3.3 Dynamic Storage Allocation

- *Best-fit & First-fit Algorithms*
- Buddy System
- Slab Allocation

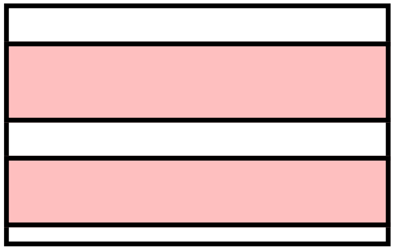
Dynamic Storage Allocation

- Where in the kernel do you need to do memory allocation?
  - stack space
  - `malloc()`
  - `fork()`
  - various OS data structures
    - process control block
    - thread control block
    - mutex (it's a queue)
  - etc.

Dynamic Storage Allocation

- Goal: allow dynamic creation and destruction of data structures
- Concerns:
  - efficient use of storage
  - efficient use of processor time
- Example:
  - *first-fit* vs. *best-fit* allocation

Allocation Example



1300 bytes free (first)

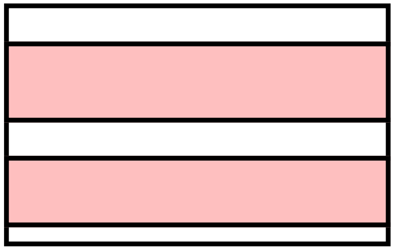
1200 bytes free (last)

Allocate 1000 bytes: **First Fit** **Best Fit**

Allocate 1100 bytes:

Allocate 250 bytes:

Allocation Example

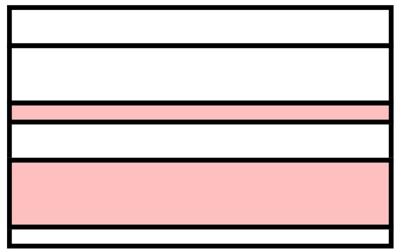


1300 bytes free (first)

1200 bytes free (last)

Allocate 1000 bytes:

First Fit



300

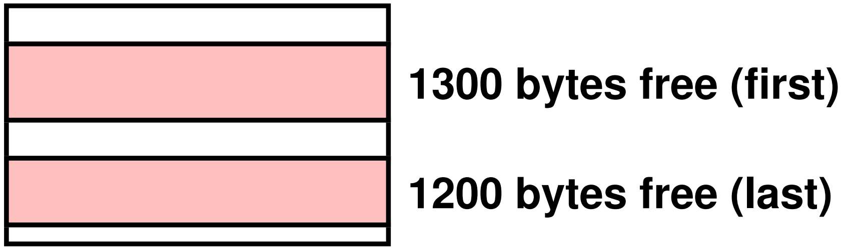
1200

Best Fit

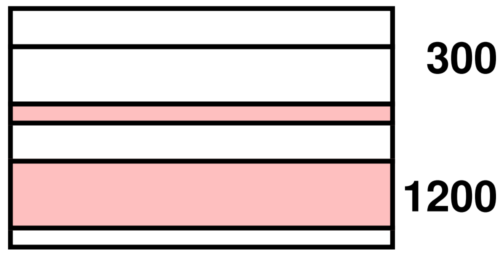
Allocate 1100 bytes:

Allocate 250 bytes:

Allocation Example



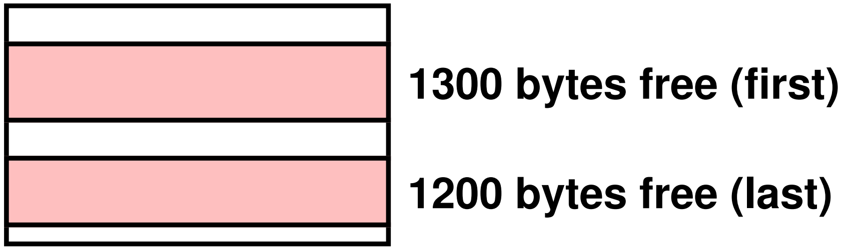
Allocate 1000 bytes: | **First Fit** | **Best Fit**



Allocate 1100 bytes:

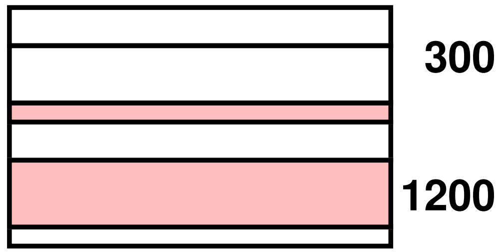
Allocate 250 bytes:

Allocation Example

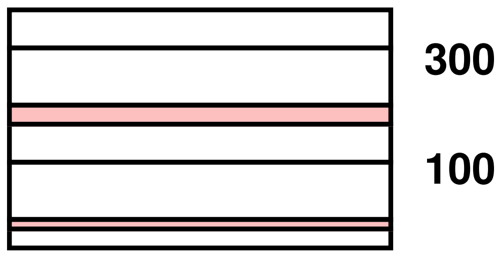


Allocate 1000 bytes:

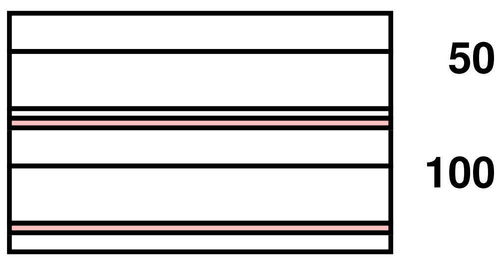
First Fit | **Best Fit**



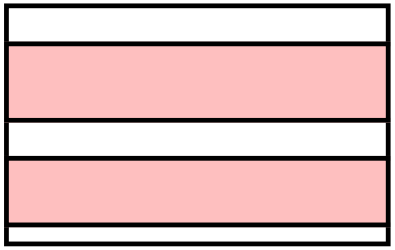
Allocate 1100 bytes:



Allocate 250 bytes:

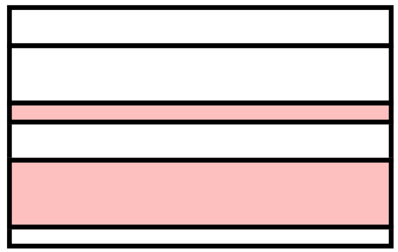


Allocation Example

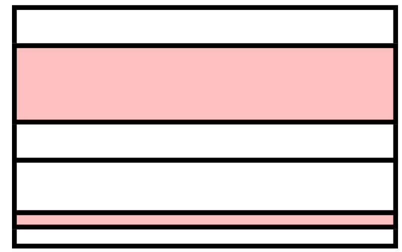


Allocate 1000 bytes:

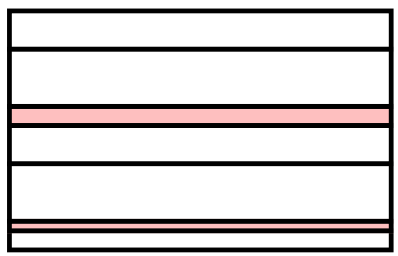
First Fit



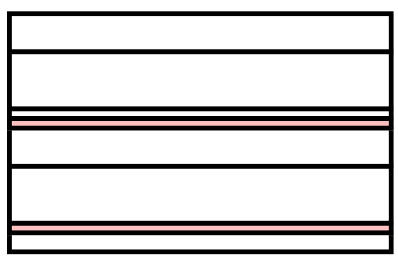
Best Fit



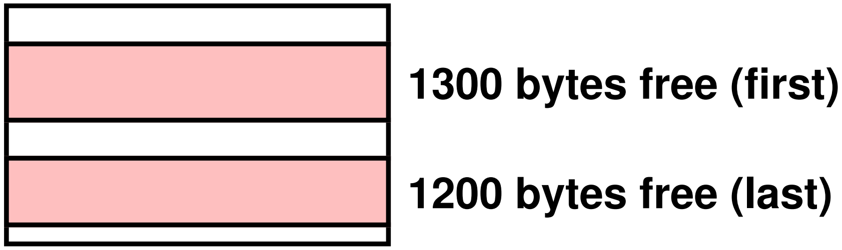
Allocate 1100 bytes:



Allocate 250 bytes:

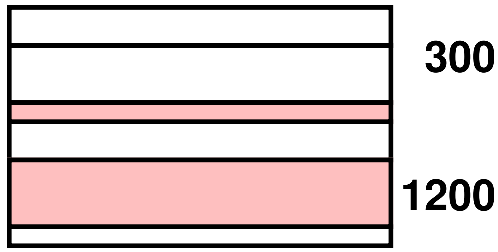


Allocation Example

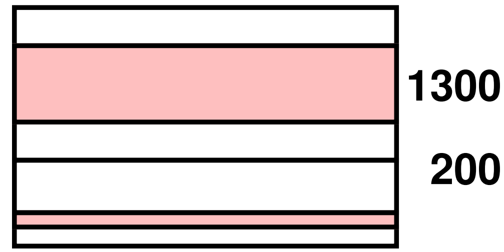


Allocate 1000 bytes:

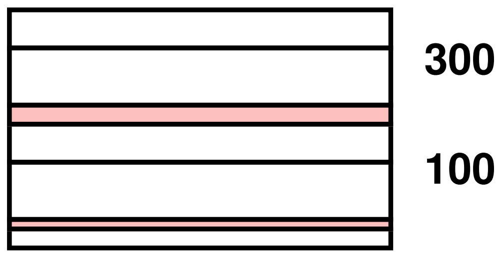
First Fit



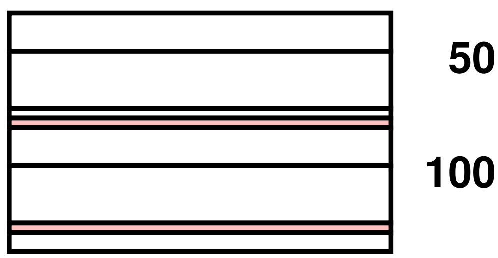
Best Fit



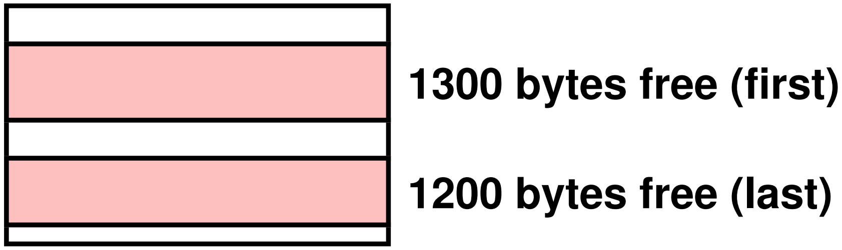
Allocate 1100 bytes:



Allocate 250 bytes:



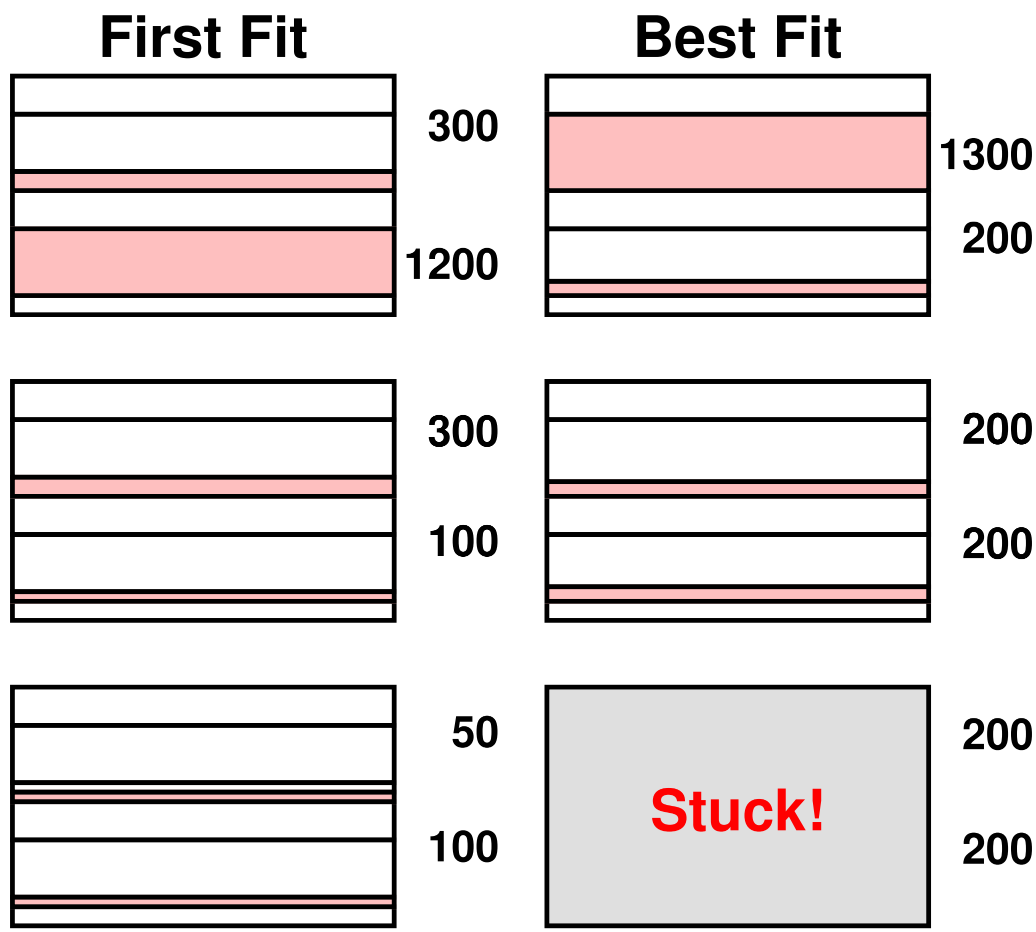
Allocation Example



Allocate 1000 bytes:

Allocate 1100 bytes:

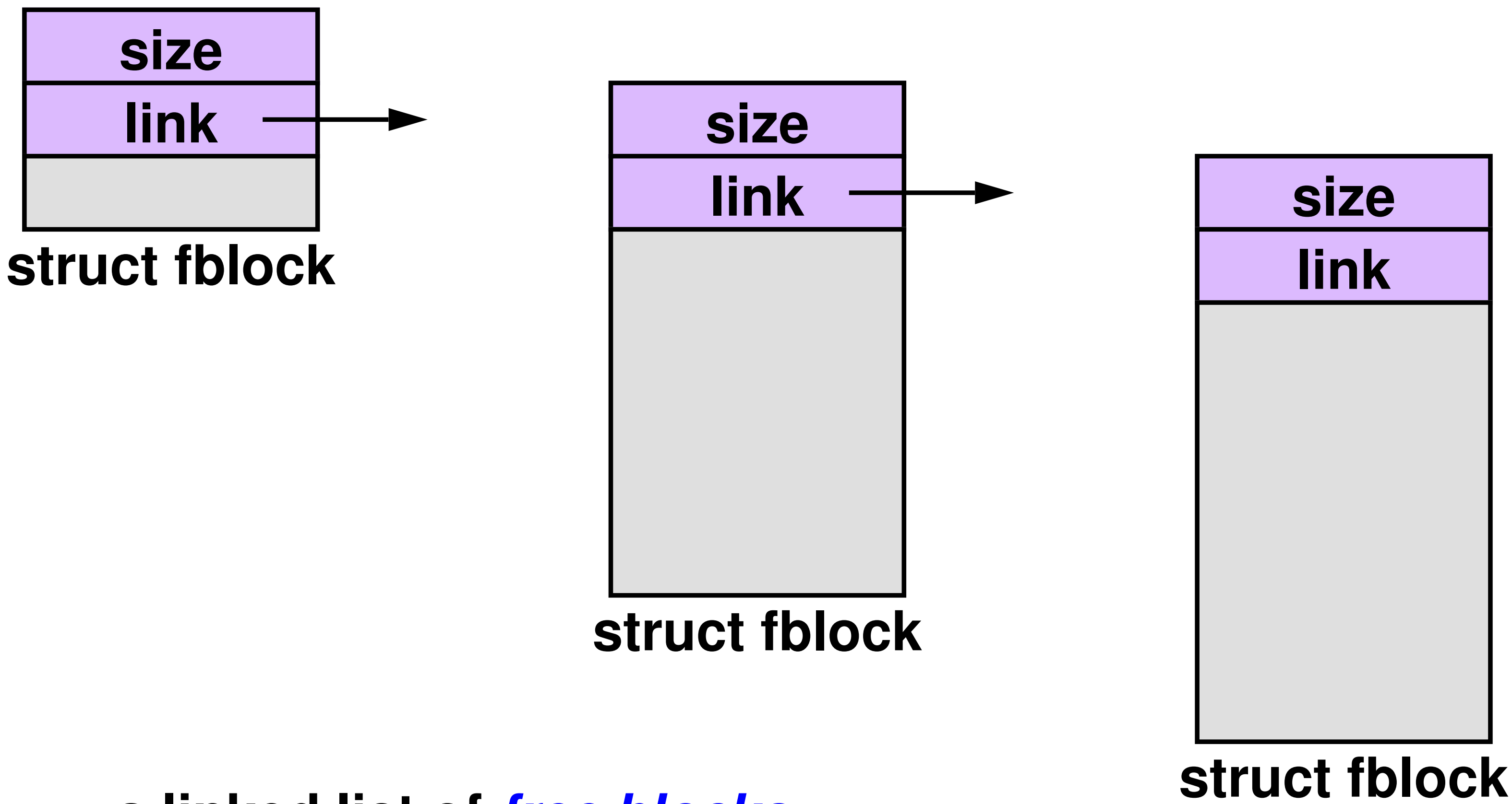
Allocate 250 bytes:



Fragmentation

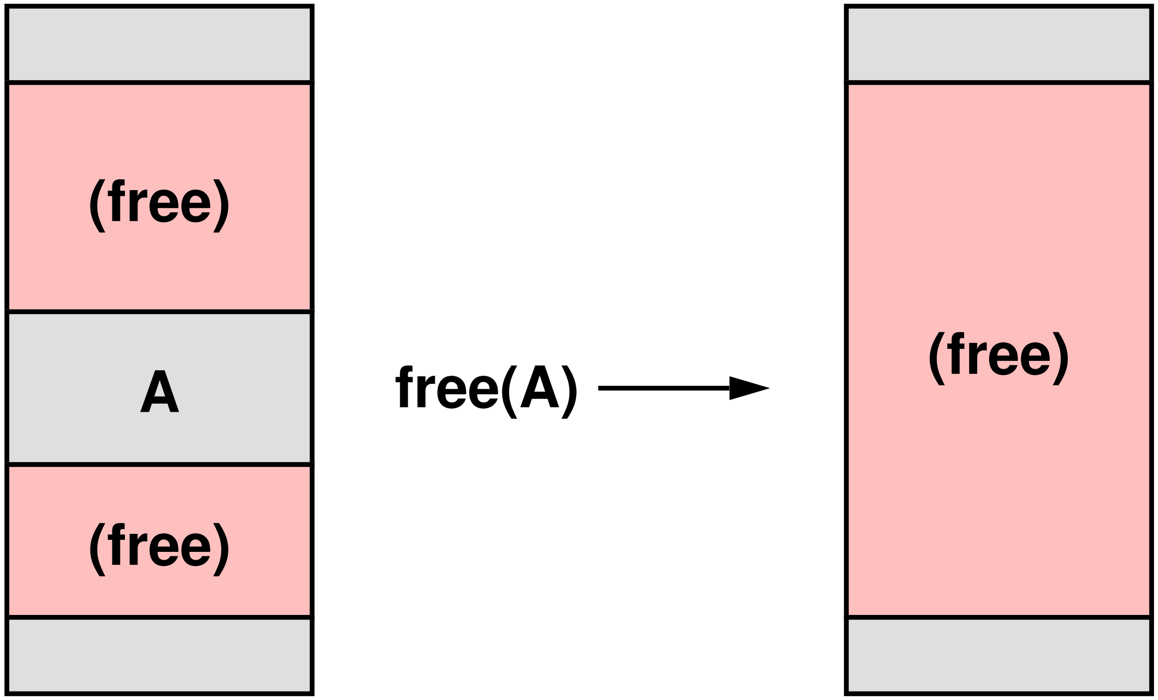
- ***First-fit*** **vs.** ***best-fit*** **allocation**
  - studies have shown that first-fit works better
  - best-fit tends to leave behind a large number of regions of memory that are too small to be useful
    - best-fit tends to create smallest left-over blocks!
  - this is the general problem of *fragmentation*
    - *internal fragmentation:* unusable memory is contained within an allocated region (e.g., buddy system)
    - *external fragmentation:* unusable memory is separated into small blocks and is interspersed by allocated memory (e.g., best-fit)

Implementing First Fit: Data Structures



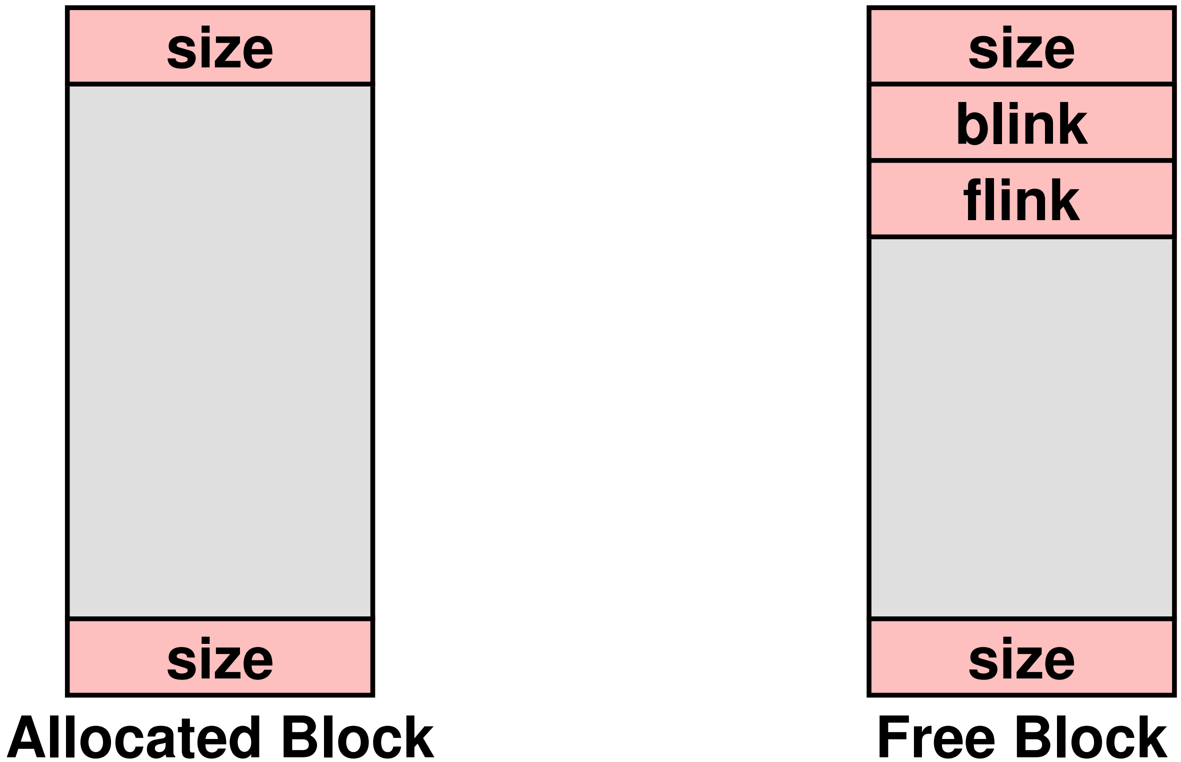
- a linked list of *free blocks*
  - don't need to manage allocated blocks
- use a doubly-linked list
  - insertion and deletion are fast, i.e., *O(1)*, once you know where to insert or delete

Liberation of Storage



- This is known as *coalescing*
  - in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
    - you also need to know if they are *allocated* or *free*

Boundary Tags



- This is known as *coalescing*
  - in order to make coalescing possible, you need to know that *size* of the blocks above and below the block being freed
    - you also need to know if they are *allocated* or *free*

Detailed Examples

Free block

start	in-use = 0	size
start+8	prev = 0	next = 0
start+16	*(Garbage, don't care)*	
start+size-8	in-use = 0	size

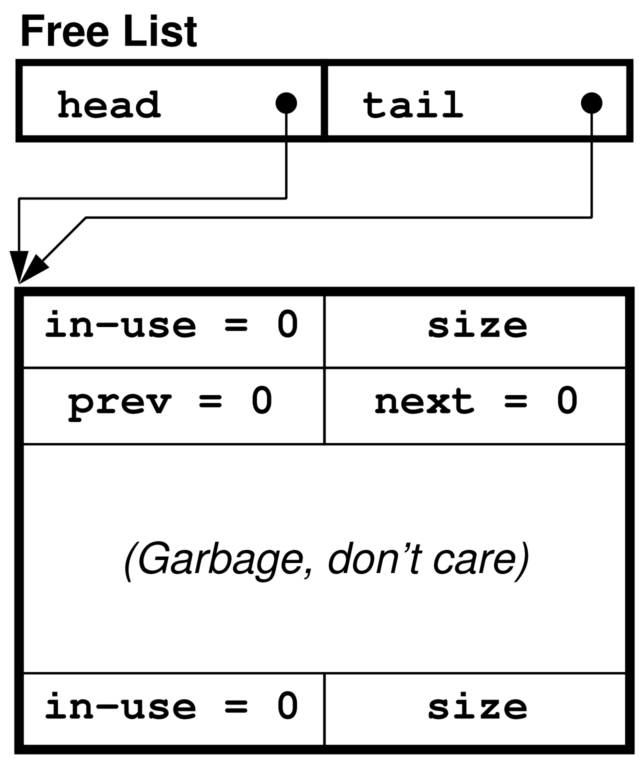
size

In-use block

start	in-use = 1	size
start+8	*Return to User! Hands off!*	
start+size-8	in-use = 1	size

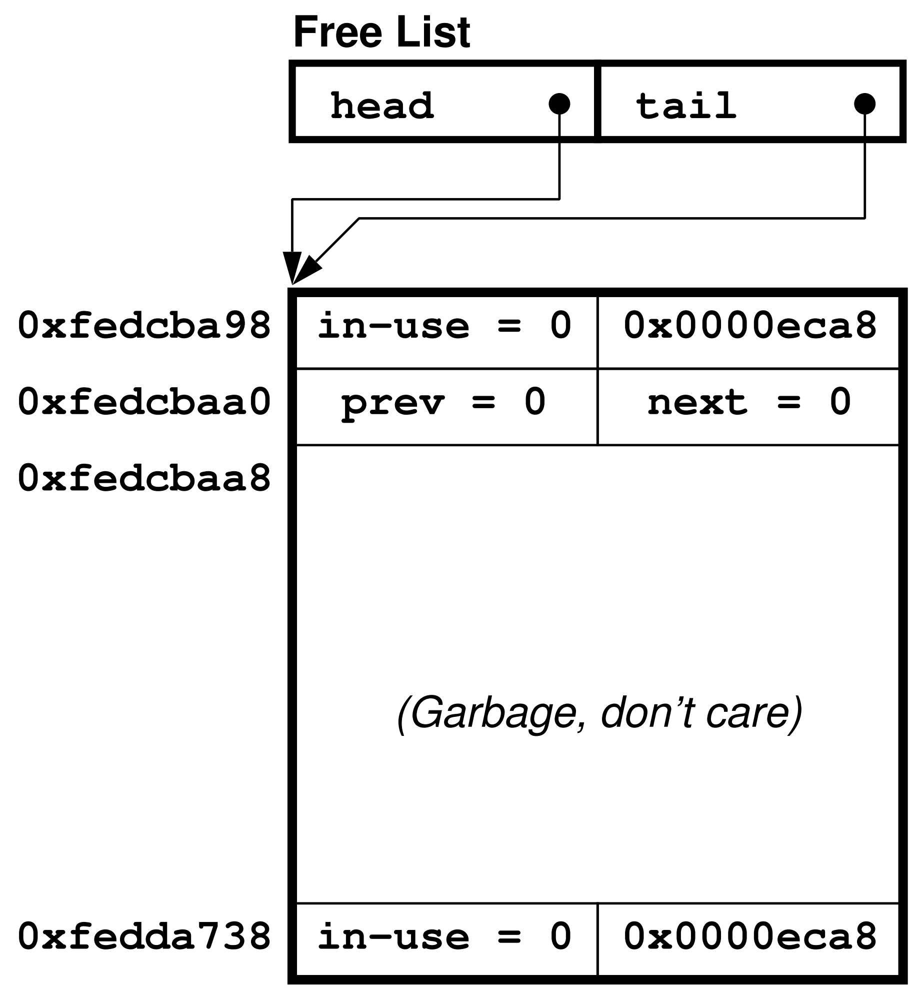
size

Free list



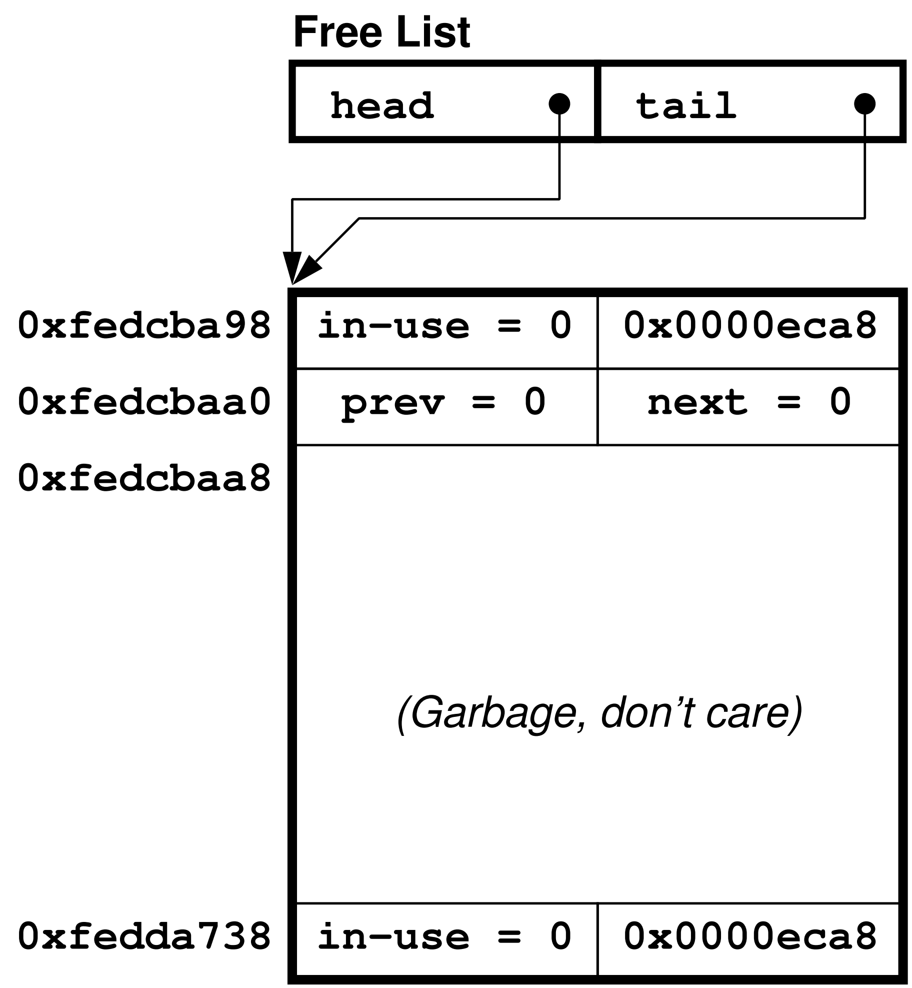
`malloc()` Example

- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



`malloc()` Example

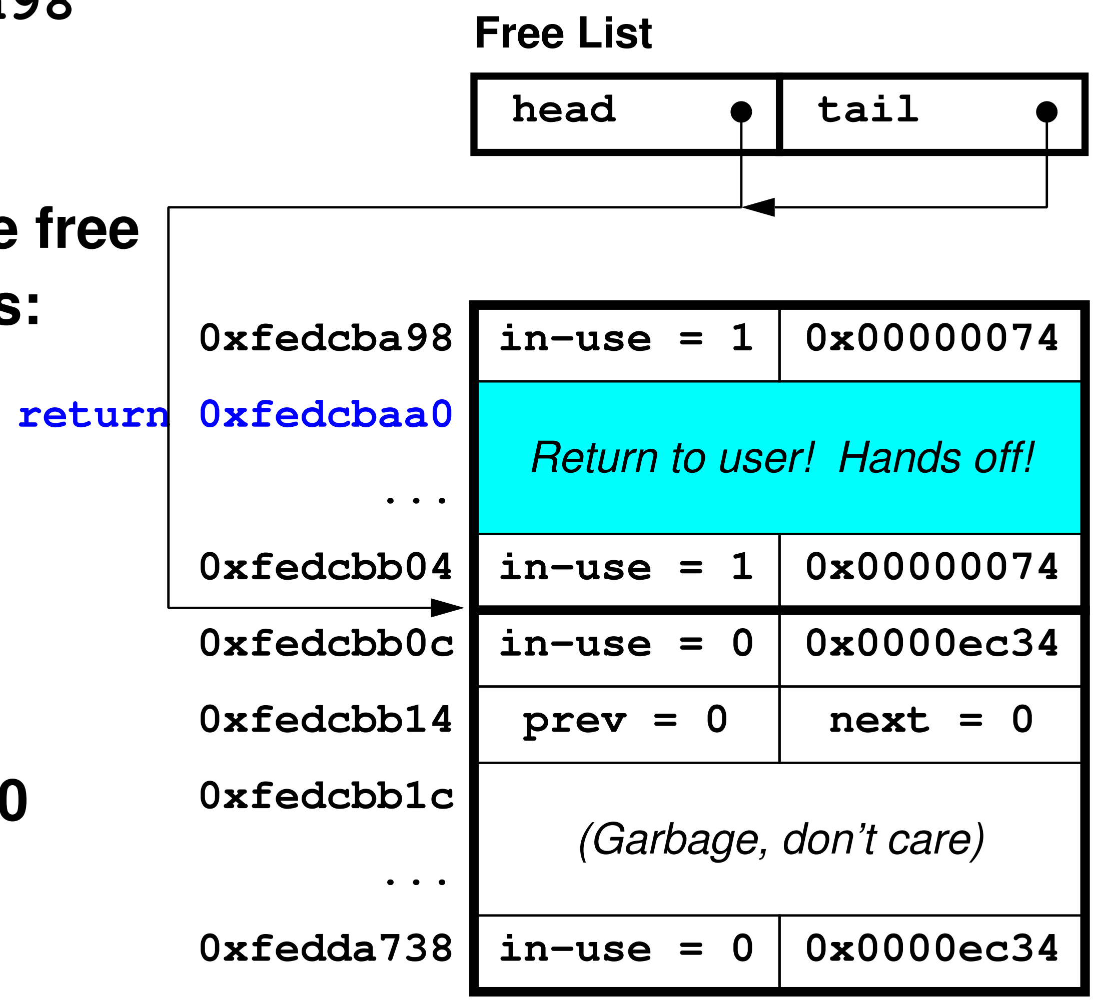
- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



- Ex: Request block size is 100
  - split the block into two
  - busy block size is 116
  - remaining free block size is 60584-116 =60468=0xec34

`malloc()` Example

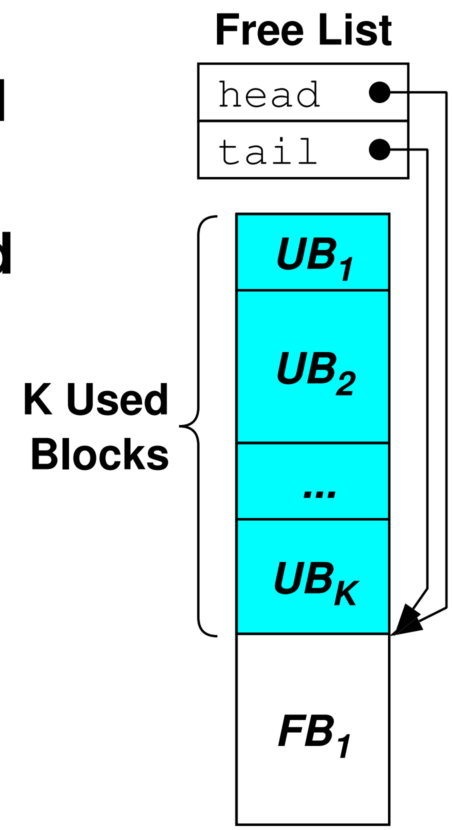
- Ex: Heap starts at `0xfedcba98` and size of the heap is `0x0000eca8` (60,584) bytes
  - the Free List contains one free block and it looks like this:



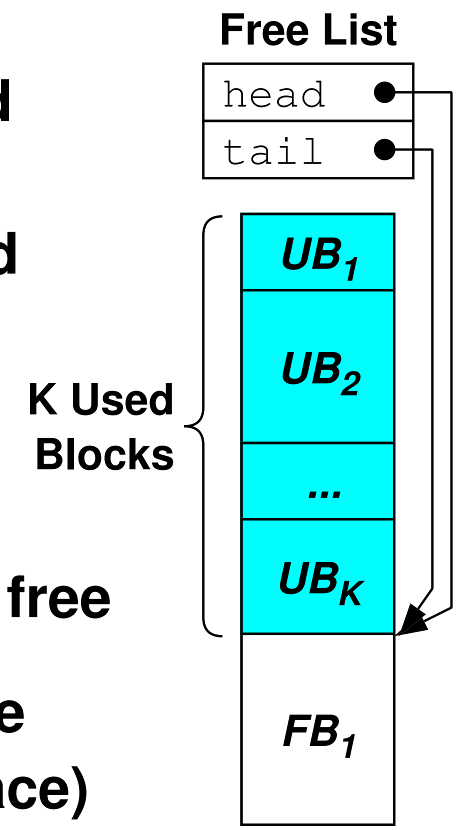
- Ex: Request block size is 100
  - split the block into two
  - busy block size is 116
  - remaining free block size is 60584-116 =60468=0xec34

`free()` Example

- After *K* blocks of memory have been allocated (and assume that none of them have been deallocated)
  - in the memory layout, the first *K* blocks are used block, followed by one free block



`free()` Example



- After *K* blocks of memory have been allocated (and assume that none of them have been deallocated)
  - in the memory layout, the first *K* blocks are used block, followed by one free block
- Memory blocks can be freed in any order
  - when a memory block is freed, we need to check if the blocks before it and after it are also free
- If neither of them are free, we just need to insert the newly freed block into the Free List (at the right place)
  - need to *search* the Free List to find insertion point
  - searching through a linear list is "slow", *O(n)*
- Otherwise, we can *merge/coalesce* the block in question with neighboring free block(s)

`free()` Example

- Ex: `free(Y)`
  - `Y-16` tells you if the *previous* block is free or not
  - `Y-8+Z` tells you if the *next* block is free or not
    - where `Z` is what's in `Y-4`

`Y-8-(*(Y-12))`	`in-use=?`	`size`
	?	
`Y-16`	`in-use=?`	`size`
`Y-8`	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
	`in-use=1`	`size=Z`
`Y-8+Z`	`in-use=?`	`size`
	?	
	`in-use=?`	`size`

- *Coalescing:*
  - need to make sure that everything is consistent

`free()` Example

- Ex: `free(Y)` and ***previous block is free*** and ***next block is busy***
  - i.e., `Y-16` is `0` and `Y-8+Z` is `1`
    - where `Z` is what's in `Y-4` and `W` is what's in `Y-12`
  - furthermore, `Y-8-W` is on the Free List
  - coalesce this block and the ***previous*** block

`Y-8-W`	`in-use=0`	`size=W`
	`prev`	`next`
	(Garbage, don't care)	
`Y-16`	`in-use=0`	`size=W`
`Y-8`	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
`Y-16+Z`	`in-use=1`	`size=Z`
`Y-8+Z`	`in-use=1`	`size`
	Hands off!	
	`in-use=1`	`size`

`free()` Example

- Ex: `free(Y)` and *previous block is free* and *next block is busy*
  - i.e., `Y-16` is `0` and `Y-8+Z` is `1`
    - where `Z` is what's in `Y-4` and `W` is what's in `Y-12`
  - furthermore, `Y-8-W` is on the Free List
  - coalesce this block and the *previous* block
    - easy!
    - just change `Y-12+Z` and `Y-4-W` to `W+Z` and `Y-16+Z` to `0`
    - don't even need to change prev and next!

`Y-8-W`	`in-use=0`	`size=W+Z`
	`prev`	`next`
`Y`	*(Garbage, don't care)*	
`Y-16+Z`	`in-use=0`	`size=W+Z`
`Y-8+Z`	`in-use=1`	`size`
	Hands off!	
	`in-use=1`	`size`

`free()` Example

- Ex: `free(Y)` and ***previous block is busy*** and ***next block is free***
  - i.e., `Y-16` is `1` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4` and `X` is what's in `Y-4+Z`
  - furthermore, `Y-8+Z` is on the Free List
  - coalesce this block and the ***next*** block

`Y-8-W`	`in-use=1`	`size=W`
	Hands off!	
`Y-16`	`in-use=1`	`size=W`
`Y-8`	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
	`in-use=1`	`size=Z`
`Y-8+Z`	`in-use=0`	`size=X`
	`prev`	`next`
	(Garbage, don't care)	
`Y-16+Z+X`	`in-use=0`	`size=X`

`free()` Example

- Ex: `free(Y)` and *previous block is busy* and *next block is free*
  - i.e., `Y-16` is `1` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4` and `X` is what's in `Y-4+Z`
  - furthermore, `Y-8+Z` is on the Free List
  - coalesce this block and the *next* block
    - just change `Y-4` and `Y-12+Z+X` to `Z+X` and `Y-8` to `0`
    - move prev and next pointers
    - adjust `next` field in previoius block in Free List
    - adjust `prev` field in next block in Free List
    - may need to update where Free List points

`Y-8-W`	`in-use=1`	`size=W`
	Hands off!	
`Y-16`	`in-use=1`	`size=W`
`Y-8`	`in-use=0`	`size=Z+X`
`Y`	`prev`	`next`
	(Garbage, don't care)	
`Y-16+Z+X`	`in-use=0`	`size=Z+X`

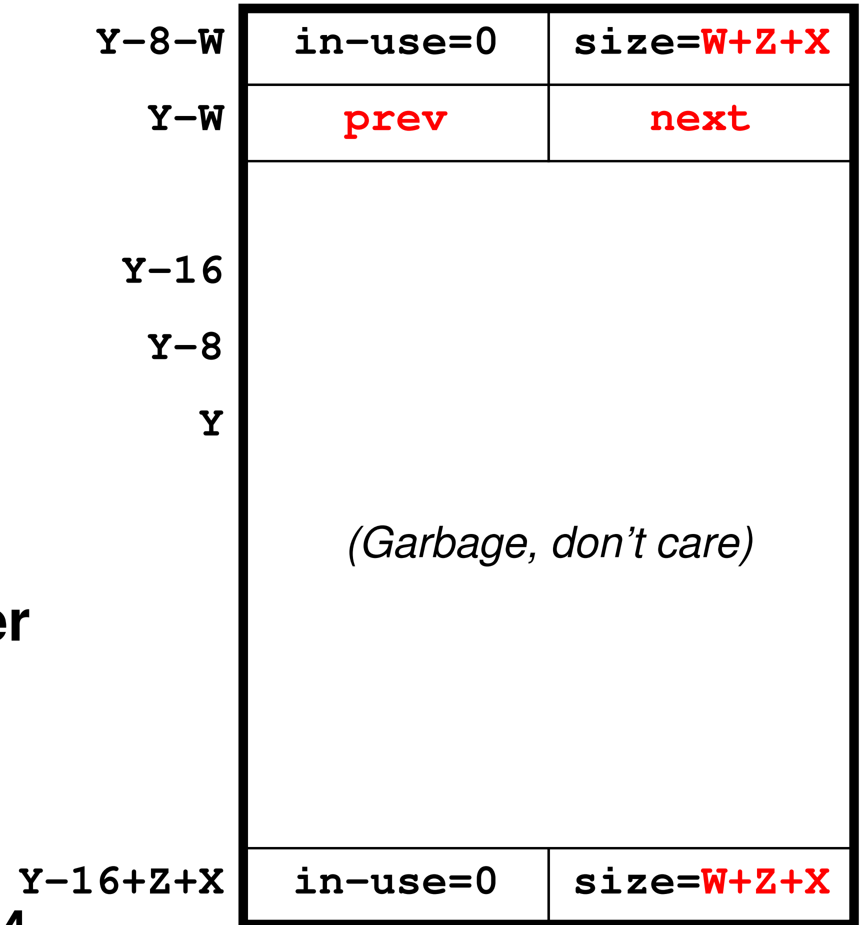
`free()` Example

- Ex: `free(Y)` and *previous block is free* and *next block is also free*
  - i.e., `Y-16` is `0` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4`, `X` is what's in `Y-4+Z`, and `W` is what's in `Y-12`
  - blocks starting at `Y-8-W` and `Y-8+Z` are both on the Free List and next to and point at each other
  - coalesce all 3 blocks

`Y-8-W`	`in-use=0`	`size=W`
`Y-W`	`prev`	`Y-8+Z`
	(Garbage, don't care)	
`Y-16`	`in-use=0`	`size=W`
`Y-8`	`in-use=1`	`size=Z`
`Y`	*Return to user! Hands off!*	
	`in-use=1`	`size=Z`
`Y-8+Z`	`in-use=0`	`size=X`
`Y+Z`	`Y-8-W`	`next`
	(Garbage, don't care)	
	`in-use=0`	`size=X`

`free()` Example

- Ex: `free(Y)` and ***previous block is free*** and ***next block is also free***
  - i.e., `Y-16` is `0` and `Y-8+Z` is `0`
    - where `Z` is what's in `Y-4`, `X` is what's in `Y-4+Z`, and `W` is what's in `Y-12`
  - blocks starting at `Y-8-W` and `Y-8+Z` are both on the Free List and next to and point at each other
  - coalesce all 3 blocks
    - just change `Y-4-W` and `Y-12+Z+X` to `W+Z+X`
    - copy `next` from `Y+Z+4` to `Y-W+4`
    - adjust `prev` field in the new `next` block in Free List to point to `Y-8-W`
    - may need to update where Free List points



First-fit & Best-fit Algorithms

- **Memory allocator must run fast**
  - it does not check if the free list is in a consistent state
    - just like our warmup 1 assignment

- **One bad bit in the memory allocator data structure and it can break the memory allocator code**
  - if you write into a *boundary tag*, your program may die in `malloc()` or `free()`
  - what would happen if you call `free()` twice on the same address?
  - user/application code can *corrupt the memory allocation chain* easily
    - the result can lead to *segmentation faults*
    - unfortunately, the corruption can *stay hidden* for a long time and *eventually* lead to a segmentation fault
      - memory corruption bugs are very difficult to squash

First-fit Algorithm

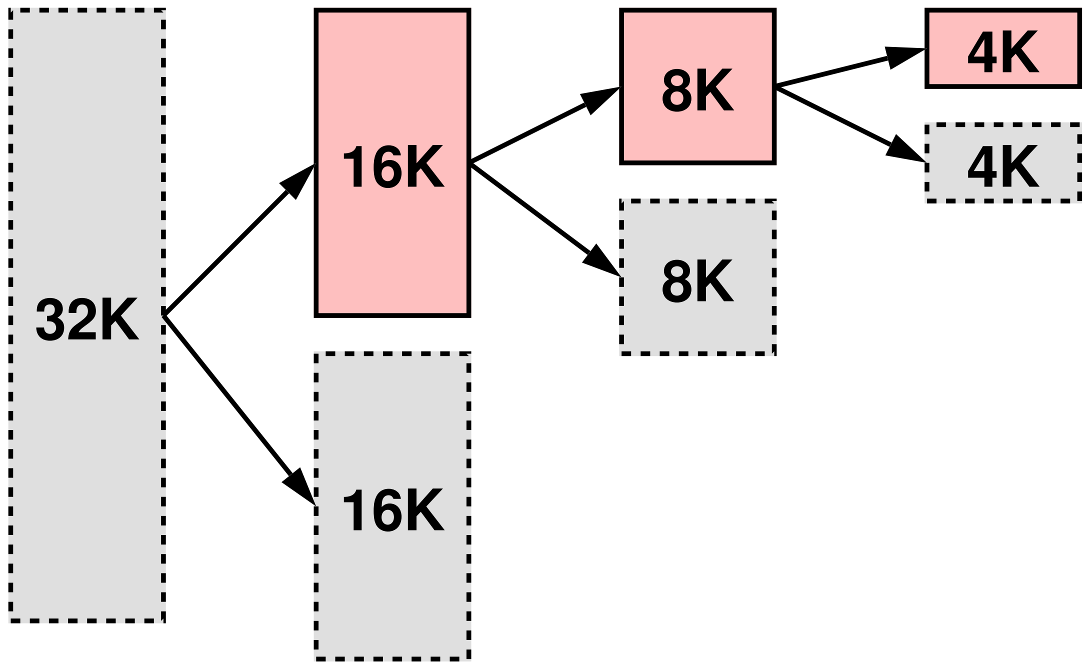
- Let *n* be the number of free blocks on the free list
  - `malloc()` is *O(n)*
  - `free(ptr)` is *O(n)*
    - occurs when the blocks around the block containing `ptr` are both in-use
- Such performance in unacceptable in the kernel

3.3 Dynamic Storage Allocation

- Best-fit & First-fit Algorithms
- *Buddy System*
- Slab Allocation

Buddy Lists

Ex: malloc(4000)



- blocks get evenly divided into two blocks that are buddies with each other
  - can only merge with your buddy if your buddy is also free
- *internal fragmentation*
  - Ex: malloc(4000)
  - return a 4K block

Buddy Systems

- Faster memory allocation system (at the cost of more fragmentation, internal fragmentation)
  - restrict block size to be a power of 2
    1) all blocks of size 2^k start at location x where $x \bmod 2^k = 0$
    2) given a block starting at location x such that $x \bmod 2^k = 0$
       - $BUDDY_k(x) = x + 2^k$ if $x \bmod 2^{k+1} = 0$
       - $BUDDY_k(x) = x - 2^k$ if $x \bmod 2^{k+1} = 2^k$
       - Ex: $BUDDY_2(1010100) = 1010000$
    3) only buddies can be merged
    4) try to coalesce buddies when storage is deallocated
  - k different available block lists, one for each block size
  - When request a block of size 2^k and none is available:
    1) split smallest block $2^j > 2^k$ into a pair of blocks of size 2^{j-1}
    2) place block on appropriate `free` list and try again

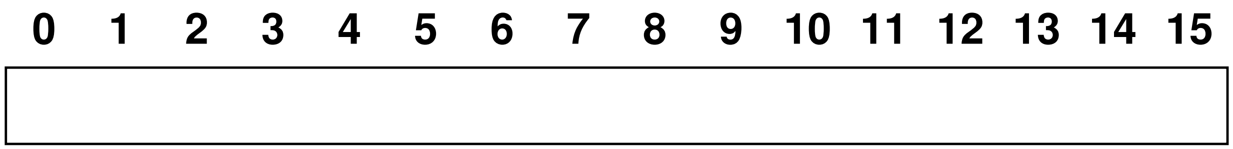
Buddy Systems

- Data Structure
  1) doubly-linked list (not circular) `FREE` list indexed by *k*
     - links stored in actual blocks
     - `FREE`[*k*] points to first available block of size 2^k
  2) each block contains
     - `in-use` bit
     - `size`
     - `NEXT` and `PREV` links for `FREE` list
  - lots of details
    - read `weenix` source code for its "page allocator"

- Can get greater variety in block sizes using Fibonacci sequence of block sizes so $b_j = b_{j-1}+b_{j-2}$
  - ratio of successive block sizes is *2/3* instead of *1/2*

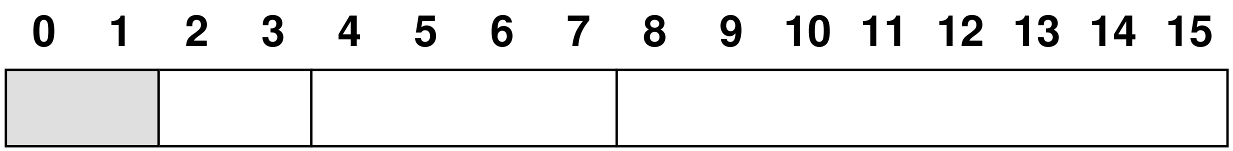
High-level Example of Buddy Algorithm

- Ex: 16 "pages" (minimum allocation is 1 page)



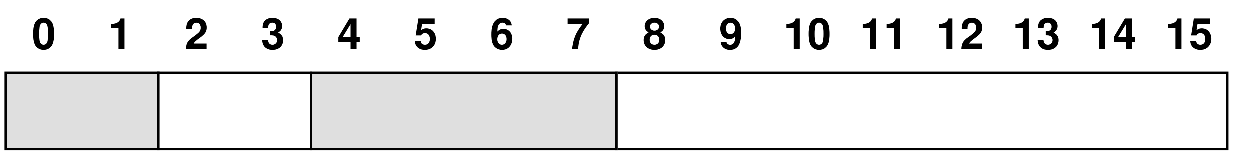
k	free[k]
0	Ω
1	Ω
2	Ω
3	Ω
4	0

1) allocate a block of size 2



k	free[k]
0	Ω
1	~~Ω~~ 2
2	~~Ω~~ 4
3	~~Ω~~ 8
4	~~0~~ Ω

2) allocate a block of size 4



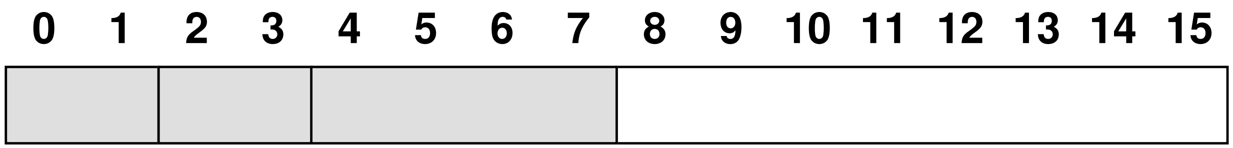
k	free[k]
0	Ω
1	~~Ω~~ 2
2	~~Ω~~ ~~4~~ Ω
3	~~Ω~~ 8
4	~~0~~ Ω

High-level Example of Buddy Algorithm

- Ex: 16 "pages" (minimum allocation is 1 page)

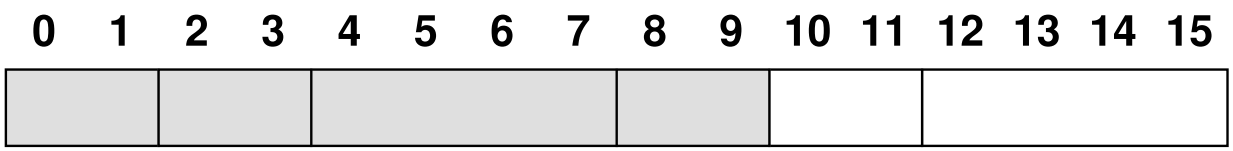
k	free[k]
0	Ω
1	~~Ω~~ ~~2~~ Ω
2	~~Ω~~ ~~4~~ Ω
3	~~Ω~~ 8
4	~~0~~ Ω

3) allocate a block of size 2



k	free[k]
0	Ω
1	~~Ω~~ ~~2~~ 10
2	~~Ω~~ ~~4~~ 12
3	~~Ω~~ ~~8~~ Ω
4	~~0~~ Ω

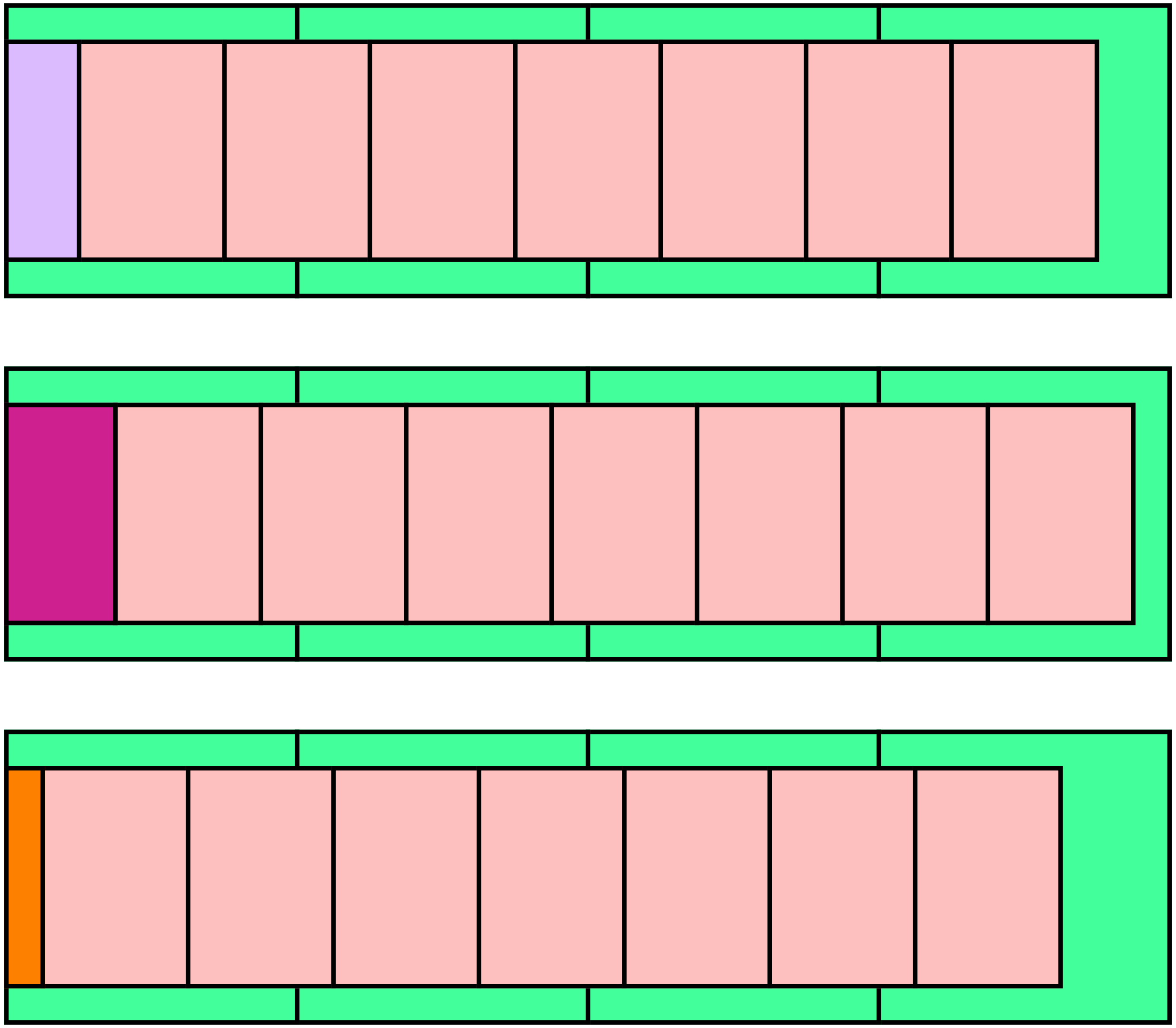
4) allocate a block of size 2



3.3 Dynamic Storage Allocation

- **Best-fit & First-fit Algorithms**
- **Buddy System**
- ***Slab Allocation***

Slab Allocation



- see `weenix` kernel code!

Slab Allocation

- Objects are allocated and freed frequently
  - allocation involves
    - finding an appropriate-sized storage
    - initialize it
      - pointers need to point at the right places
      - may even need to initialize synchronization data structures
  - deallocation involves
    - tearing down the data structures
    - freeing the storage
  - lots of "overhead"

- Difficulties with dynamic storage allocation
  - you cannot predict what an application will ask for
  - but it’s not true for the kernel
    - e.g., can allocate a slab of process control blocks at a time
      - return one of them from a slab

Slab Allocation

- ***Slab Allocation***
  - sets up a separate cache for each type of object to be managed
  - contiguous sets of pages called ***slabs***, allocated to hold objects
    - we will cover "pages" later, won't get into too much detail now
- Whenever a ***slab*** is allocated, a constructor is called to initialize all the objects it hold
  - this is where you pay for initialization, but it's done in a ***batch***
- As ***objects*** are being allocated, they are taken from the set of existing slabs in the cache
  - objects are considered "preallocated" since they have all been initialized already
- As ***objects*** are being freed, they are simply marked as free
  - don't have to free up storage
  - when appropriate can free up an entire slab