Memory management system calls 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
- Demand paging: pages assigned physical frames only when used, else saved on swap space in disk

Recap: What happens on a memory access

- CPU accesses code/data at a certain virtual address (VA)
- MMU translates VA to PA, memory accessed using PA
  - If TLB hit, PA directly available, access memory directly
  - If TLB miss, MMU walks page table to compute PA, then access memory
- If any error during memory access, MMU traps to OS
  - If page fault due to frame not being allocated to the page, OS allocates free memory frame, updates page table entry, restarts process
  - Other types of page faults handled suitably
- Page replacement policy decides which pages to evict to swap and which to keep in memory frames

Memory allocation from user point of view

- So far, memory management from OS/hardware point of view
  - How OS manages memory, what data structures used, what happens on memory access, how is address translated, how is page fault handled
- Next: memory management from user program's view point
  - How is memory allocated to store user data in programs?
  - What system calls are available to user process for managing its memory?
  - How can a process request for more memory from OS as it runs?

Memory allocation to user processes

- When process created, OS assigns some memory to store memory image of process (code/data from executable, stack, heap, ..)
  - Stores allocation info in page table, given to MMU for address translation
- With demand paging, OS may not allocate memory for all pages initially, assigns frames only when process runs and accesses memory
  - Too little initial memory allocation → page faults when process starts
  - Too much initial memory allocation → possible waste of memory
- Memory allocation from OS to user processes always at page granularity (one or more pages)
- Within userspace, libraries can allocate memory in smaller chunks
  - C library function malloc assigns smaller chunks of memory from heap

Memory allocation system calls

- Address space created during fork, changed during exec
- No system calls needed to allocate memory that is already part of address space but unallocated by OS
  - Valid page but not present in RAM: automatically allocated memory by OS during page fault handling
- System calls to add pages, expand virtual address space
  - sbrk/brk system call adds pages to increase "program break", i.e., ending virtual address of heap
  - mmap system call adds virtual pages at any address, not just at end of heap
- Stack expansion automatically handled by OS (stack pointer crosses existing stack page, page fault occurs) Doubt? How?

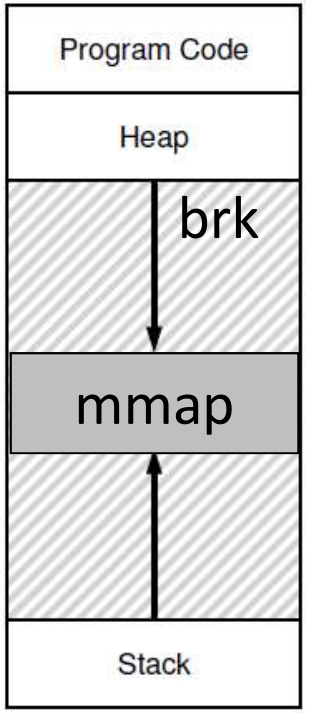


Image credit: OSTEP

System call mmap

```
char *buf = mmap(start, length, ..)
buf[0] = …

munmap(start, length)
```

- mmap system call used for mapping new memory into virtual address space of a process
- OS allocates new valid pages to address space, preferably at the requested starting virtual addresses
  - Physical memory allocated on demand
- Syscall returns the starting virtual address of the allocated chunk, used to read/write into the memory
- Memory contiguous in virtual address space, not necessarily in physical address space
- Correspondingly unmapped by munmap system call

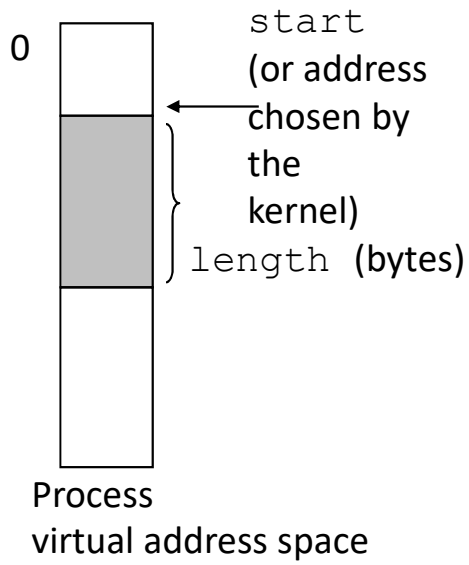


Image credit: CSAPP

More on memory mapping

- Two ways of memory mapping
  - Memory map an anonymous page, initialized to zero (e.g., heap page)
  - Memory map a file-backed page, initialized with file contents (e.g., code page)
- Physical memory corresponding to the virtual addresses allocated on demand in both cases
  - User accesses page, page fault, OS allocates and initializes physical frames
- How is memory-mapped page used?
  - Anonymous pages can be split into smaller chunks by heap manager, or can directly store user data
  - File-backed pages can be used to read/write files

Arguments to mmap

```
#include <sys/mman.h>
void *mmap(void *start, size_t length, int prot, int
flags, int fd, off_t offset);
```

- Starting virtual address (preference)
- Length of memory mapped chunk
- Protocol, i.e., permission bits (read, write)
- Flags (whether private or shared)
  - Shared implies changes by one process visible to another, not so with private
- File descriptor and offset if we are memory mapping a file into virtual address space (e.g., executable code or shared library object)

Heap vs Stack

- Program executable already allocates space for compile-time data (global and static local variables)
- Local variables in a function stored on stack

```
void func() {
    int x; // declares an integer on the stack
    ...
}
```

- Memory allocated dynamically via malloc stored on heap (the virtual address of allocated chunk on heap may be stored in a local variable on stack)

```
void func() {
    int *x = (int *) malloc(sizeof(int));
    ...
}
```

Image credit:OSTEP

Memory allocation via malloc

- Normally user programs don't need to call brk/sbrk/mmap functions to dynamically allocate memory at run time
- User programs calls C library functions like malloc to allocate chunks of memory from heap
- C library implementation of malloc uses brk/sbrk/mmap to obtain memory pages from OS when heap runs out of space
- User program can also directly allocate page-sized memory using mmap system call, instead of using malloc

Common errors with malloc

- Must allocate memory of the correct size before use

```
char *src = "hello";
char *dst;          // oops! unallocated
strcpy(dst, src); // segfault and die
```

```
char *src = "hello";
char *dst = (char *) malloc(strlen(src) + 1);
strcpy(dst, src); // work properly
```

- Allocated memory must be explicitly freed by user, else memory leak
  - Some language libraries automatically clean unused chunks via garbage collection algorithms

```
int *x = malloc(10 * sizeof(int));
...
free(x);
```

Image credit:OSTEP

Types of memory allocators

- General-purpose heap allocators 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 allocators optimize for fixed size allocation
  - 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
- Choice of allocator managed by programming language libraries, but can be configured/changed by users

Variable sized memory allocation

- How do we design a general purpose memory allocator?
- Given a large block of memory (one or more pages), how do we use it to satisfy variable-sized memory allocation requests?
- Challenge: need to track information about variable sized chunks, which chunks are allocated and which are free
- External fragmentation: after memory is allocated and freed-up, many gaps of different sizes throughout heap

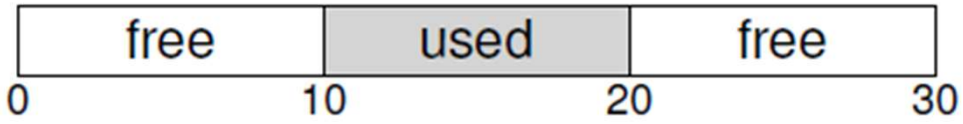


Image credit:OSTEP

Free list

- Free list: data structure (e.g., linked list) to track free chunks in heap
- Initialized with entire heap, changes with alloc/free operations
- Malloc searches free list to find a suitable chunk
- Freed up chunks added back to free list
- Example: 30 bytes heap, three chunks of 10 types each allocated, two chunks freed up

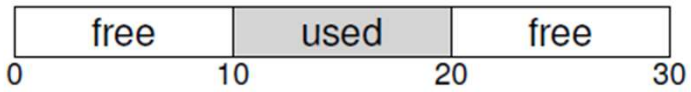


Image credit:OSTEP

Splitting and coalescing

- Exact size free chunk needed for alloc may not be available
- May need to split a bigger chunk, e.g., request to allocate 1 byte, split 10 byte chunk into 1 byte and 9 byte chunks

- May need to coalesce adjacent free chunks into bigger chunk, e.g., once middle 10 byte chunk is also free, can coalesce all free chunks into one 30 byte chunk, to satisfy requests for larger than 10 bytes

Image credit:OSTEP

Which free chunk to pick in malloc?

- First fit: allocate first free chunk that fits
- 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 request for malloc(15)

- Best fit allocates the 20-byte chunk

- Worst fit allocates 30-byte chunk, remaining chunk is bigger and more usable

Image credit:OSTEP

Headers

- Allocated chunk stores some metadata (header), not used by user calling malloc
- Header stores size of allocation
  - Why? We should know how much to free later
- Header also has some random number (“magic”)
  - Detects if previous chunk has overflown into this chunk

```
typedef struct __header_t {
    int size;
    int magic;
} header_t;
```

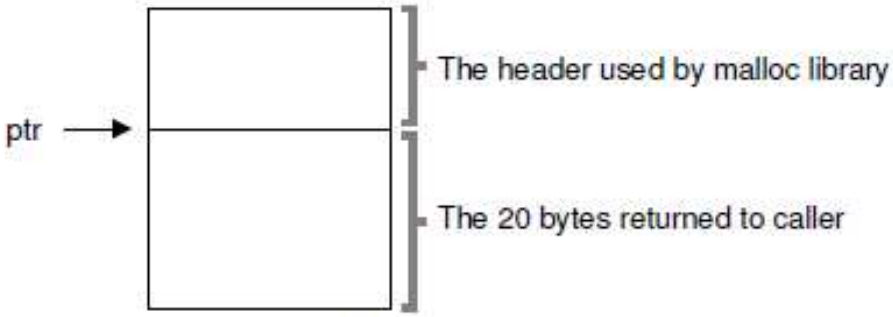


Figure 17.1: **An Allocated Region Plus Header**

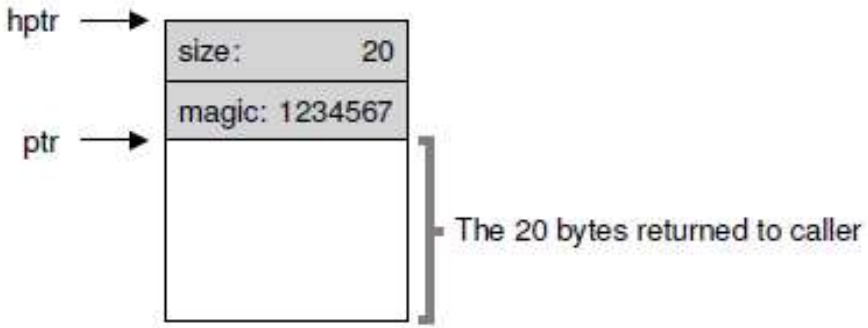


Figure 17.2: **Specific Contents Of The Header**

Image credit:OSTEP

Embedded free list

- Where is free list stored? Information can be embedded within free chunk itself
- Pointer to the next free chunk is embedded within the free chunk
- Need to only remember head of list within allocator, e.g., allocations happen from the head, free chunks added back to head
- Example: 100 bytes allocated on heap

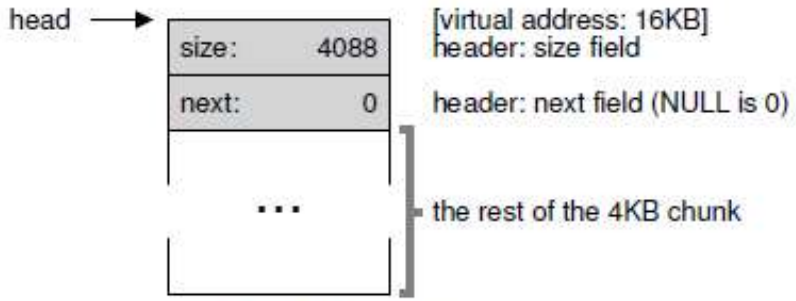


Figure 17.3: A Heap With One Free Chunk

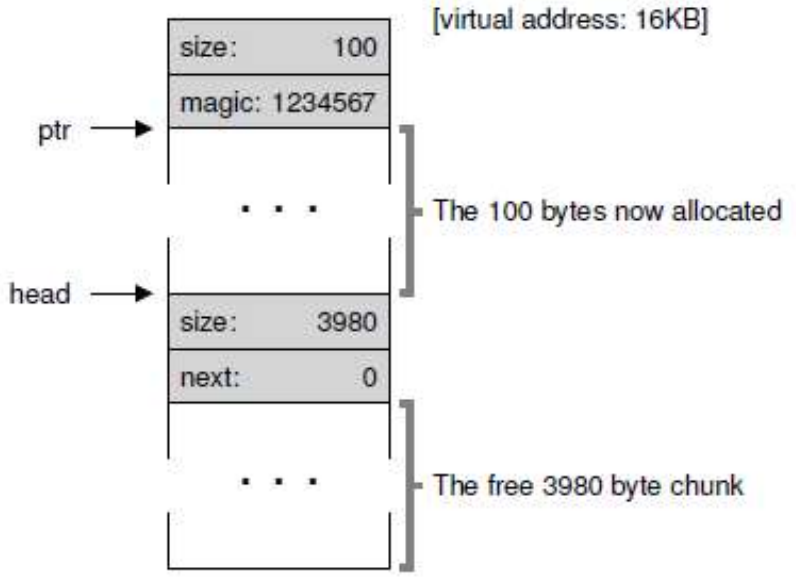


Figure 17.4: A Heap: After One Allocation

Image credit:OSTEP

requested bytes along with the header space is kept occupied.

Example (contd)

- Heap has 4088 bytes initially (4KB = 4096, less 8 byte header)
- Three calls to malloc for 100 bytes each
- Each call allocates 108 bytes on heap (100 bytes to user, 8 byte header)
- Free list has one chunk left of 3764 bytes

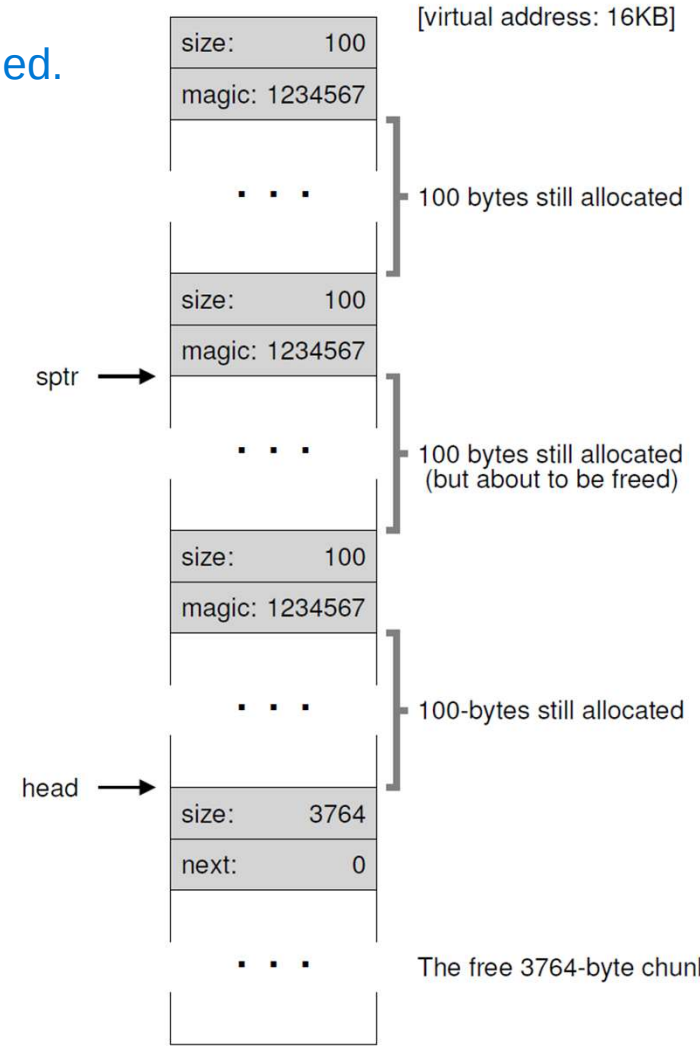


Figure 17.5: **Free Space With Three Chunks Allocated**

Image credit:OSTEP

Example (contd)

- Now the middle chunk pointed to by `sptr` is freed
- What is the free list now? Two non-contiguous chunks of 100 bytes each
- Free space scattered due to external fragmentation
  - Cannot satisfy a request for 3800 bytes even though we have the free space in heap

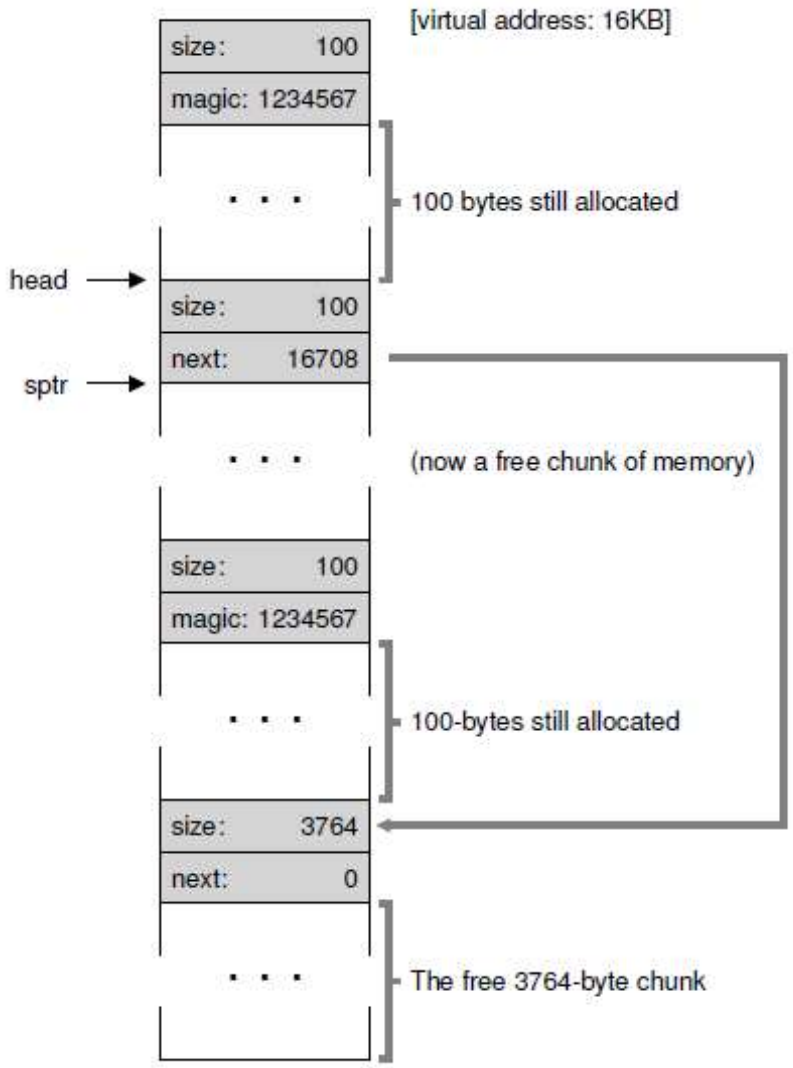


Image credit:OSTEP

Example (contd)

- 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
- Lot of work to perform variable sized memory allocation

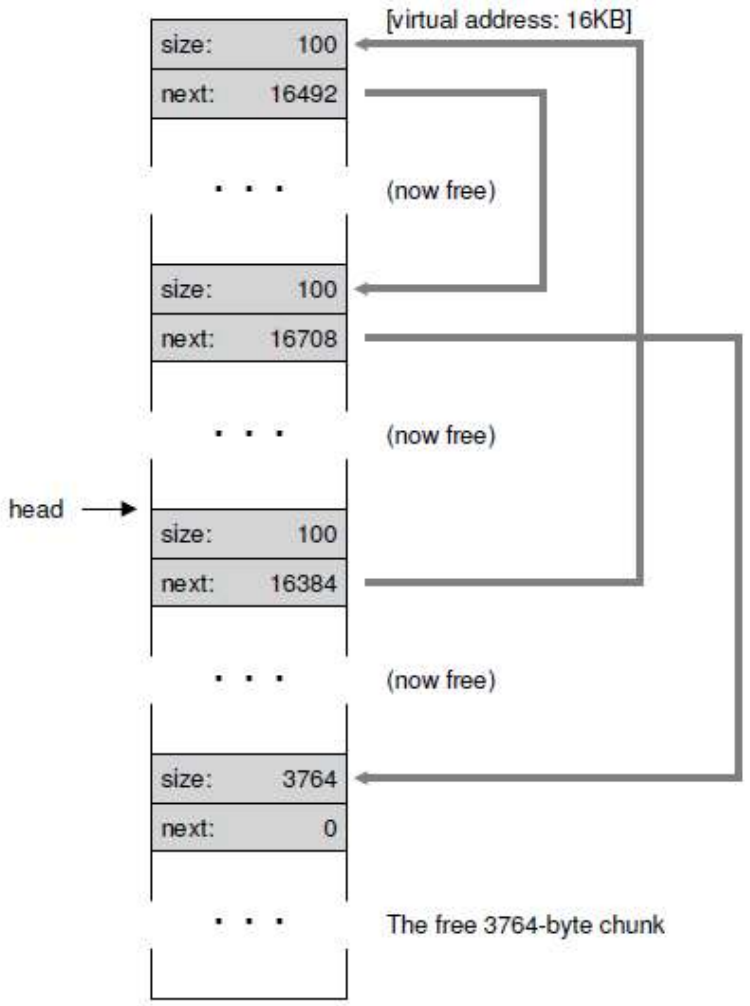


Figure 17.7: **A Non-Coalesced Free List**

Image credit:OSTEP

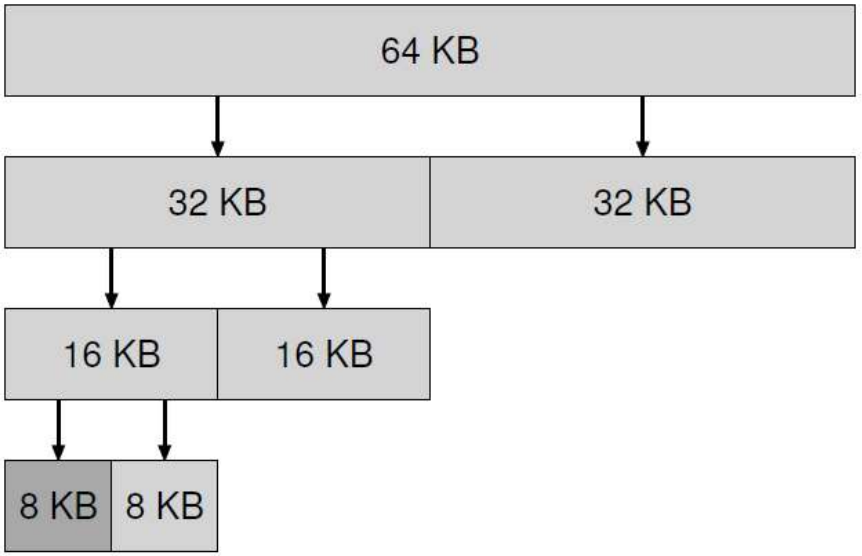
Segregated free lists and slab allocators

- One idea for efficiency: maintain separate free lists for free chunks of different sizes
  - Useful when application calls malloc with popular chunk sizes
  - Requests that cannot be satisfied by segregated lists are handled by variable sized allocator
  - Less worry about external fragmentation, no need to split/coalesce
- Slab allocator: slab of memory with chunks of a particular size pre-allocated, easily available for use
  - Even more efficient than segregated free list, avoids alloc/free overheads

You just have to use it, no overhead of alloc/free.

Buddy allocation for easy coalescing

- 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



High disadvantage of internal fragmentation.

Image credit:OSTEP