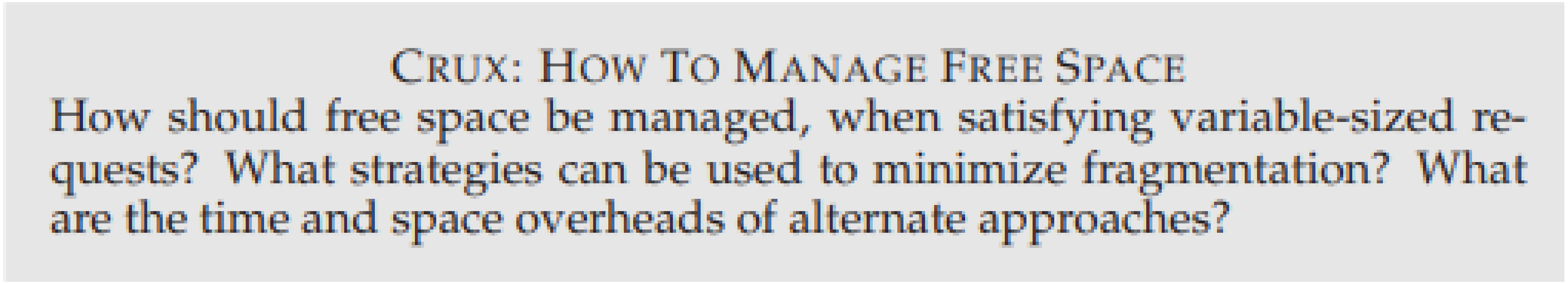
Chapter 17: Free-Space Management

Adam Disney

THE UNIVERSITY OF
TENNESSEE
KNOXVILLE

Crux: How To Manage Free Space



Crux: How To Manage Free Space

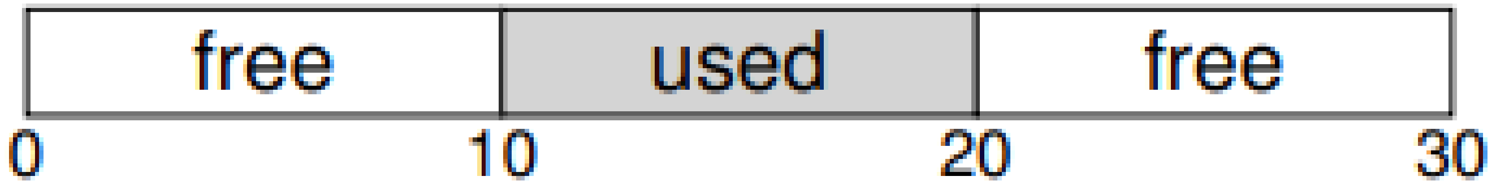
How should free space be managed, when satisfying variable-sized re-quests? What strategies can be used to minimize fragmentation? What are the time and space overheads of alternate approaches?

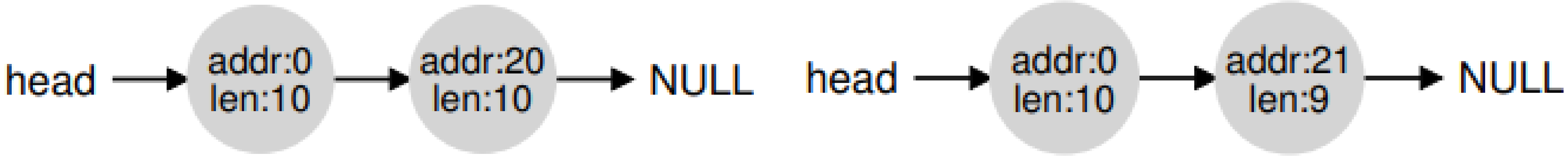
Assumptions

- Malloc()
  - Takes a single argument, size
  - Returns void *
- Free()
  - Takes a single argument, pointer returned by malloc()
  - This means the library must be able to solve memory size
- Need a data structure to keep up with free space.
- Focus on external fragmentation
- No compaction of free space
- Heap is a single fixed size

Low-level Mechanisms - Splitting

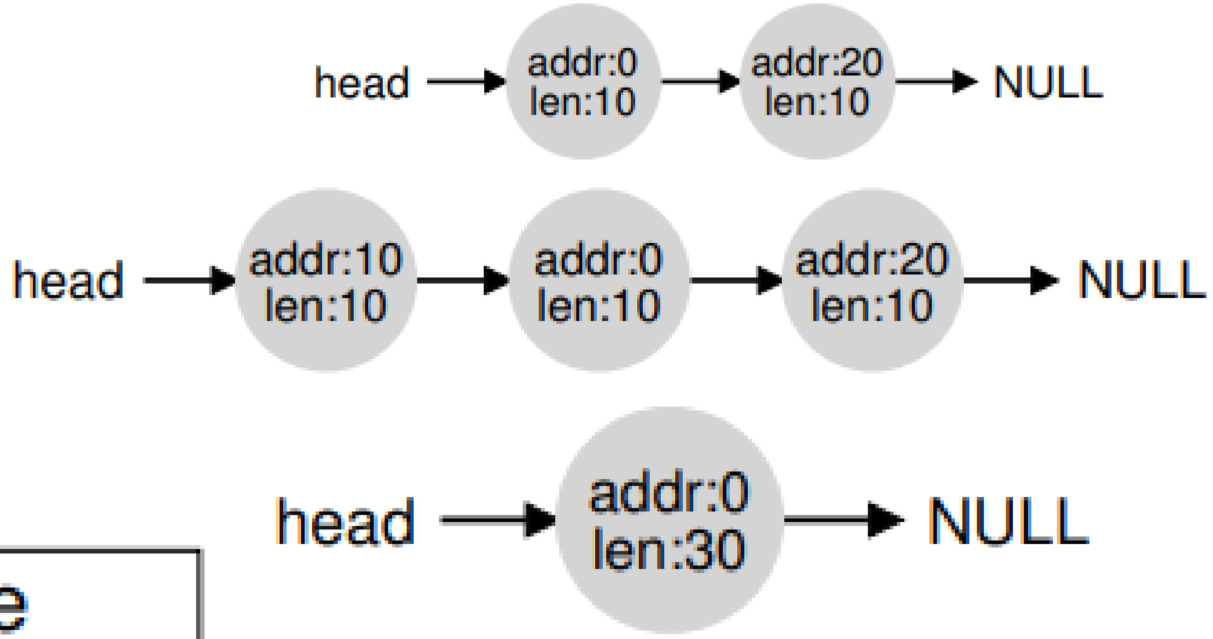
- When we malloc() for less than a free chunk of memory, we split it. Suppose we ask for 1 byte.

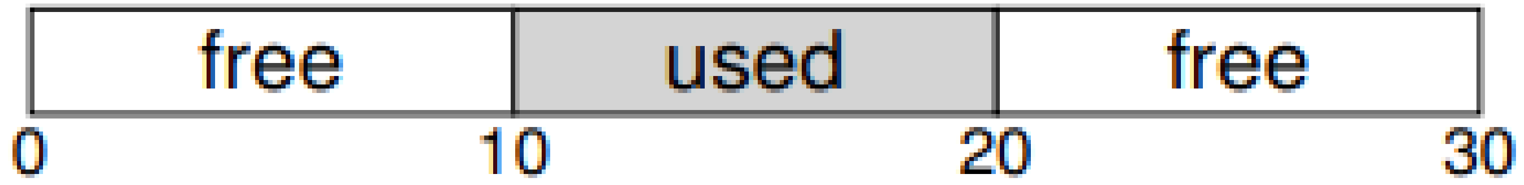




Low-level Mechanisms – Coalescing

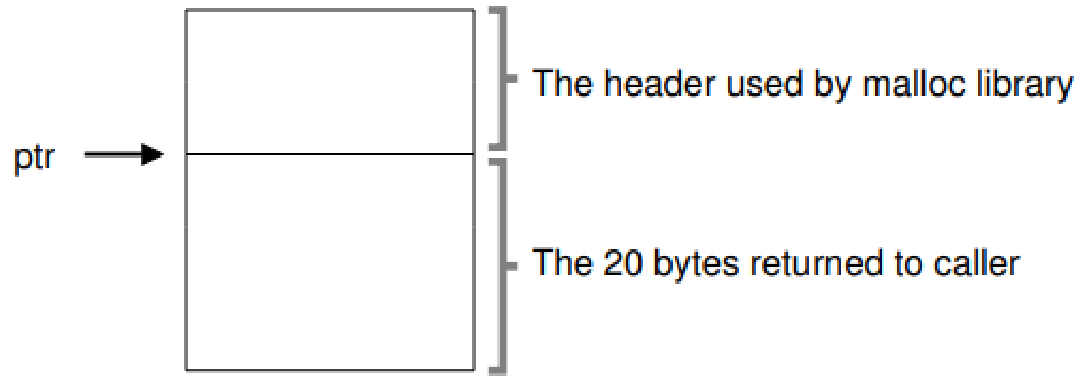
- When we free() memory, we might coalesce the free space.





Low-level Mechanisms
Tracking The Size Of Allocated Regions

- When free() is called, how does the library know the size?
- Magic is to detect corruption quickly.
- Notice this means a malloc(n) needs to find a chunk of n + sizeof(header) bytes.



ptr
The header used by malloc library
The 20 bytes returned to caller

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

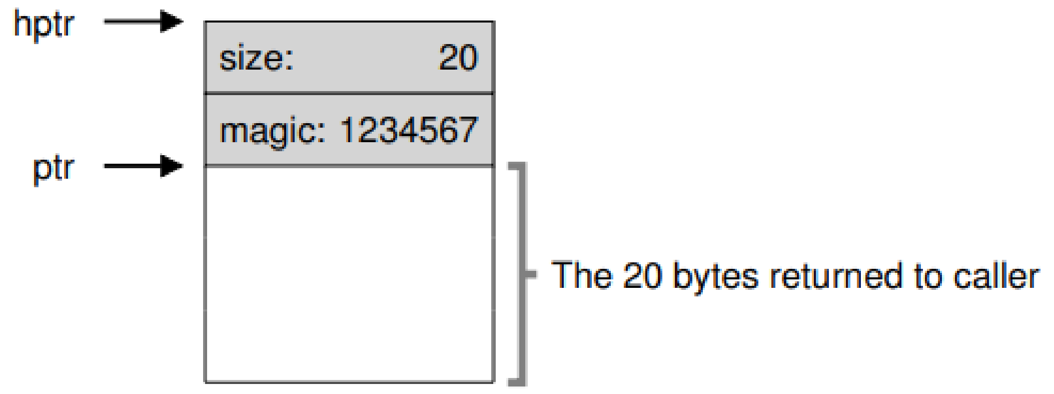
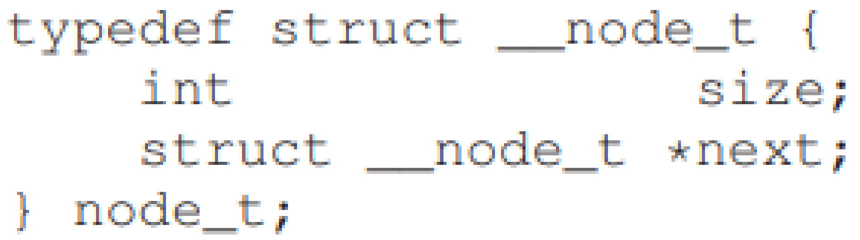


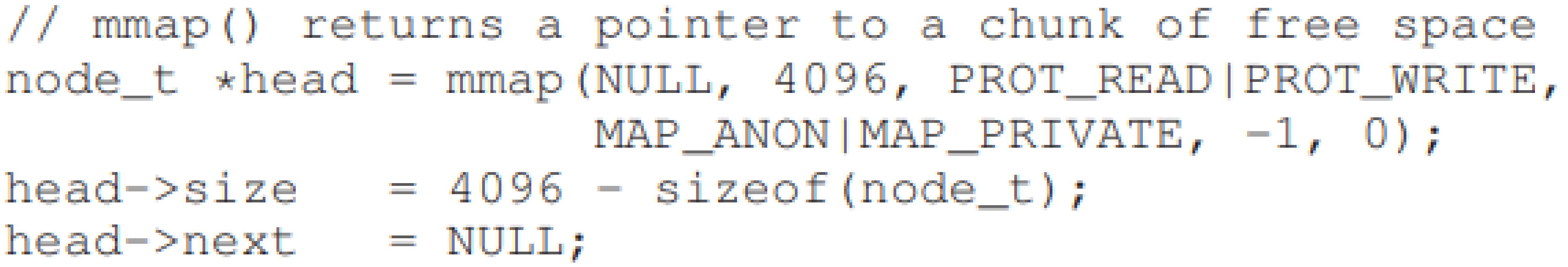
Figure 17.2: **Specific Contents Of The Header**

Low-level Mechanisms
Embedding A Free List

- Since the library is the one implementing malloc()/free(), it cannot use them itself! Instead, it will build the list inside the free space itself.



```
typedef struct __node_t {
    int              size;
    struct __node_t *next;
} node_t;
```



```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size   = 4096 - sizeof(node_t);
head->next   = NULL;
```

Low-level Mechanisms
Embedding A Free List

- Since the library is the one implementing malloc()/free(), it cannot use them itself! Instead, it will build the list inside the free space itself.

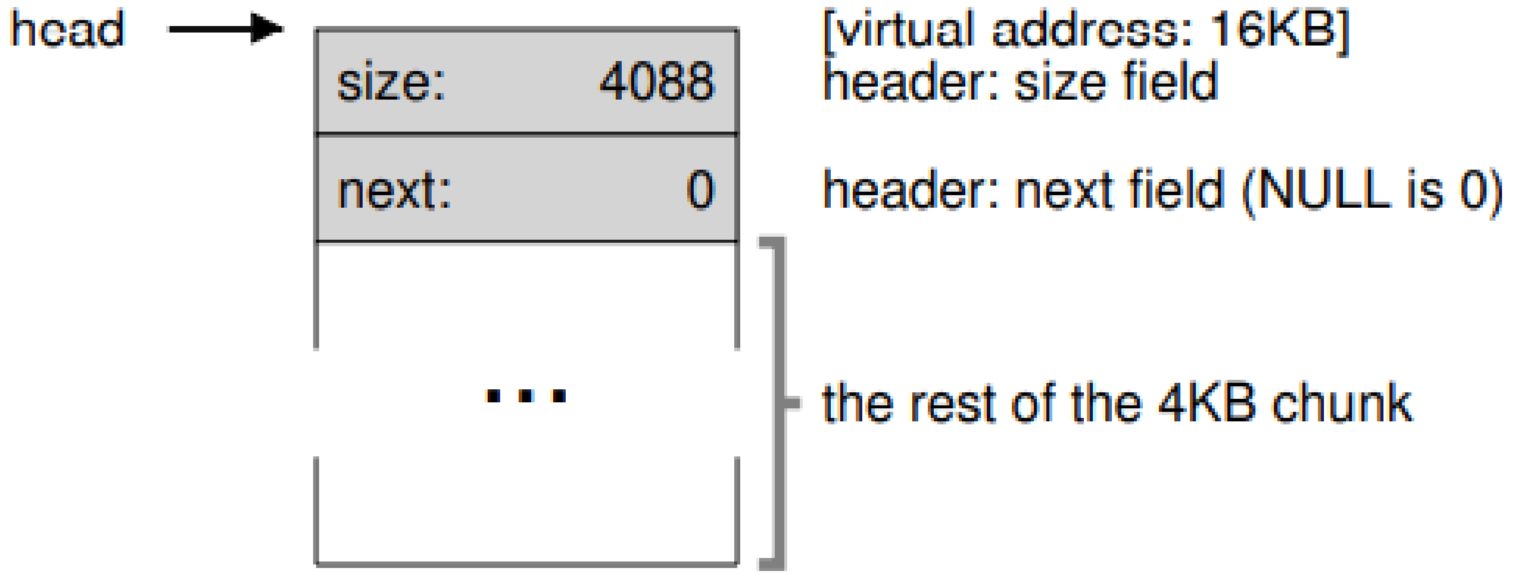


Figure 17.3: **A Heap With One Free Chunk**

Low-level Mechanisms
Embedding A Free List (Malloc)

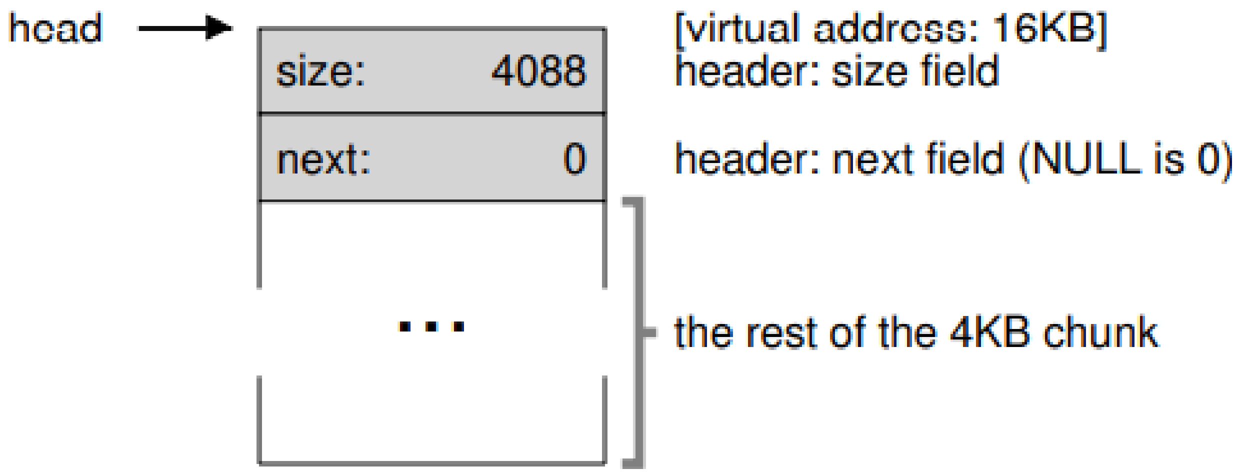


Figure 17.3: **A Heap With One Free Chunk**

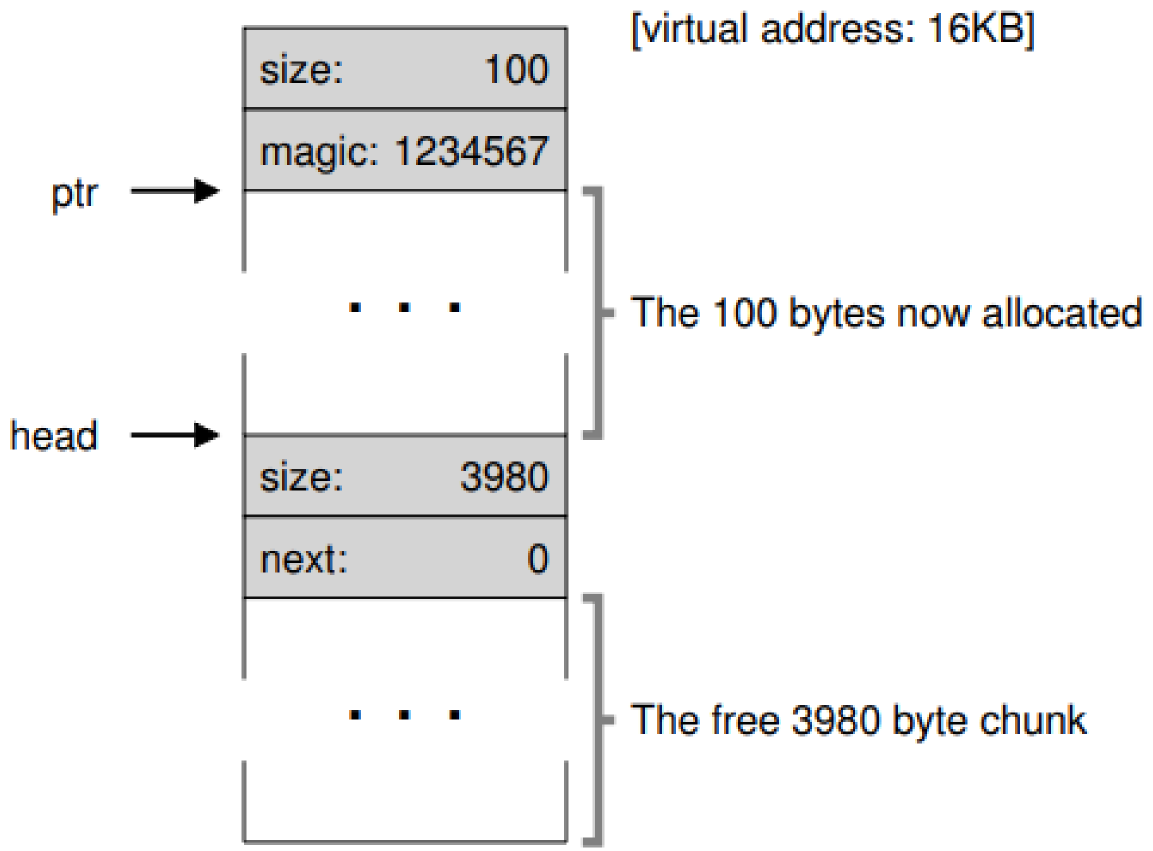


Figure 17.4: **A Heap: After One Allocation**

Low-level Mechanisms Embedding A Free List (Free)

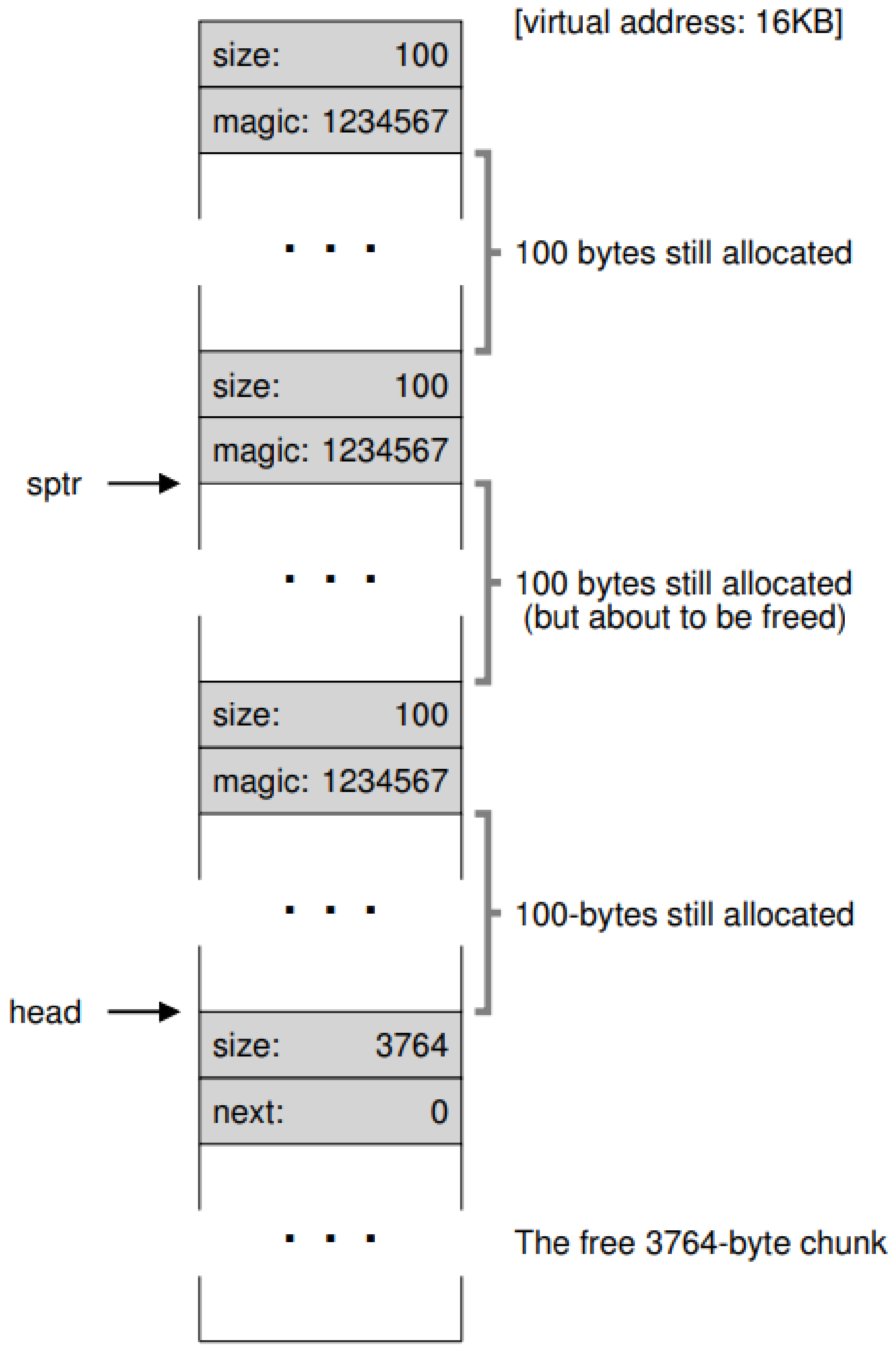


Figure 17.5: **Free Space With Three Chunks Allocated**

Low-level Mechanisms Embedding A Free List (Free)

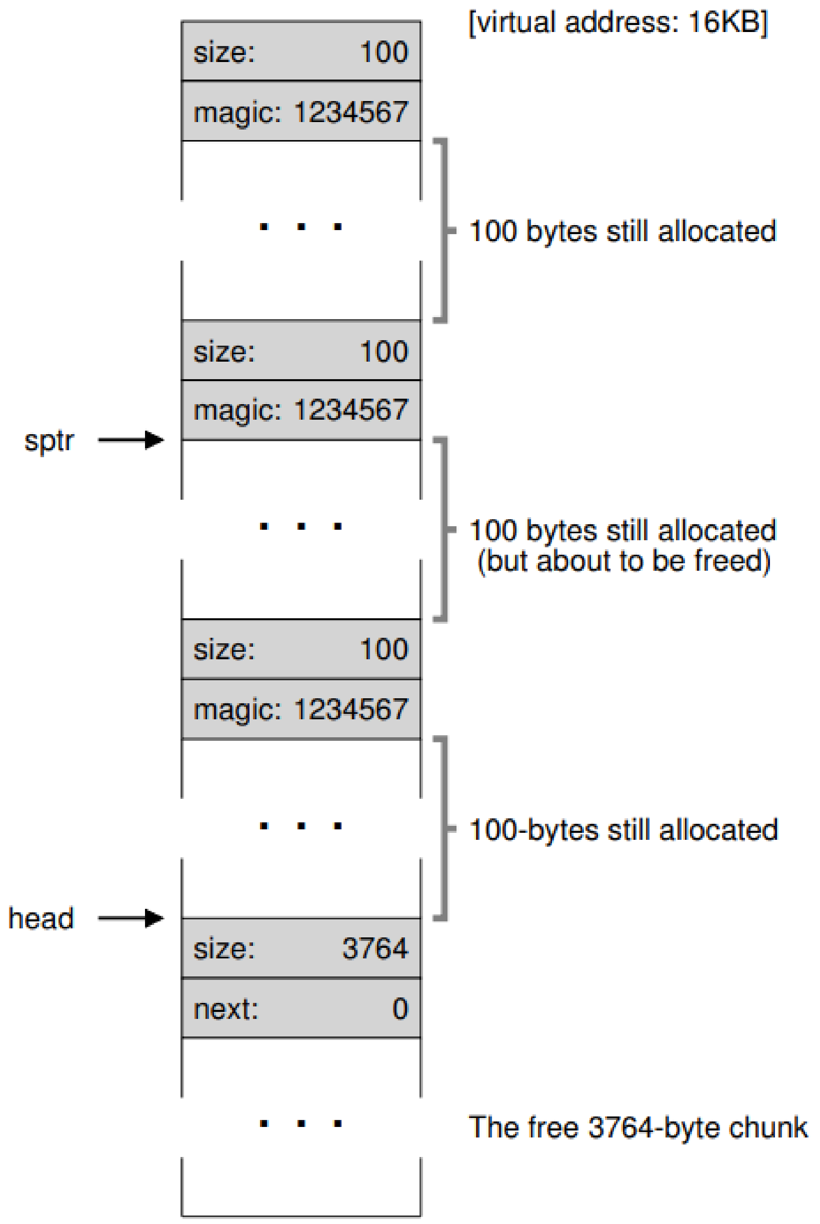


Figure 17.5: **Free Space With Three Chunks Allocated**

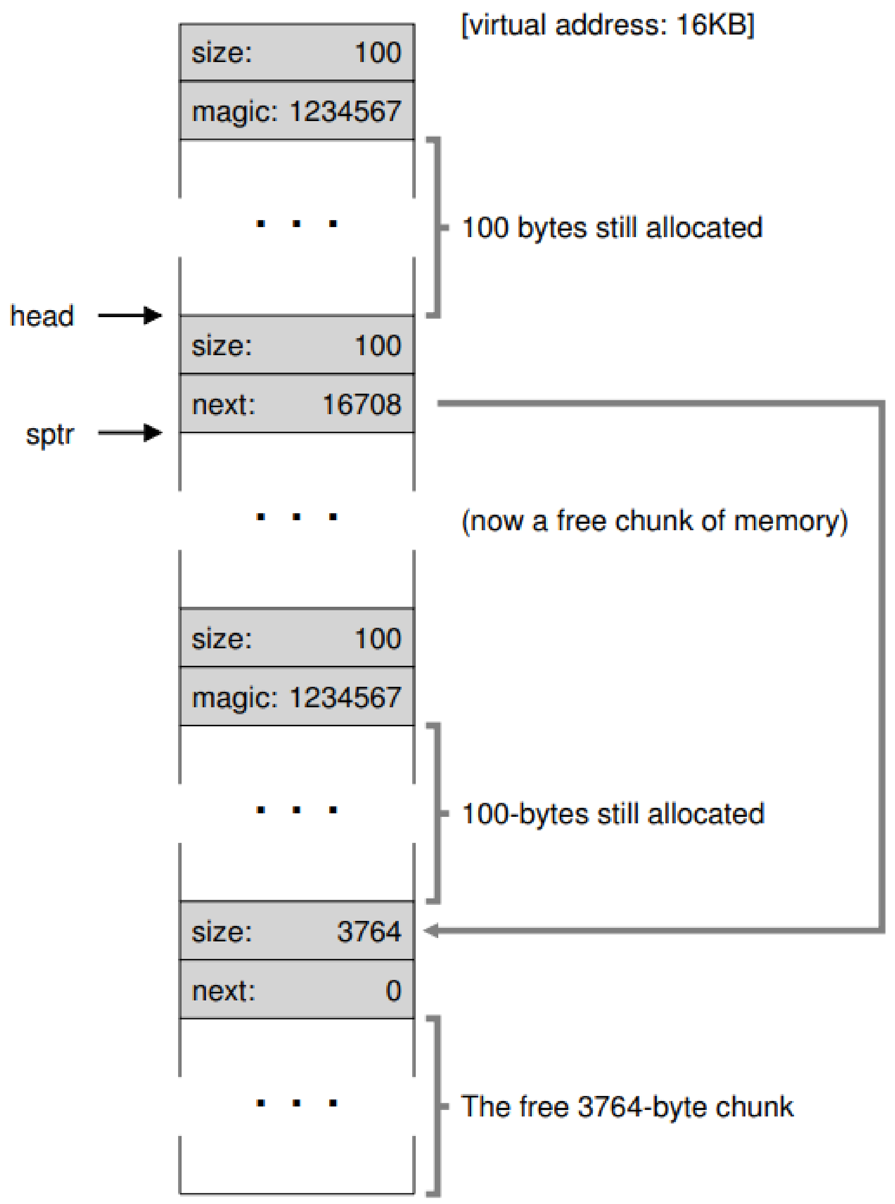


Figure 17.6: **Free Space With Two Chunks Allocated**

Low-level Mechanisms
Embedding A Free List (Free)

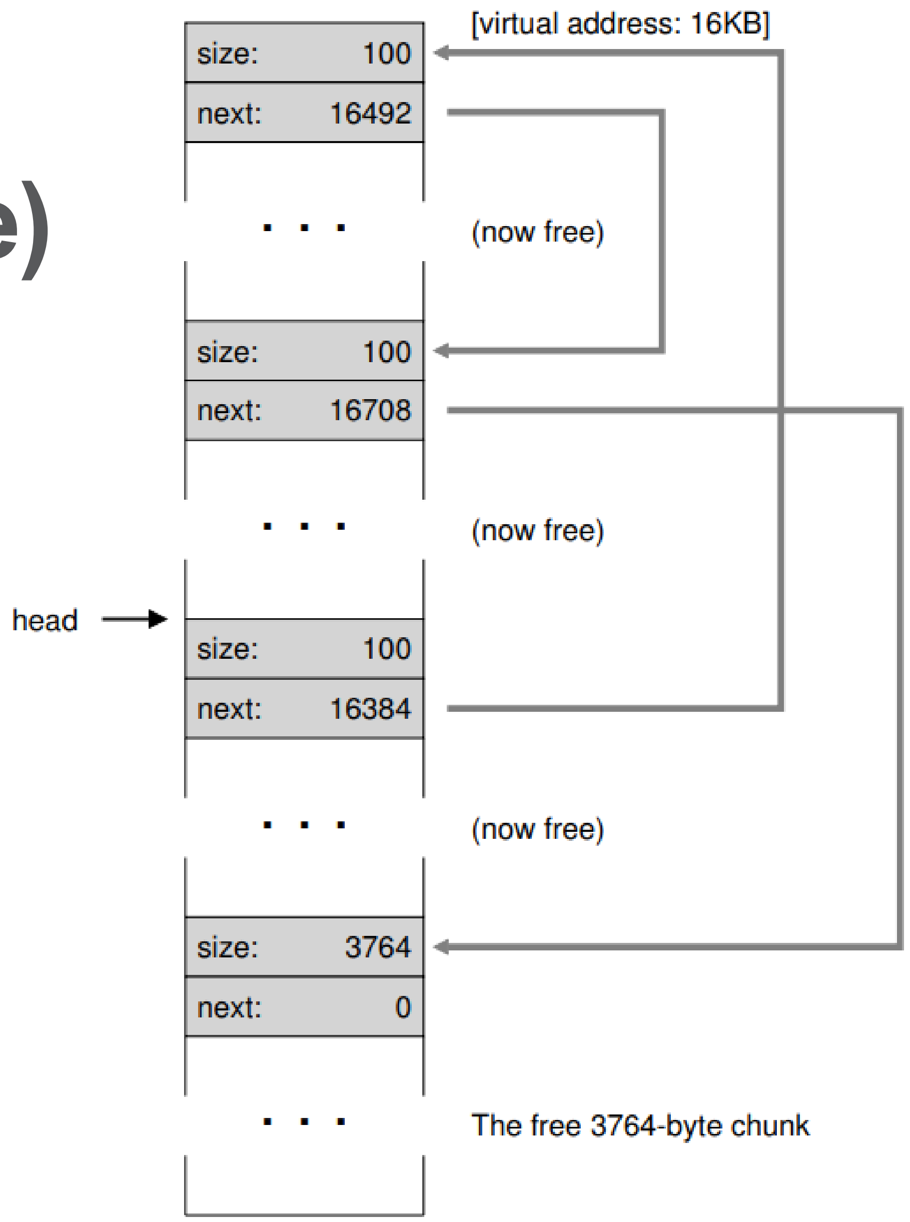


Figure 17.7: **A Non-Coalesced Free List**

Low-level Mechanisms
Growing The Heap

- What if the heap runs out of space?
  - Ask the OS to extend the heap (sbrk())

Basic Strategies (Policies)
Best Fit

- Search the list for the smallest chunk that can fulfill the request.
- Can be expensive to exhaustive search for the correct free chunk.

Basic Strategies (Policies)
Worst Fit

- Search the list for the largest chunk that can fulfill the request.
  - This tries to leave big chunks free instead of lots of small chunks.
- Can be expensive to exhaustive search for the correct free chunk.

Basic Strategies (Policies)
First Fit

- Pick the first chunk that can service the request.
- This is fast but can pollute the free list with small objects.
- Often paired with address-based ordering of the list to make coalescing easier and reduce fragmentation.

Basic Strategies (Policies)
Next Fit

- Instead of searching from the beginning of the list every time, we remember where we last looked in the list.
- Now from this point we pick the first chunk that fits.
- This avoids splintering the beginning of the list while giving similar performance to first fit.

Other Approaches (Policies) Segregated Lists

- If a particular application has one or a few popular-sized requests, keep a list just to manage objects of that size.
- Other requests go to a general allocator.
- Benefits:
  - Allocating the popular size is quick and easy.
  - Fragmentation much less of a concern.
- Issues:
  - How much memory do we dedicate to this pool of memory?

Other Approaches (Policies) Segregated Lists

- Slab allocator:
  - Designed for Solaris kernel
  - Object caches made at boot up for common kernel structures
    - Locks, inodes, etc.
  - When a cache runs low, it asks for more memory from the general allocator. (slab of memory)
  - When the reference count to the slab hits zero, the general allocator can reclaim it.
  - Can also do something like leaving free objects in an initialized state.

Other Approaches (Policies)
Buddy Allocation

- Since coalescing is critical, some approaches are designed around making it simple.
- Binary buddy allocator views free memory as one big space of size 2^N
- On allocation request, recursively divide the free space in half until we reach the smallest size that will service the request.
- Internal fragmentation is an issue because it can only give out a power-of-two-sized block.

Other Approaches (Policies) Buddy Allocation

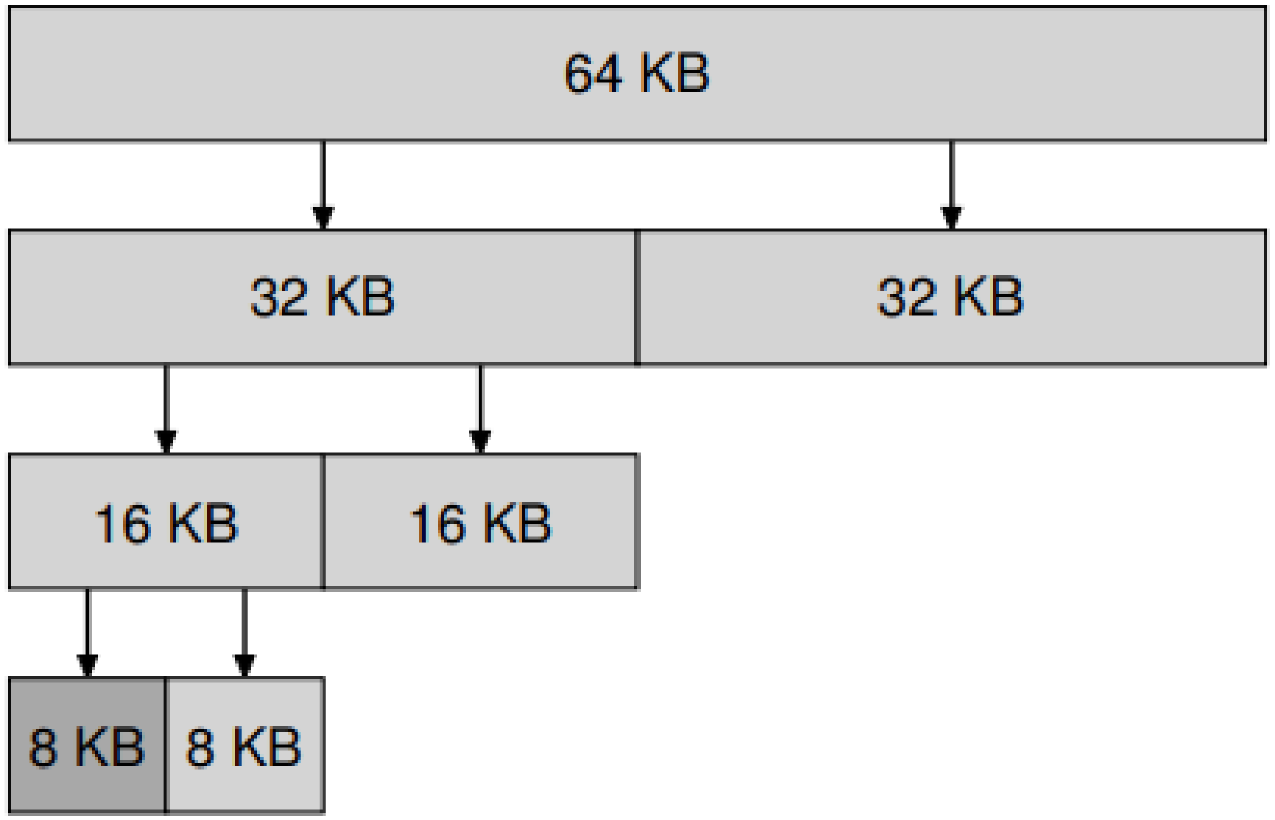


Figure 17.8: **Example Buddy-managed Heap**

Other Approaches (Policies)
Buddy Allocation

- When a block is freed, the allocator checks whether the "buddy" is free. If so, it coalesces the two then recursively checks if that block's "buddy" is also free continuing until a "buddy" is found to be in use.