
For this problem, assume that physical memory can hold at most four pages. What pages remain in memory at the end of the following sequence of page table operations and what are the use bits set to for each of these pages?

Page	A	B	C	A	C	D	B	D	A	E	F
------	---	---	---	---	---	---	---	---	---	---	---

The clock hand only advances on page faults.

No page replacement occurs until $t = 10$, when all pages are full. At $t = 10$, all pages have the use bit set.

The clock hand does a full sweep, setting all use bits to 0, and selects page 1 (currently holding A) to be paged out.

The clock hand advances and now points to page 2 (currently holding B).

At $t = 11$, we check page 2's use bit, and since it is not set, select page 2 to be paged out.

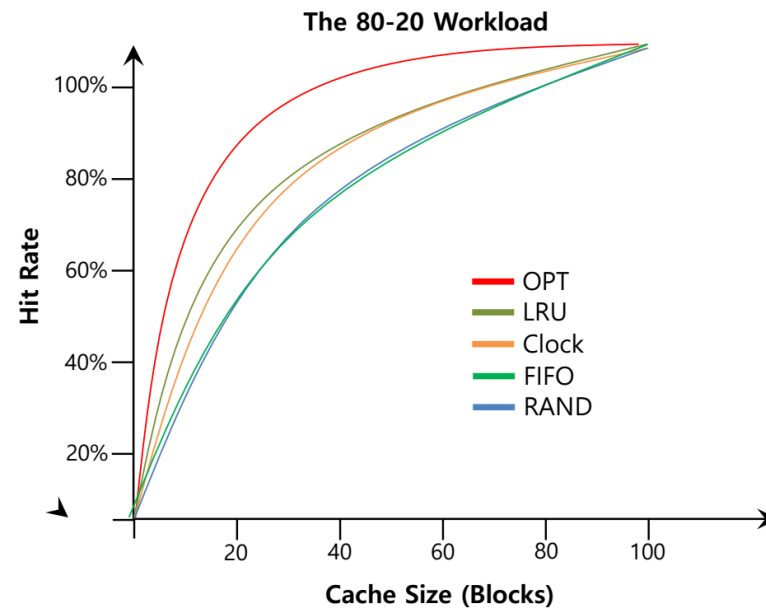
F is brought in to page 2. The clock hand advances and now points to page 3.

We reach the end of the input and end. Note: The table shows the clock hand position before page faults occur.

Page	A	B	C	A	C	D	B	D	A	E	F
1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	A: 1	E: 1	E: 1
2		B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 1	B: 0	F: 1
3			C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 1	C: 0	C: 0
4						D: 1	D: 1	D: 1	D: 1	D: 0	D: 0
Clock	1	2	3	4	4	4	1	1	1	1	2

Workload – Clock Algorithm

- Clock algorithm doesn't do as well as perfect LRU, it does better than approach that don't consider history at all.



Clock extensions

Replace multiple pages at once

Intuition: Expensive to run replacement algorithm and to write single block to disk

Find multiple victims each time and track free list

Add software counter (“chance”)

Intuition: Better ability to differentiate across pages (how much they are being accessed)

Increment software counter if use bit is 0

Replace when chance exceeds some specified limit



Nth chance: The clock algorithm is also called the "second-chance" algorithm, as the pages with the accessed bit equal to one, are given a second chance. A straight-forward extension to this is the Nth chance algorithm, wherein a page is given N chances before evicting it. Here is the Nth chance algorithm:

- With each page, OS maintains a counter to indicate the number of sweeps that page has gone through.
- On page fault, OS checks accessed bit:
 - If 1, then clear it, and also clear the counter.
 - If 0, then increment the counter; if count == N, replace page.
 - Large N implies better approximation to LRU, e.g., N = 1000 is a very good LRU approximation. However, a large N implies more work by the OS before a page can be replaced.
 - N = 1 implies the default clock algorithm.


Use dirty bit to give preference to dirty pages

Intuition: More expensive to replace dirty pages

Dirty pages must be written to disk, clean pages do not

Replace pages that have use bit and dirty bit cleared

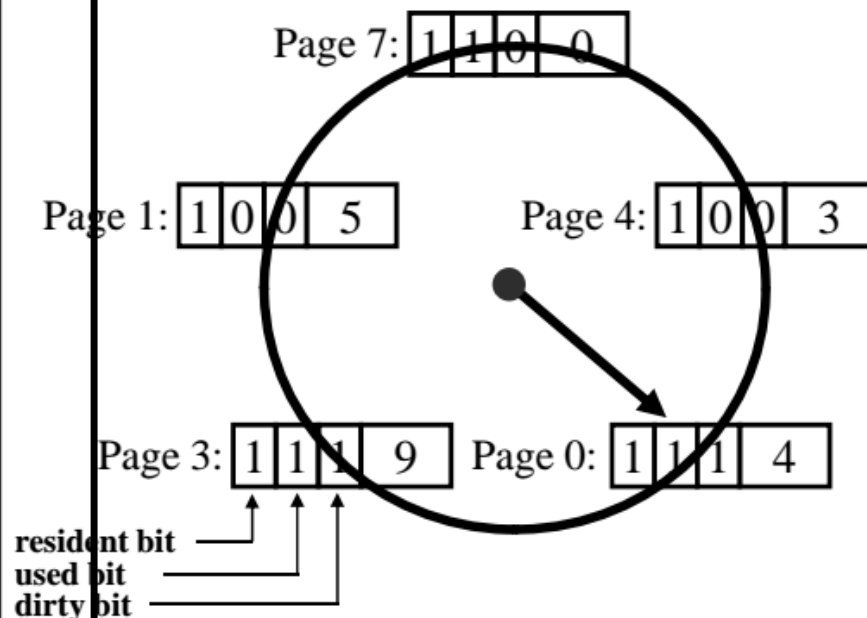
Algo could be modified to scan for pages that are both unused and clean to evict first; failing to find those, then for unused pages that are dirty, and so forth.



Optimizing Approximate LRU Replacement

The Second Chance algorithm

- There is a significant cost to replacing “dirty” pages
- Modify the Clock algorithm to allow dirty pages to always survive one sweep of the clock hand
 - Use both the *dirty bit* and the *used bit* to drive replacement



Second Chance Algorithm

Before clock sweep

used *dirty*

0	0
0	1
1	0
1	1

After clock sweep

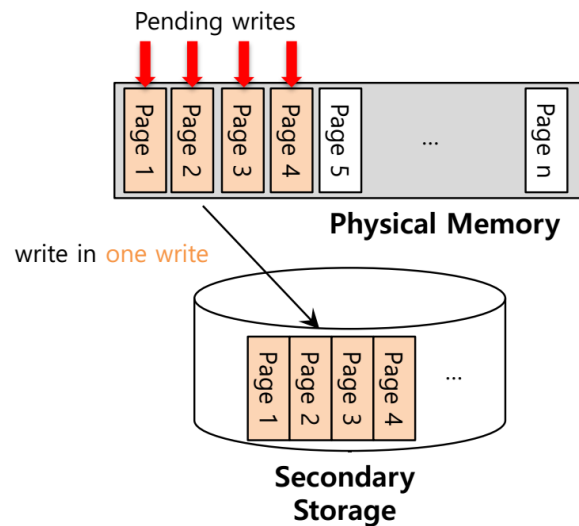
used *dirty*

replace page

0	0
0	0
0	0
0	1

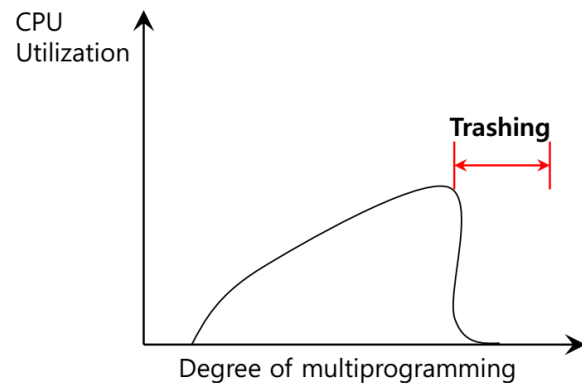
Clustering/Grouping

- Collect a number of **pending writes** together in memory and write them to disk in **one write**.
 - ◆ Perform a **single large write** more efficiently than **many small ones**.



Thrashing

- ❑ Memory is **oversubscribed** and the memory demands of the set of running processes **exceeds** the available physical memory.
 - ◆ Decide not to run a subset of processes.
 - ◆ Reduced set of processes working sets fit in memory.



Normally, if a thread takes a page fault and must wait for the page to be read from disk, the OS runs a different thread while the I/O is occurring.

But what happens if memory gets overcommitted? Suppose the pages being actively used by multiple processes do not all fit in the physical memory. Each page fault causes one of the active pages to be moved to disk, so another page fault is expected soon. The system will spend most of its time reading and writing pages instead of executing useful instructions. The OS wanted to provide the size of disk and the access time of main memory. However, at this point, it is providing the access time of disk. This situation is called *thrashing*; it was a serious problem in early demand paging systems.

Dealing with thrashing:

If a single process is too large for memory, there is nothing the OS can do. That process will simply thrash.

If the problem arises because of the sum of many processes:

- Figure out how much memory each process needs.
- Change scheduling priorities to run processes in groups that fit comfortably in memory: must shed load.

Working sets

Informally, a working set of a process is the collection of pages that a process is using actively, and which must thus be memory resident for the process to make progress (and avoid thrashing).

If the sum of all working sets of all runnable threads exceeds the size of memory, then stop running some of the threads for a while.

Divide processes into two groups: active and inactive:

- When a process is active, its entire working set must always be in memory: never execute a thread whose working set is not resident.
- When a process becomes inactive, its working set can migrate to disk
- Threads from inactive processes are never scheduled for execution.
- The collection of active processes is called the *balance set*.
- Gradually move processes in and out of the balance set.
- As working sets change, the balance set must be adjusted.

How to compute ?

Denning proposed a parameter T ; all pages referenced in the last T seconds comprise the working set. Note that recency of access (and not frequency of access) is used to determine the page's value (as in LRU).

Computation of working set can be done by maintaining an *idle time* with each page.

Clock algorithm extended to also increment the idle time, each time the hand passes through the page and the page is found "not-accessed". If the page is found accessed, the idle time is reset to zero.

Pages with idle times less than T are in the working set.


Current systems take more aggressive approach to memory overload.

For example, some versions of Linux run an **out-of-memory killer** when memory is oversubscribed;

this daemon chooses a memory-intensive process and kills it, thus reducing memory.

While successful at reducing memory pressure, this approach can have problems,

if, for example, it kills the X server and thus renders any applications requiring the display unusable.

A solid blue horizontal bar spanning the width of the slide, located at the bottom.

Conclusion

Illusion of virtual memory:

Processes can run when sum of virtual address spaces $>$ amount of physical memory

Mechanism:

Extend page table entry with “present” bit

OS handles page faults (or page misses) by reading in desired page from disk

Policy:

Page selection – demand paging, prefetching, hints

Page replacement – OPT, FIFO, LRU, others

Implementations (clock) perform approximation of LRU

Free Space Management

malloc library (managing pages of a process's heap) or
the OS itself (managing portions of the address space of a process)

How should free space be managed, when satisfying variable-sized requests?

What strategies can be used to minimize fragmentation?

What are the time and space overheads of alternate approaches?



Assumptions


`void*malloc(size_t size)` takes a single parameter, **`size`**, which is the number of bytes requested by the application; it hands back a pointer (of no particular type, or a void pointer) to a region of that size (or greater)

`void free(void*ptr)` takes a pointer and frees the corresponding chunk.

Note the implication of the interface:

the user, when freeing the space, does not inform the library of its size;


the library must be able to figure out how big a chunk of memory is when handed just a pointer to it.



The space that this library manages is known historically as the **heap**, the generic data structure used to manage free space in the heap is some kind of free list.

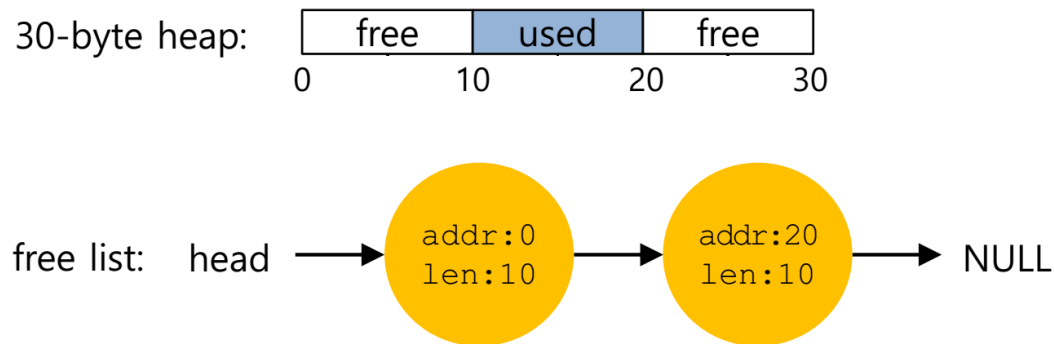
once memory is handed out to a client, it cannot be relocated to another location in memory.

For example, if a program calls `malloc()` and is given a pointer to some space within the heap, that memory region is essentially “owned” by the program (and cannot be moved by the library) until the program returns it via a corresponding call to `free()`.



Variable Sized allocation

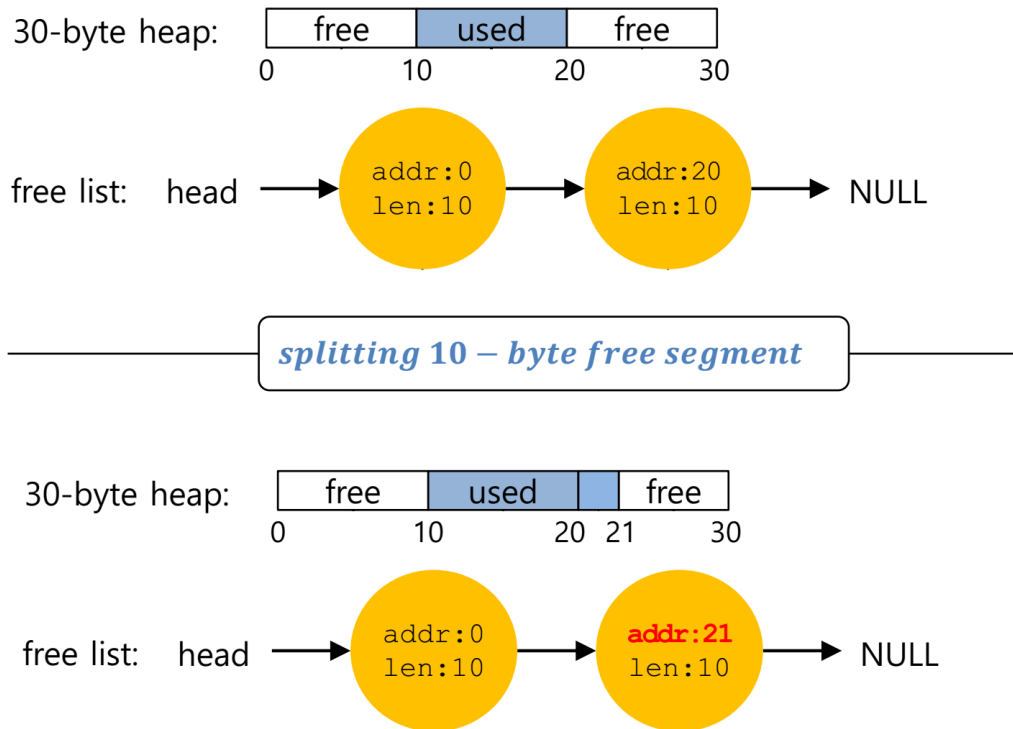
- Given a block of memory, how do we allocate it to satisfy various memory allocation requests?
- This problem must be solved in the C library
 - Allocates one or more pages from kernel via brk/sbrk or mmap system calls
 - Gives out smaller chunks to user programs via malloc
- This problem also occurs in the kernel
 - Kernel must allocate memory for its internal data structures



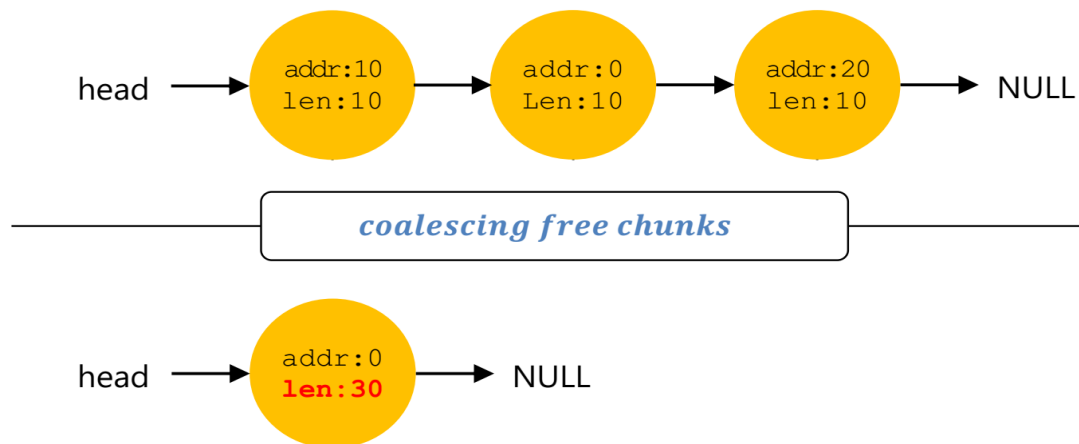
In this case, the total free space available is 20 bytes; unfortunately, it is fragmented into two chunks of size 10 each. As a result, a request for 15 bytes will fail even though there are 20 bytes free

Splitting and Coalescing

- Two 10-bytes free segment with **1-byte request**



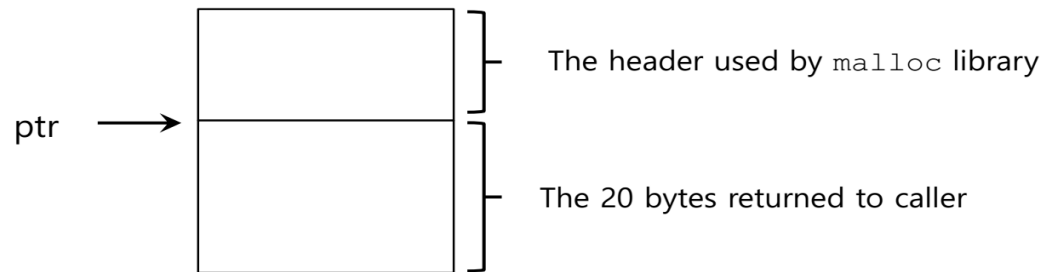
-
- ❑ If a user requests memory that is **bigger than free chunk size**, the list will **not find** such a free chunk.
 - ❑ Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**.



Tracking the size of allocated regions

- ▣ The interface to `free(void *ptr)` does **not take a size parameter**.
 - ◆ How does the library **know the size** of memory region that will be back **into free list**?
- ▣ Most allocators store **extra information** in a **header** block.

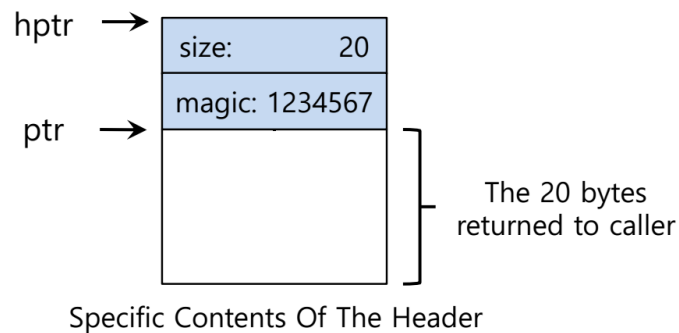
```
ptr = malloc(20);
```



An Allocated Region Plus Header

Header details

- ▣ The header minimally **contains the size** of the allocated memory region.
- ▣ The header may also contain
 - ◆ Additional pointers to speed up deallocation
 - ◆ A magic number for integrity checking



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

A Simple Header

-
- The **size** for free region is the **size of the header plus the size of the space** allocated to the user.
 - ◆ If a user **request N bytes**, the library searches for a free chunk of **size N plus the size of the header**
 - Simple pointer arithmetic to find the header pointer.

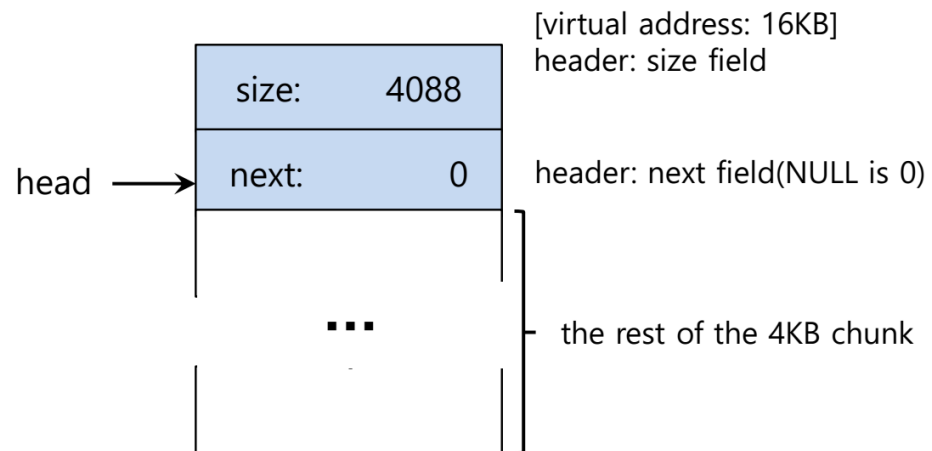
```
void free(void *ptr) {  
    header_t *hptr = (void *)ptr - sizeof(header_t);  
}
```


Free list


Free space is managed as a list

- Pointer to the next free chunk is embedded within the free chunk
- The library tracks the head of the list
- Allocations happen from the head

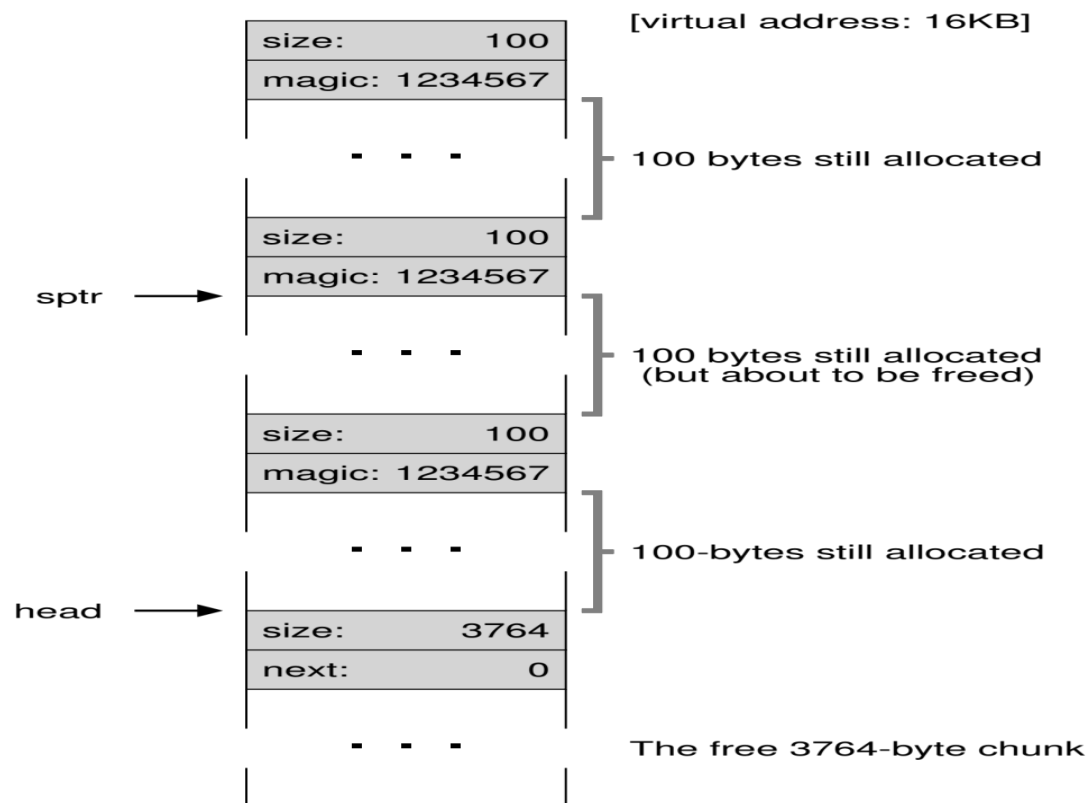
```
// mmap() returns a pointer to a chunk of free space
node_t *head = mmap(NULL, 4096, PROT_READ|PROT_WRITE,
                    MAP_ANON|MAP_PRIVATE, -1, 0);
head->size = 4096 - sizeof(node_t);
head->next = NULL;
```



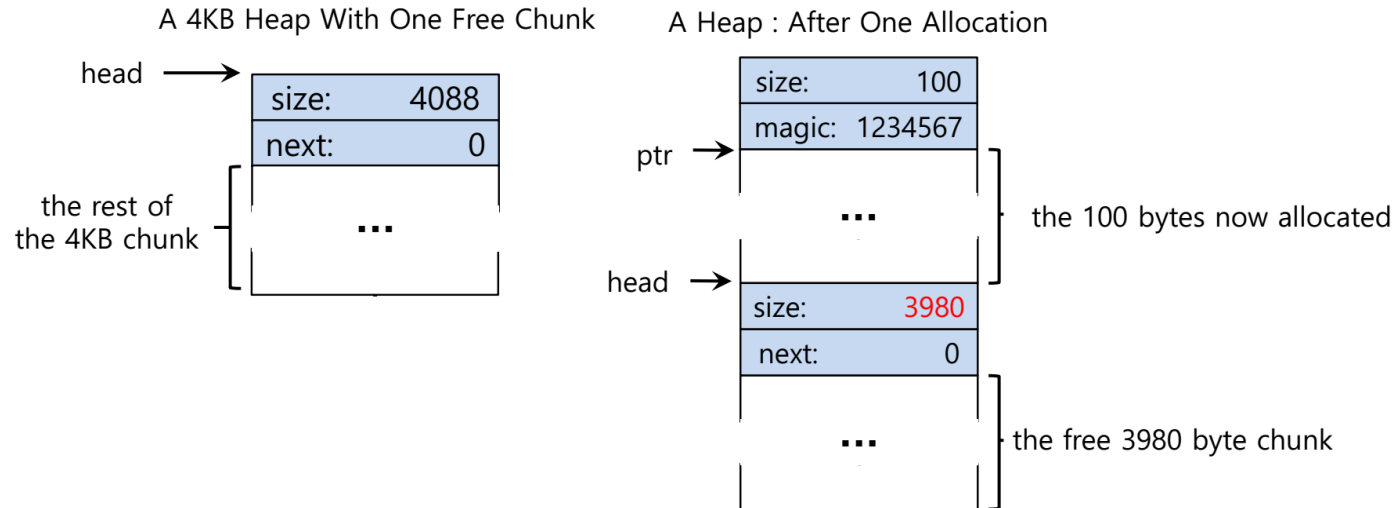
-
- ▣ If a chunk of memory is requested, the library **will first find** a chunk that is **large enough** to accommodate the request.

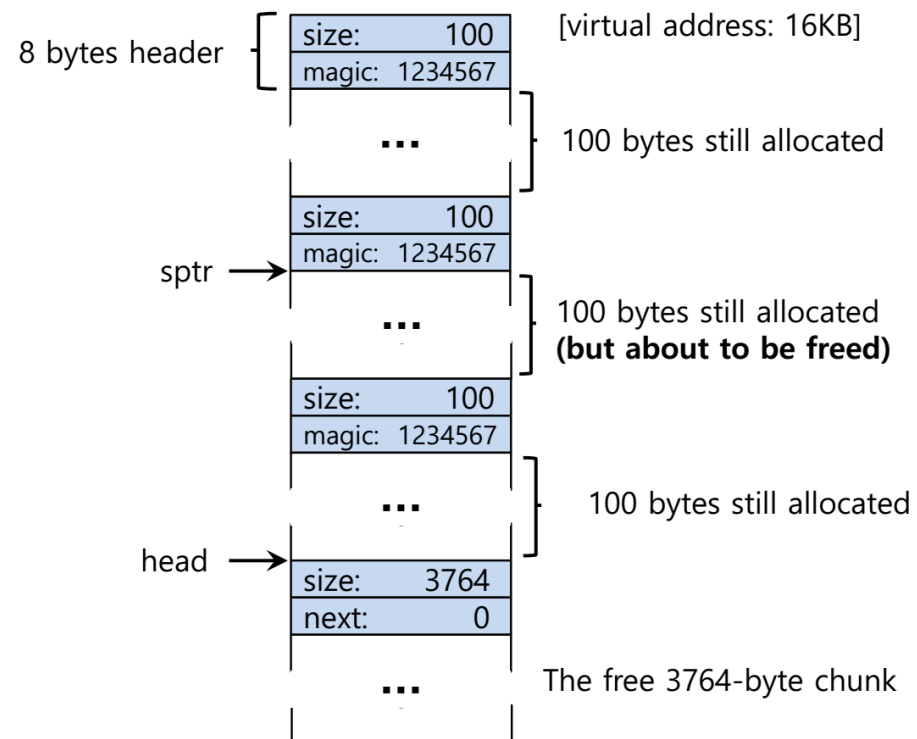
 - ▣ The library will
 - ◆ **Split** the large free chunk into two.
 - **One** for the **request** and the **remaining** free chunk
 - ◆ **Shrink** the size of free chunk in the list.
- 

Free space - Three chunks allocated



- ▣ Example: a request for 100 bytes by `ptr = malloc(100)`
 - ◆ Allocating 108 bytes out of the existing one free chunk.
 - ◆ shrinking the one free chunk to 3980(4088 minus 108).





Free Space With Three Chunks Allocated

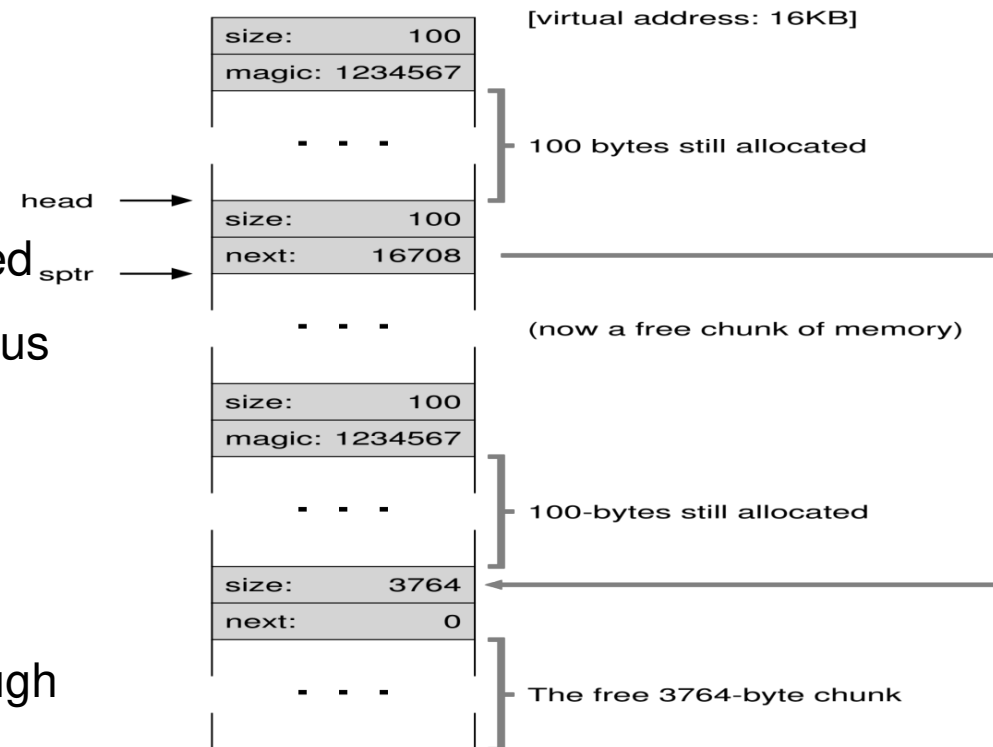
- Suppose 3 allocations of size 100 bytes each happen.

Then, the middle chunk pointed to by `sptr` is freed

- What is the free list?—It now has two non-contiguous elements

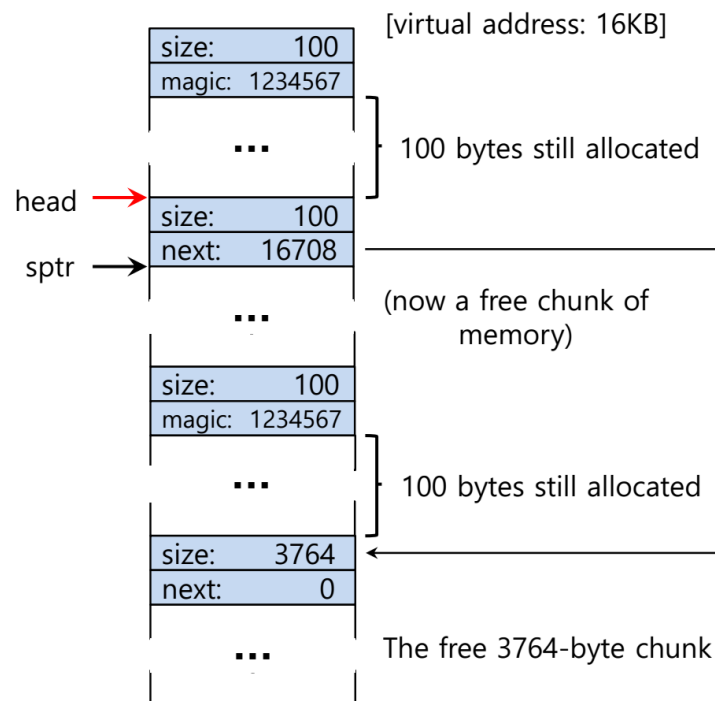
- Free space may be scattered around due to fragmentation

- Cannot satisfy a request for 3800 bytes even though we have the freespace



□ Example: `free(sptr)`

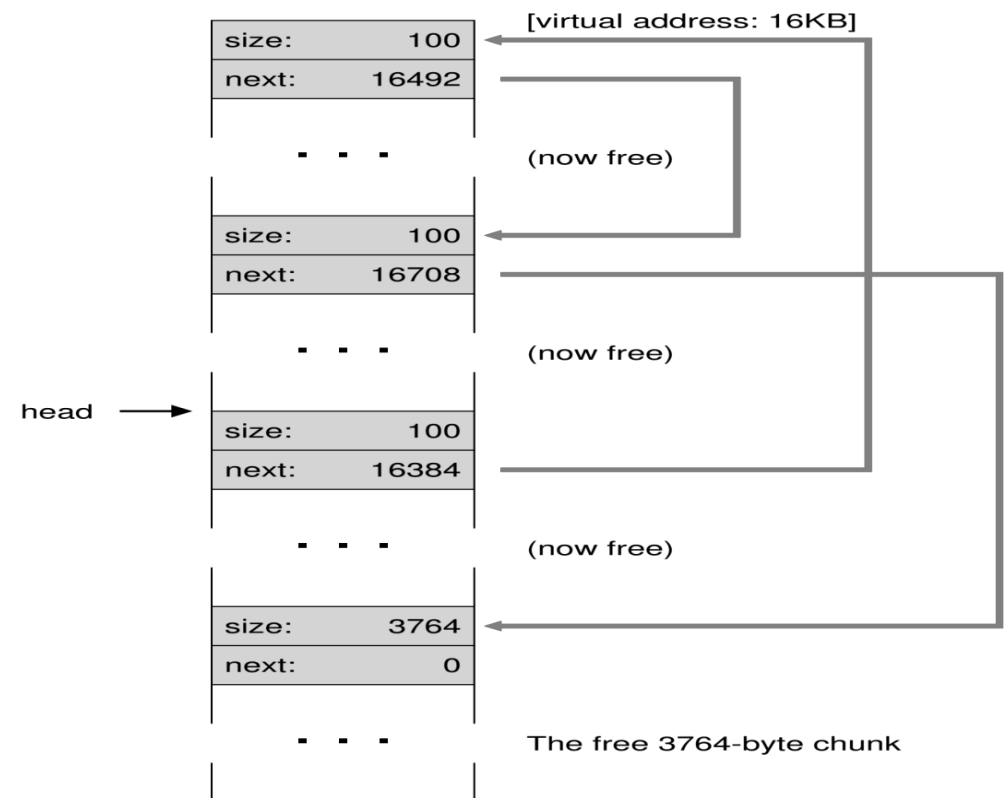
- ♦ The 100 bytes chunks is **back into** the free list.
- ♦ The free list will **start** with a **small chunk**.
 - The list header will point the small chunk



Coalescing

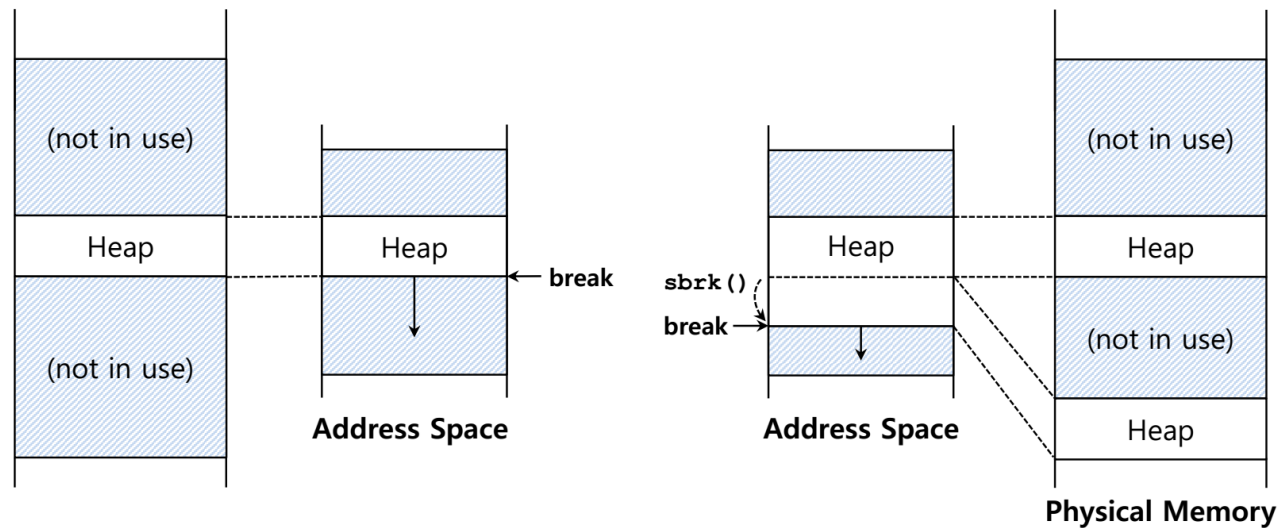
Suppose all the three chunks are freed

- 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



Growing the heap

- Most allocators **start** with a **small-sized heap** and then **request more** memory from the OS when they run out.
 - e.g., `sbrk()`, `brk()` in most UNIX systems.



Managing Free space – Basic strategies

- Best Fit:

- ◆ Finding free chunks that are **big or bigger than the request**
- ◆ Returning the **one of smallest** in the chunks **in the group** of candidates

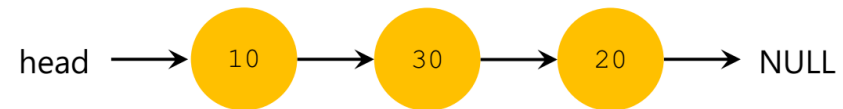
- Worst Fit:

- ◆ Finding the **largest free chunks** and allocation the amount of the request
- ◆ **Keeping the remaining chunk** on the free list.

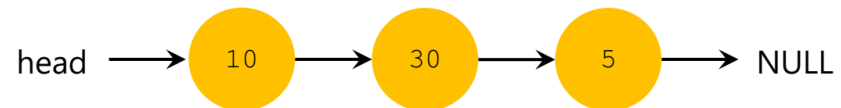
First Fit : Simply finds the first block that is big enough and returns the requested amount to the user

Next Fit : Instead of always beginning the first-fit search at the beginning of the list, the next fit algorithm keeps an extra pointer to the location within the list where one was looking last.

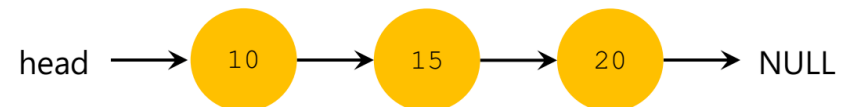
-
- Allocation Request Size 15



- Result of Best-fit



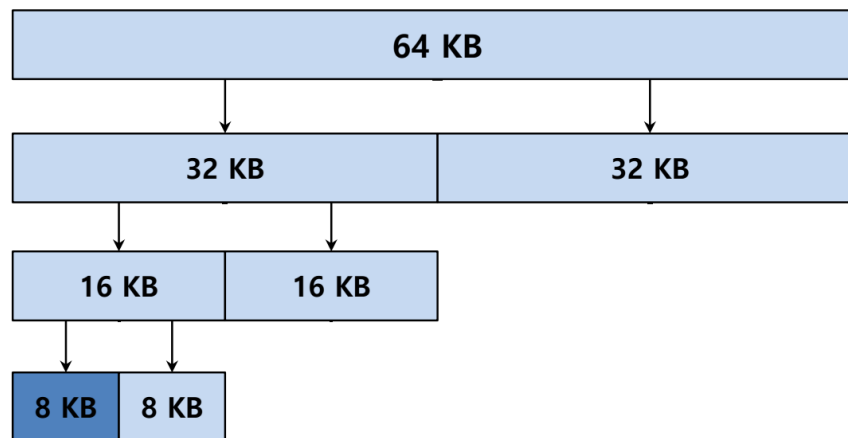
- Result of Worst-fit



Binary Buddy allocator

□ Binary Buddy Allocation

- ♦ The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**.



64KB free space for 7KB request

Allocate memory in size of power of 2 –E.g., for a request of 7000 bytes, allocate 8KB chunk

- 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

Fixed size allocations

Memory allocation algorithms are much simpler with fixed size allocations

- Page-sized fixed allocations in kernel:–Has free list of pages–Pointer to next page stored in the free page itself
- For some smaller allocations (e.g., PCB), kernel uses a slab allocator
 - Object caches for each type (size) of objects
 - Within each cache, only fixed size allocation
 - Each cache is made up of one or more “slabs”
- Fixed size memory allocators can be used in user programs also (instead of generic malloc)