- Best fit: allocate free chunk that is closest in size
- Worst fit: allocate free chunk that is farthest in size
- Example, consider this free list, and malloc(15)



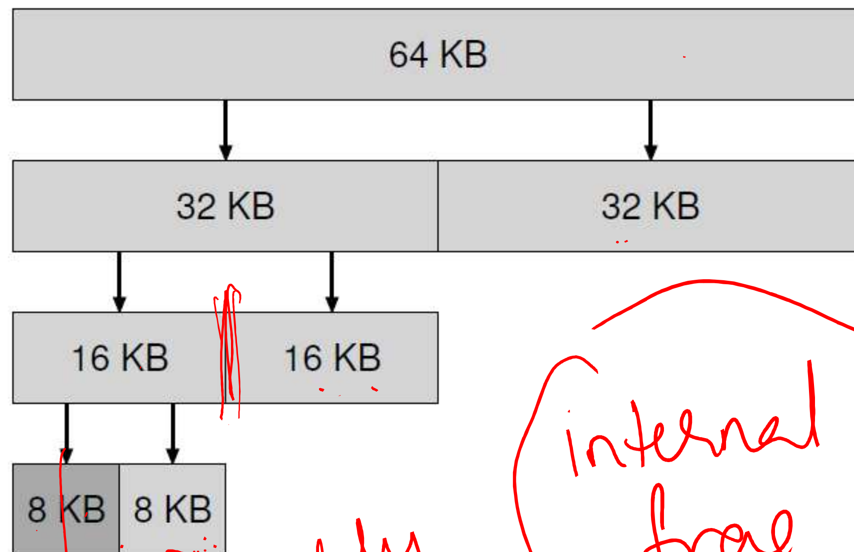
- Best fit would allocate the 20-byte chunk

- Worst fit would allocate 30-byte chunk: remaining chunk is bigger and more usable

Buddy allocation for easy coalescing

Slab
buddy

- Allocate memory in size of power of 2
  - E.g., for a request of 7000 bytes, allocate 8 KB cunk
- Why? 2 adjacent power-of-2 chunks can be merged to form a bigger power-of-2 chunk
  - E.g., if 8KB block and its "buddy" are free, they can form a 16KB chunk



buddy

internal
frag