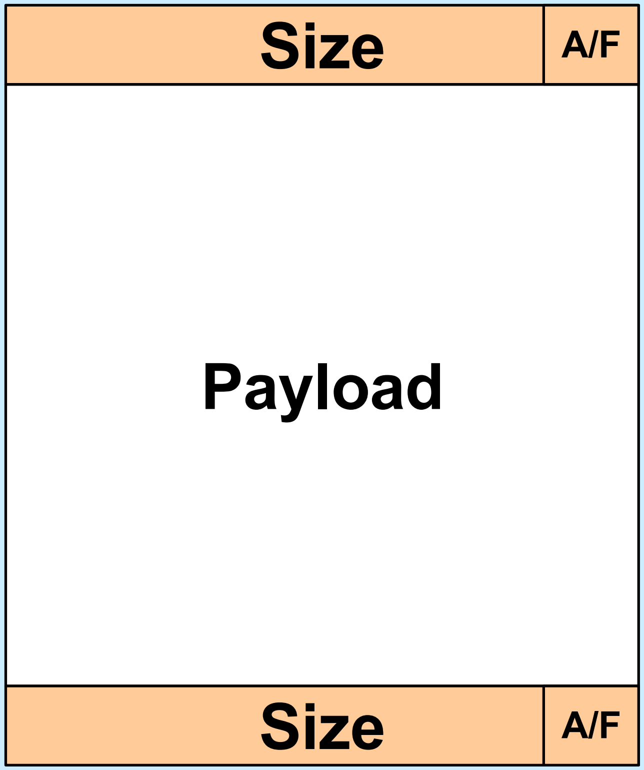
# CS 33

## Storage Allocation

# Data Structure Requirements

- All blocks
  - – we need to know how big they are
    - » when free is called, it must be known how much to free
    - » when looking at a free block in malloc, we need to know its size
  - – we need to know which they are: free or allocated
    - » needed for coalescing
- Free blocks
  - – they need to be linked into the free list

# Solution: Boundary Tags

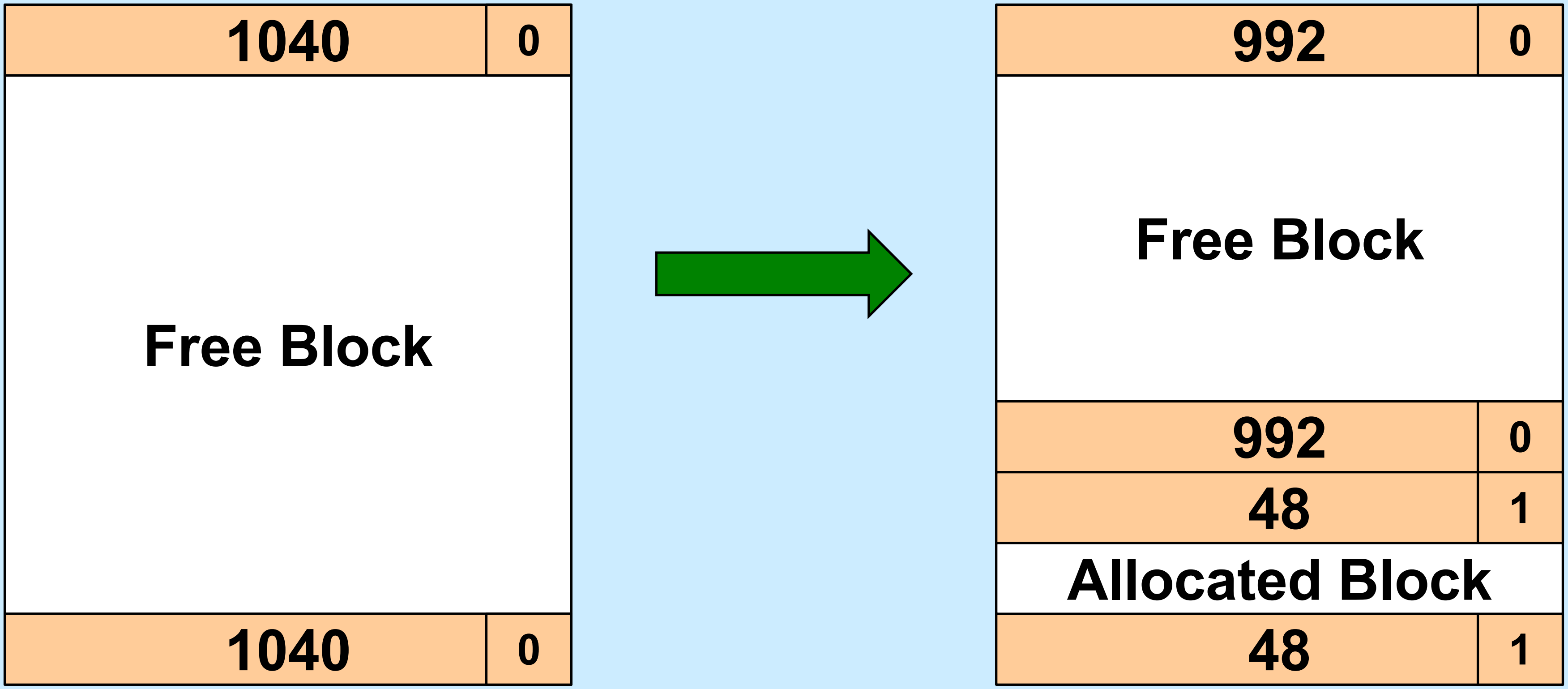


Size A/F

Payload

Size A/F

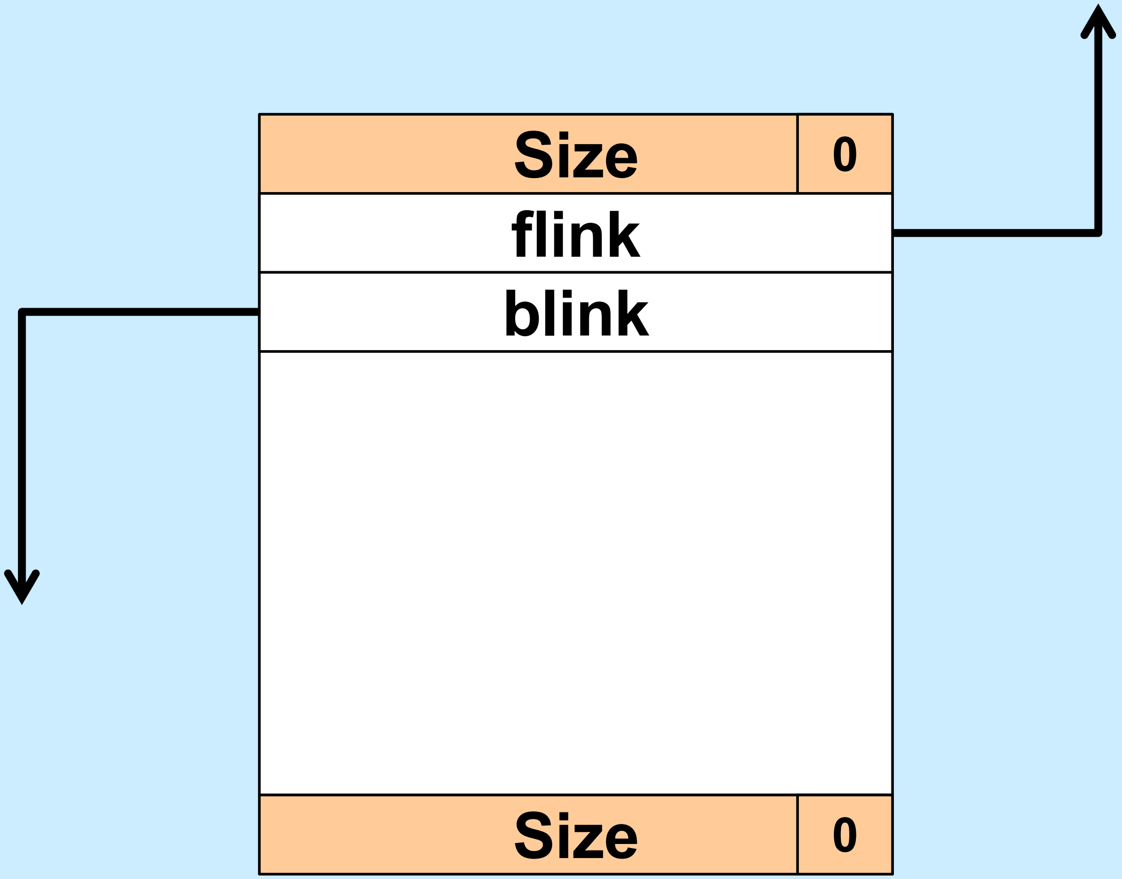
# Splitting a Block



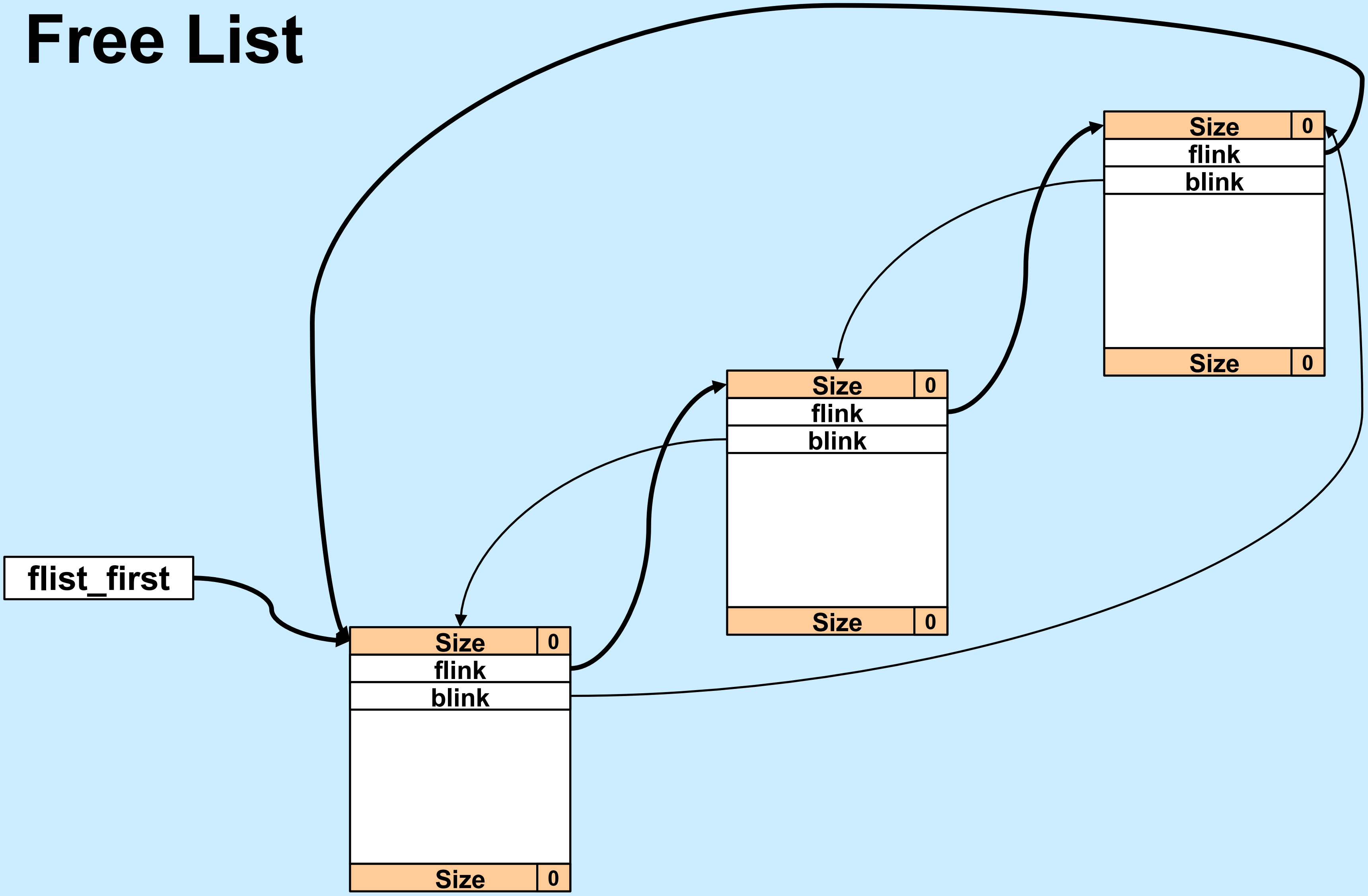
# Representing the Free List

- We need a pointer to the first element
  - *flist_first*
- We need to traverse the list from beginning to end
  - required by malloc
- We need to merge adjacent blocks
  - this may require removing a block from the free list, then reinserting it (as part of a coalesced block)
- Links may be put in the free block's payload area
  - not needed for allocated blocks!

# Free Block Representation



# Free List


flist_first
Size
0
flink
blink
Size
0
Size
0
flink
blink
Size
0
Size
0
flink
blink
Size
0

# Quiz 1

Why is the free list doubly linked?

a) we don’t really need it to be doubly linked for malloc and free, but it may be necessary for some future operations

b) so that, given a pointer to an arbitrary free block, we can easily remove the block from the list

c) to facilitate sorting the free list

d) so we can traverse it in both directions

# Quiz 2

Why is the free list circular?

a) so that we don’t have to special-case the the handling of the first and last list elements

b) to facilitate implementing the next-fit search strategy

c) both of the above

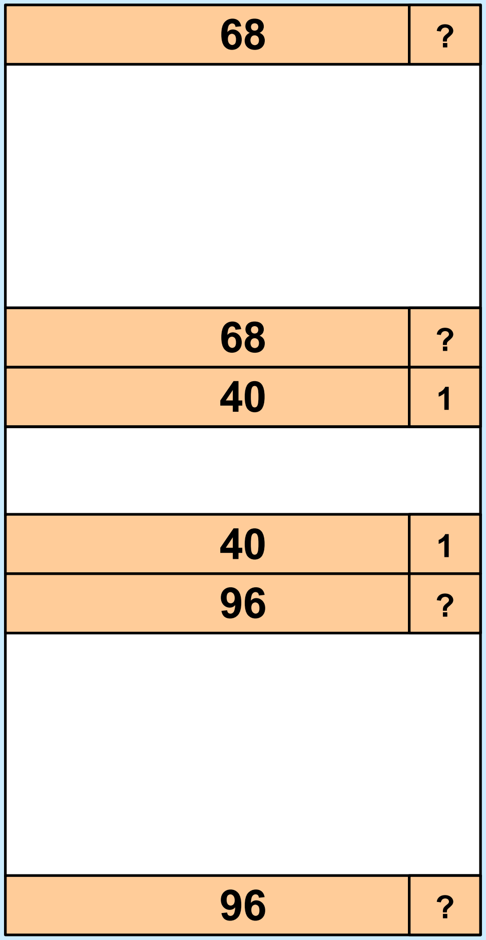
d) none of the above

# Heap ≠ Free List

- Heap
  - collection of all memory usable as dynamic storage: the dynamic portion of the address space
    - both allocated and free
- Free list
  - those blocks of the heap that are free
    - linked together (circular, doubly)

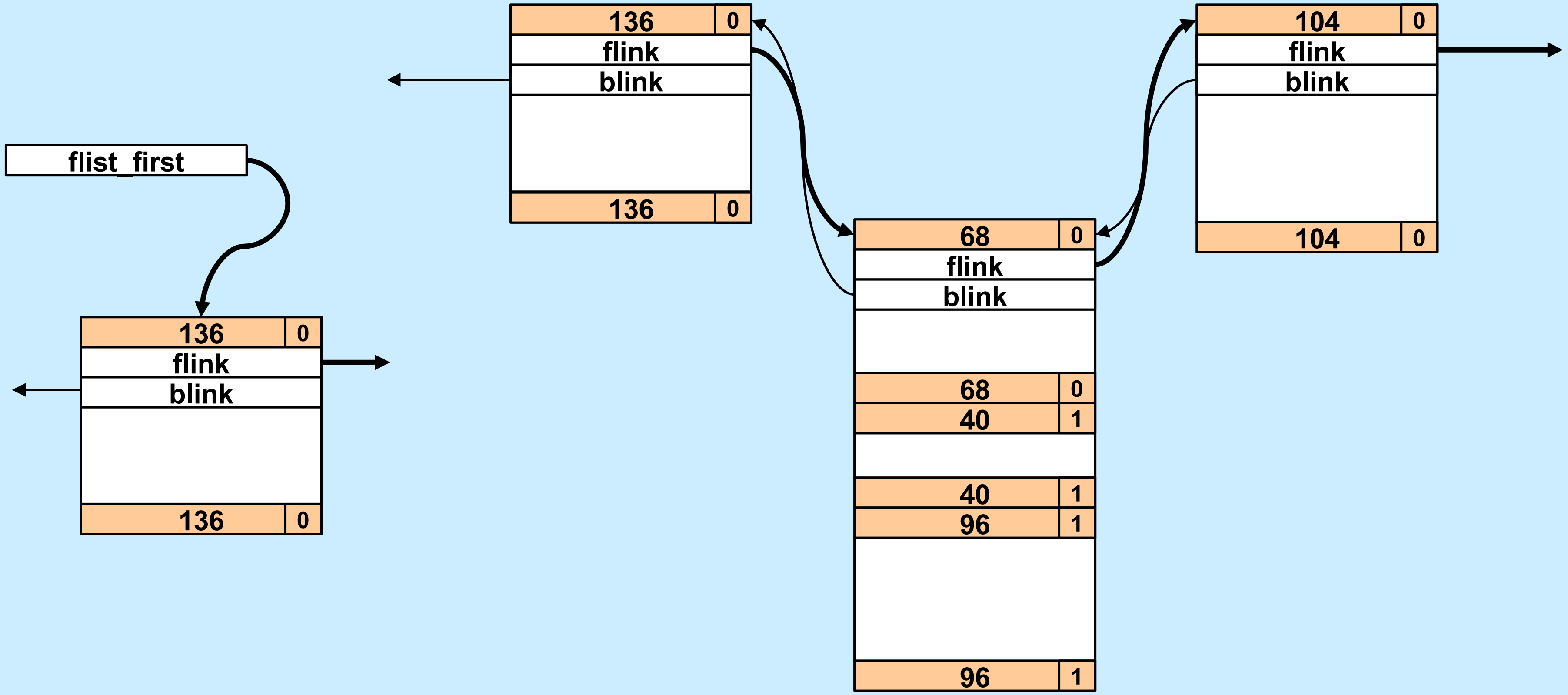
- Both important, but different
- Confusion: what does *next block* mean?
  - next adjacent block (next in heap)
  - next free block (next in free list)

# Coalescing Revisited

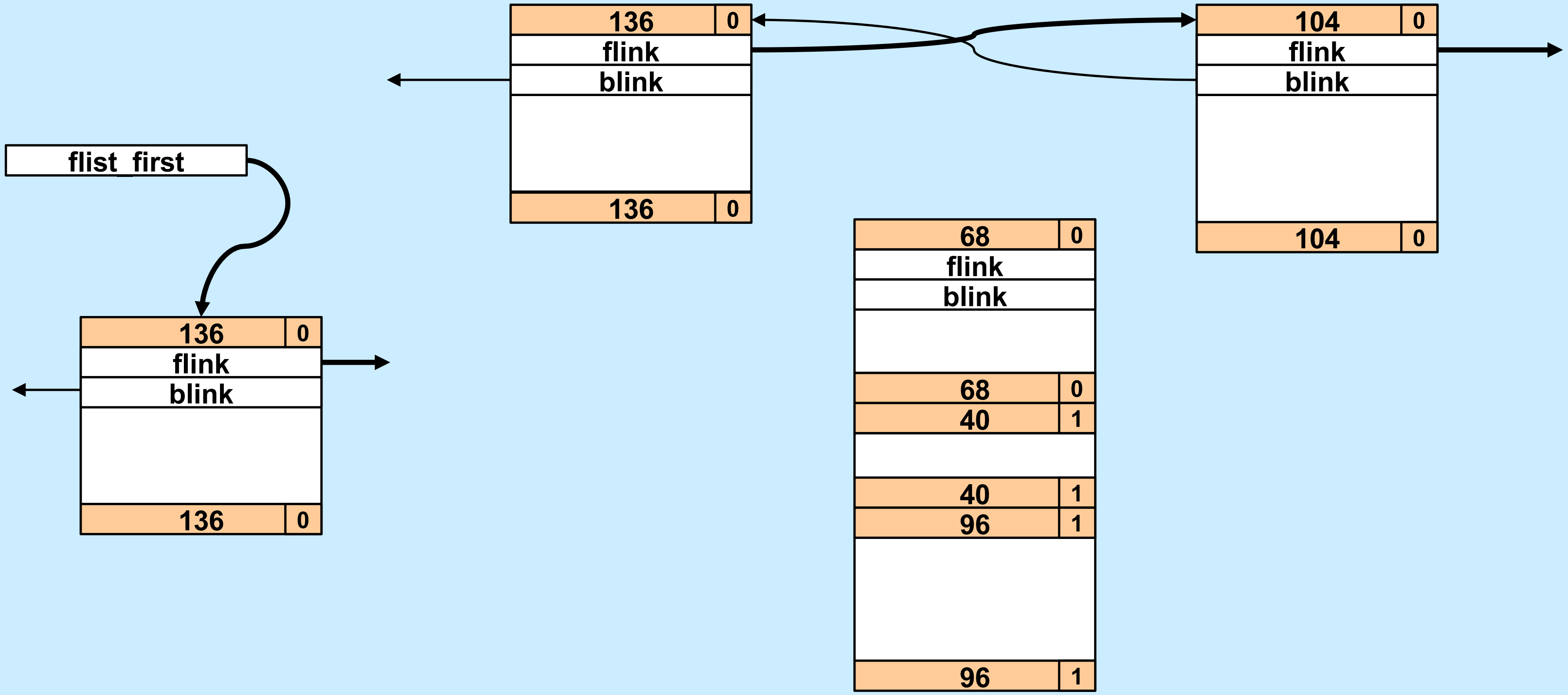


- We are freeing a block
  - is the previous block free?
  - is the next block free?
  - are both free?

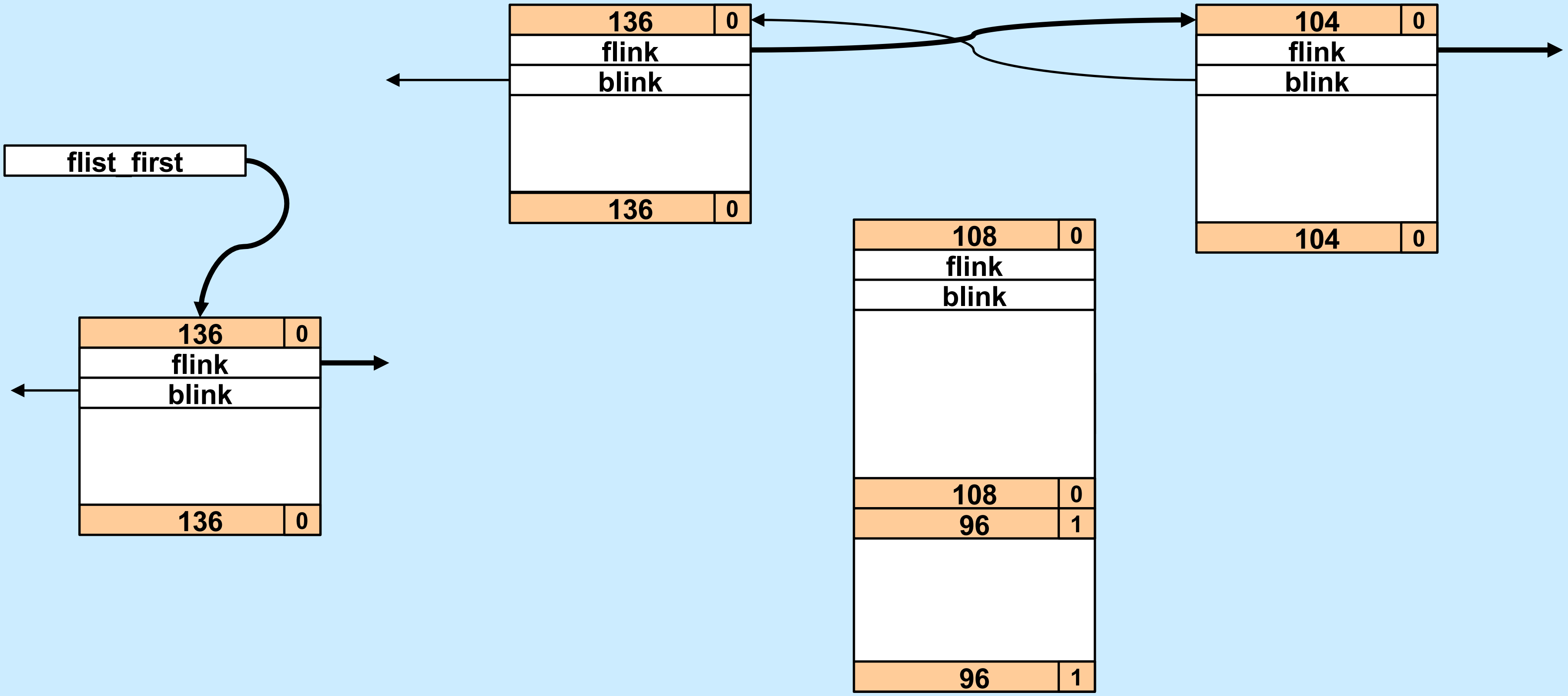
# Coalescing: Previous Free (1)



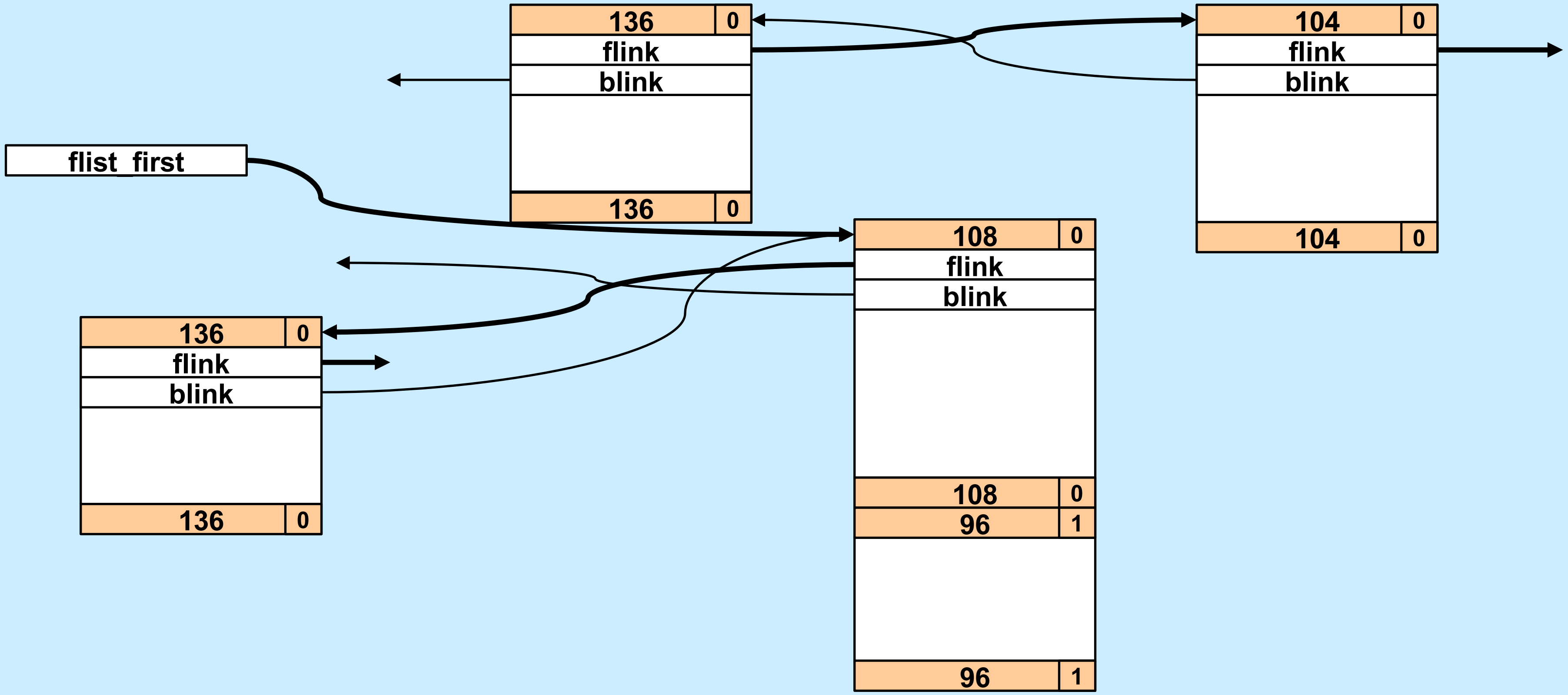
# Coalescing: Previous Free (2)



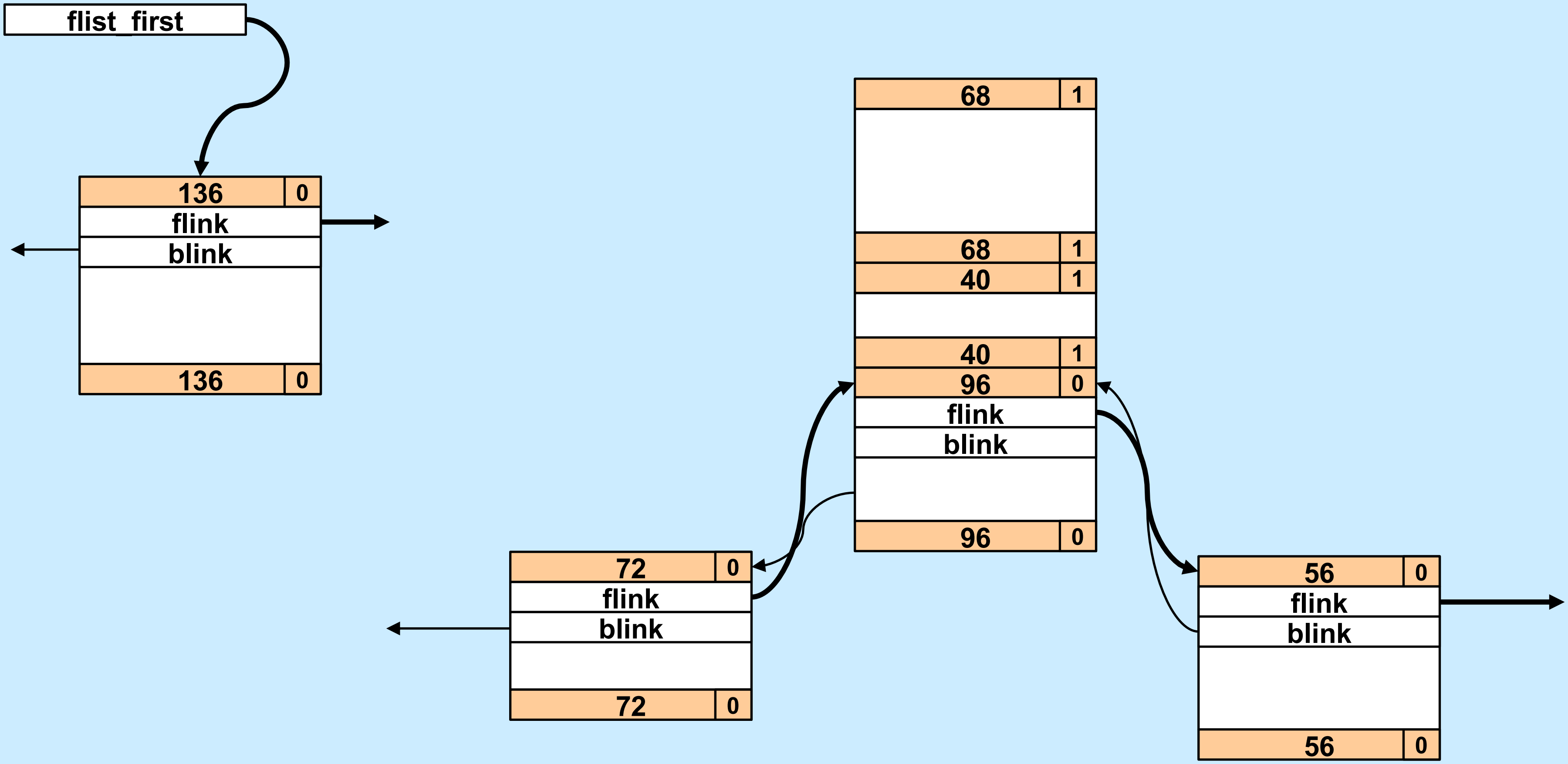
# Coalescing: Previous Free (3)



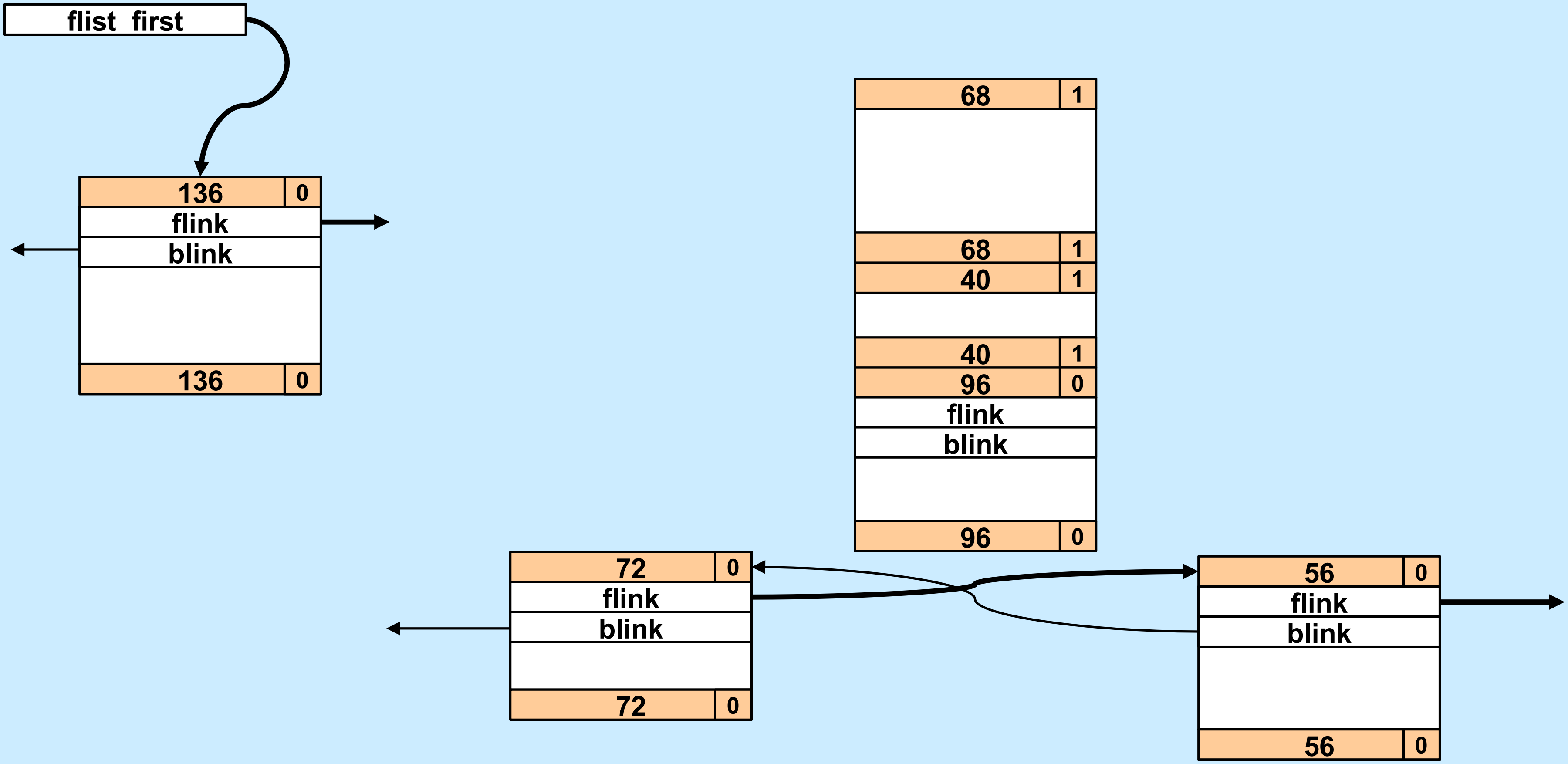
# Coalescing: Previous Free (4)



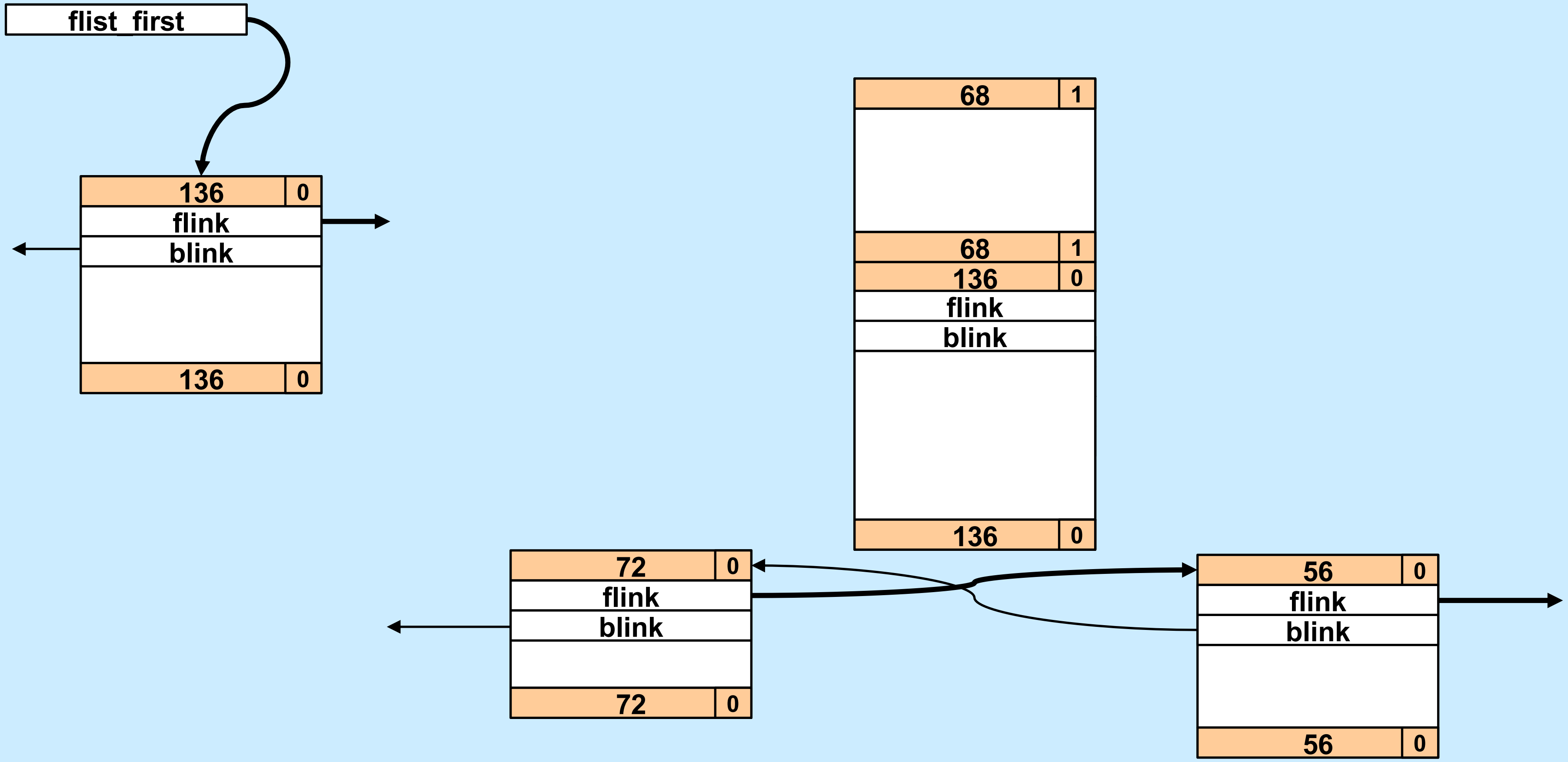
# Coalescing: Next Free (1)



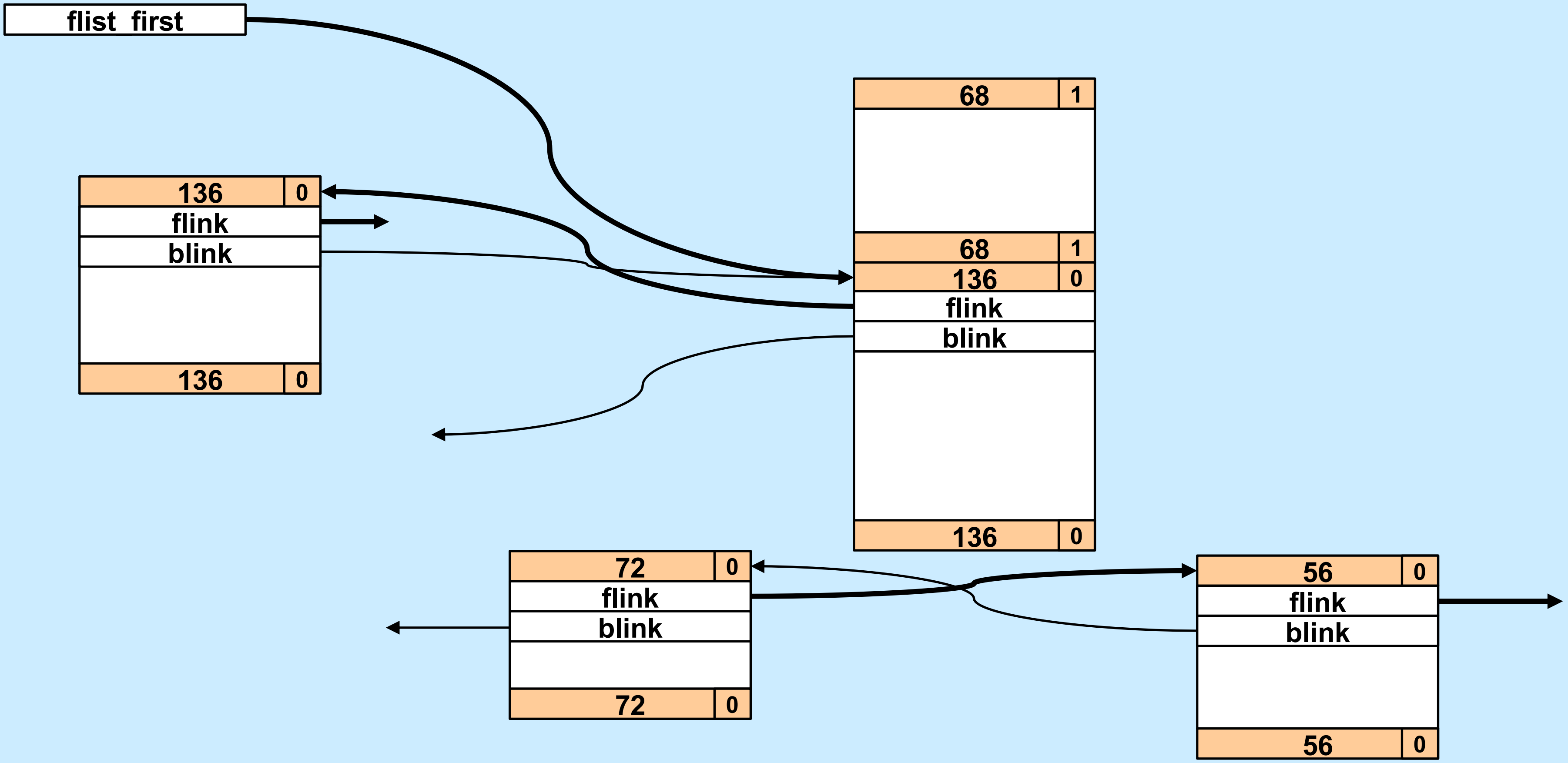
# Coalescing: Next Free (2)



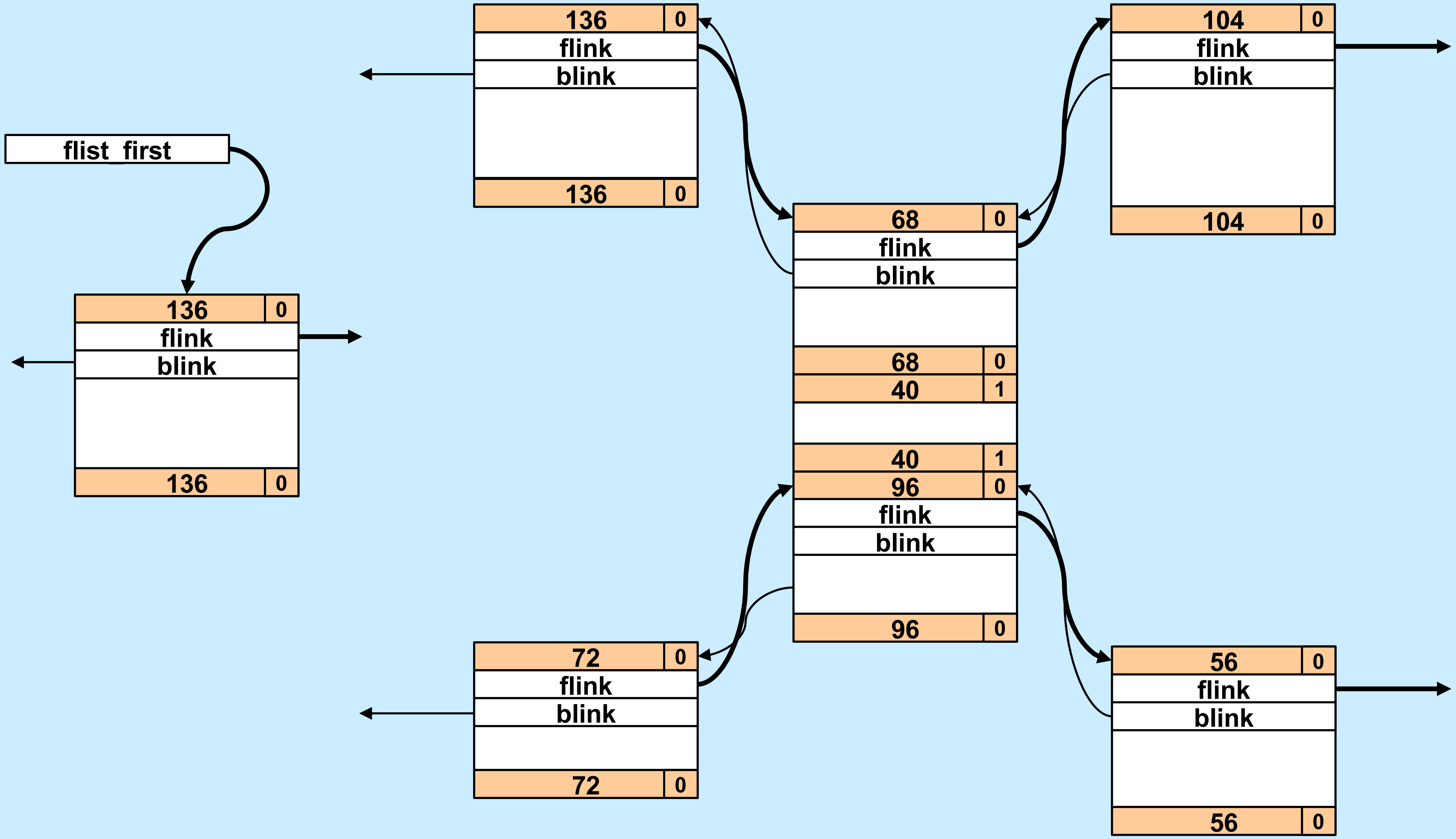
# Coalescing: Next Free (3)



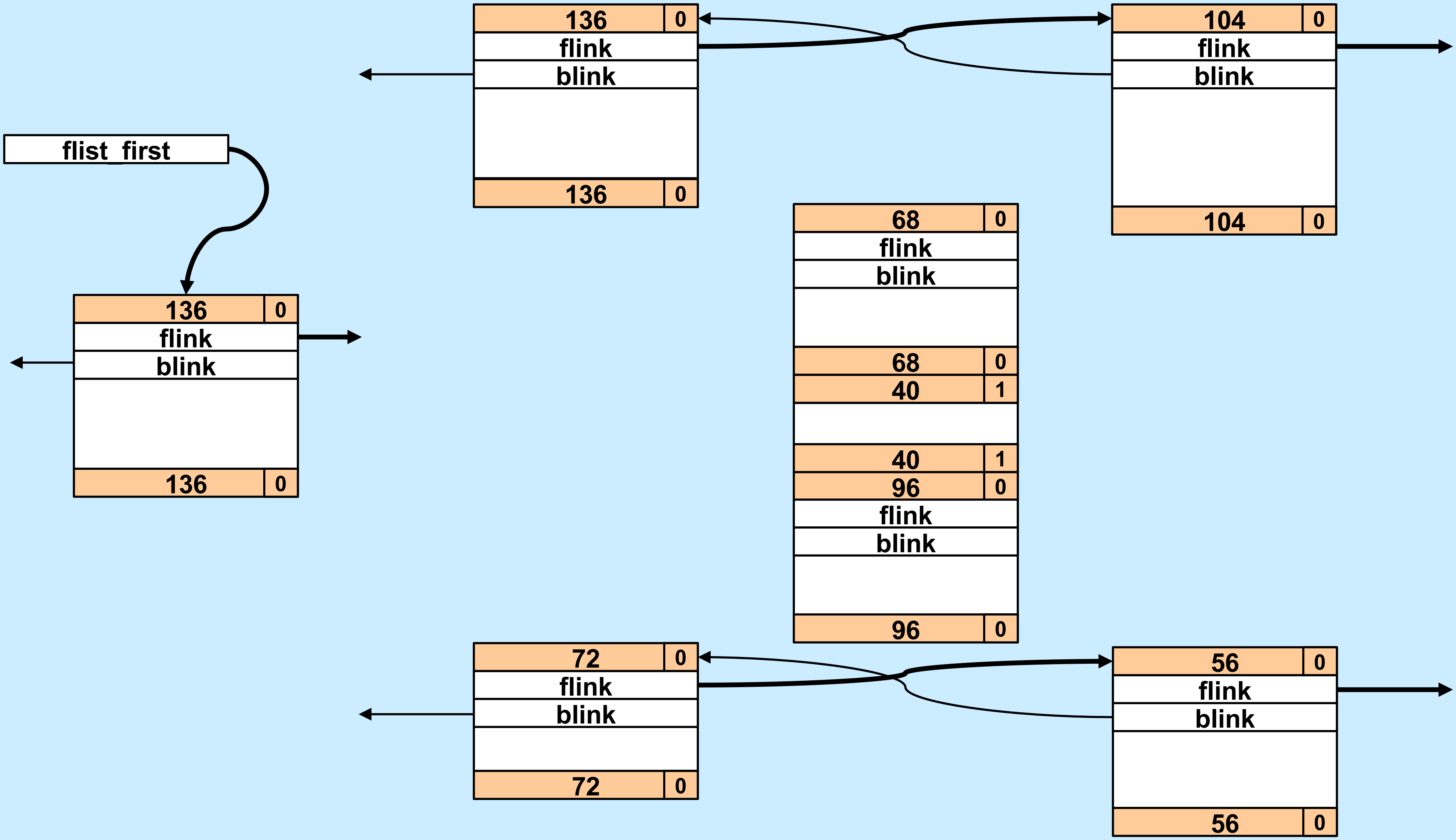
# Coalescing: Next Free (4)



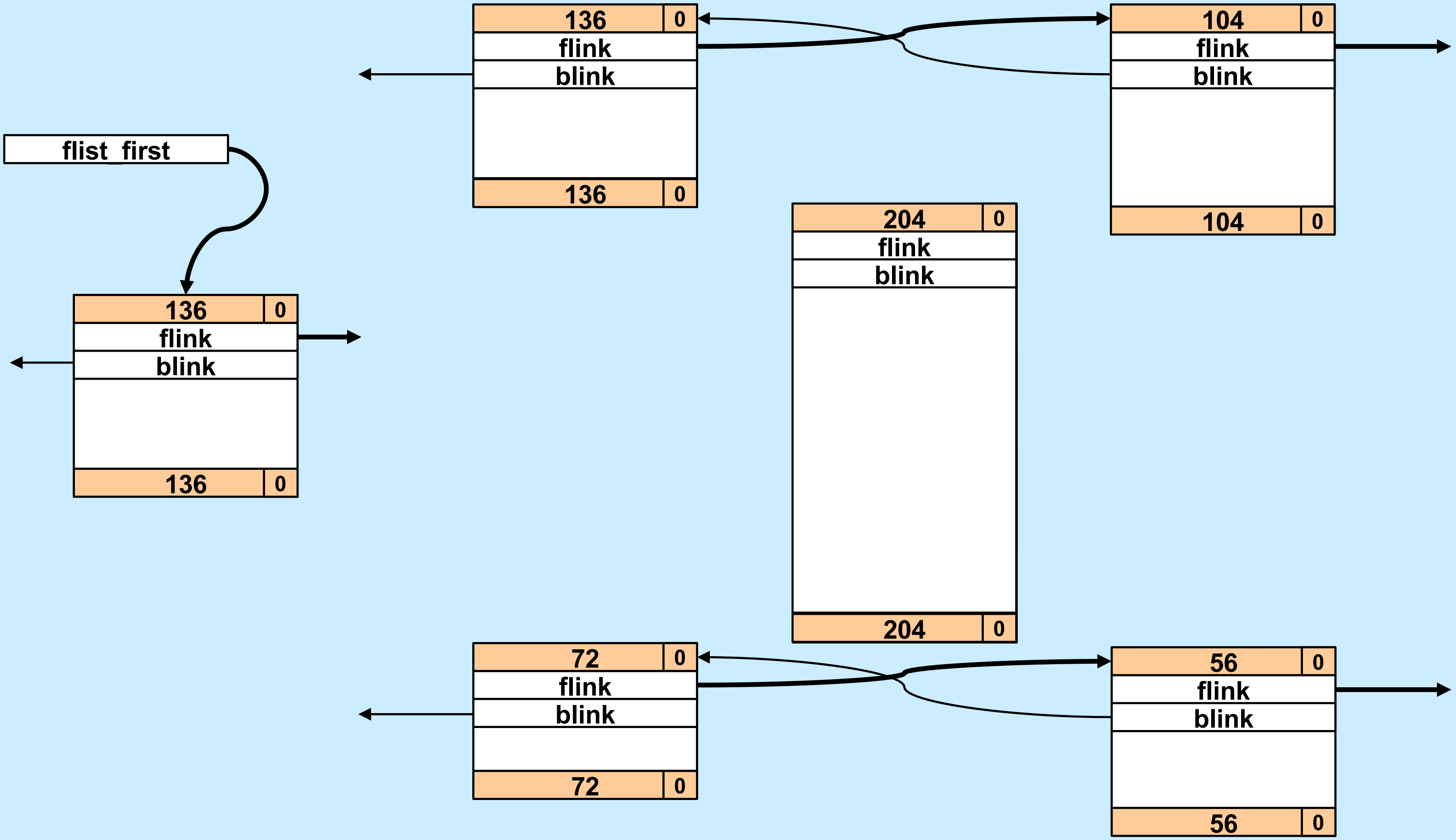
# Coalescing: Both Free (1)



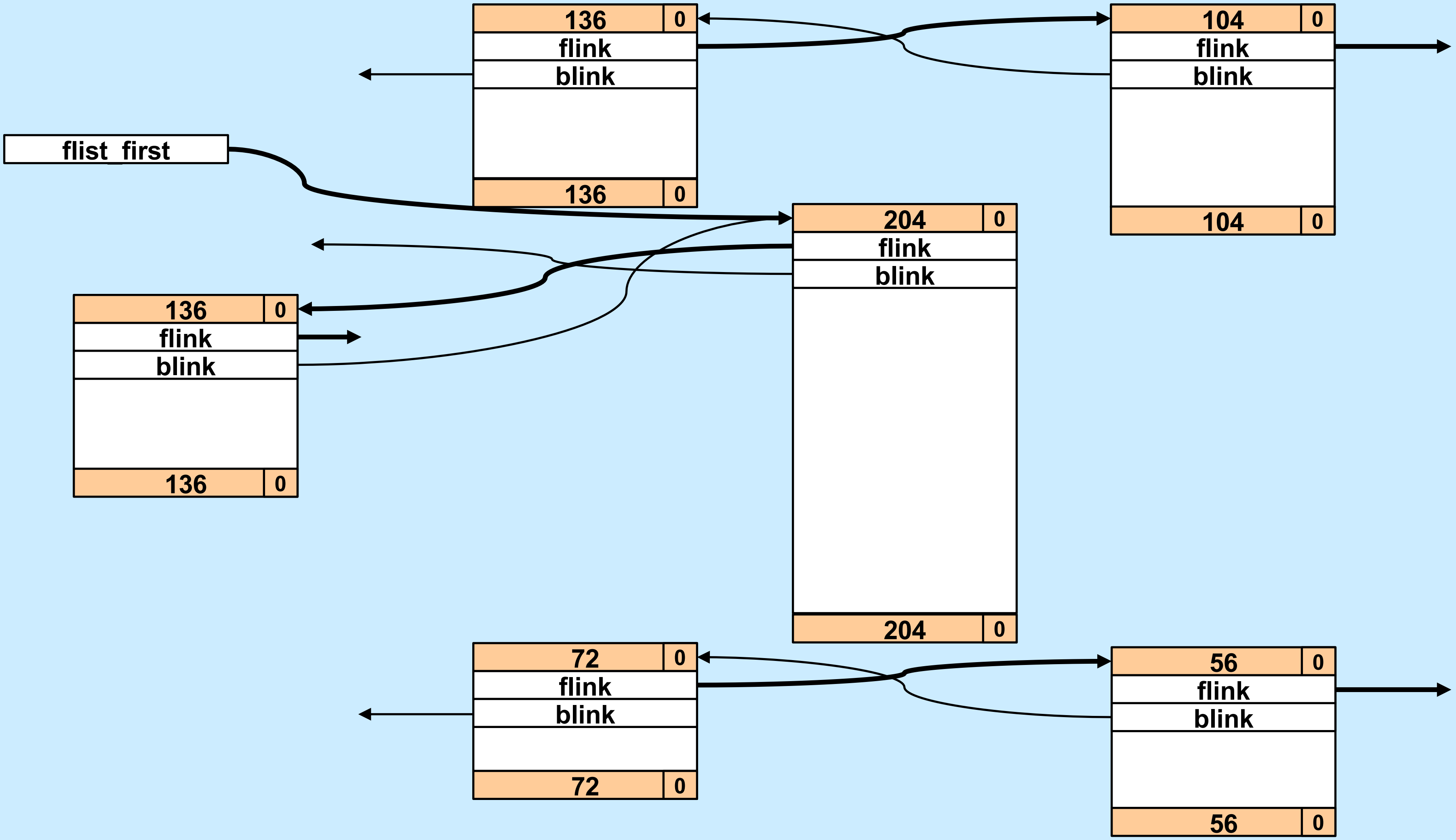
# Coalescing: Both Free (2)



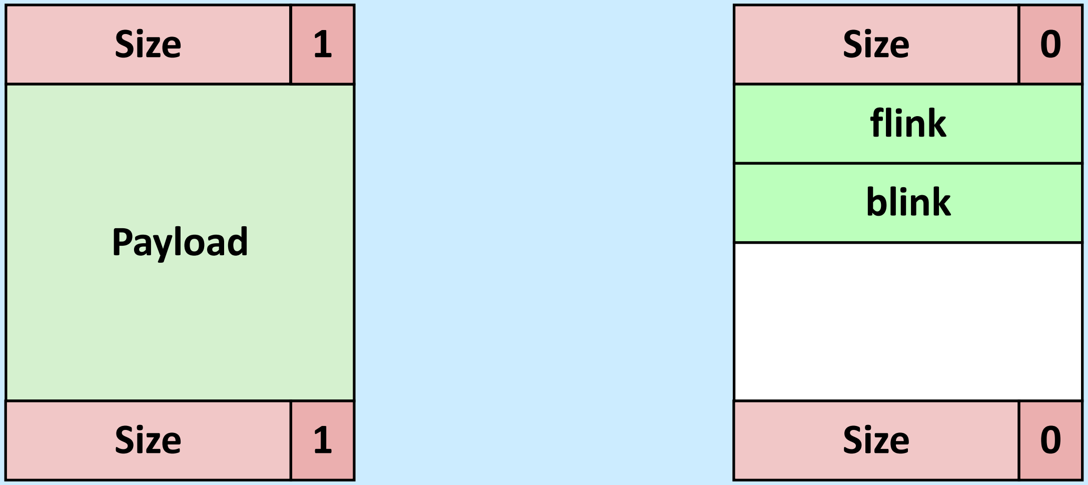
# Coalescing: Both Free (3)



# Coalescing: Both Free (4)



# C vs. Storage Allocation



```c
typedef struct block {
  long size;
  long payload[size/8 - 2];
  long end_size;
} block_t;
```

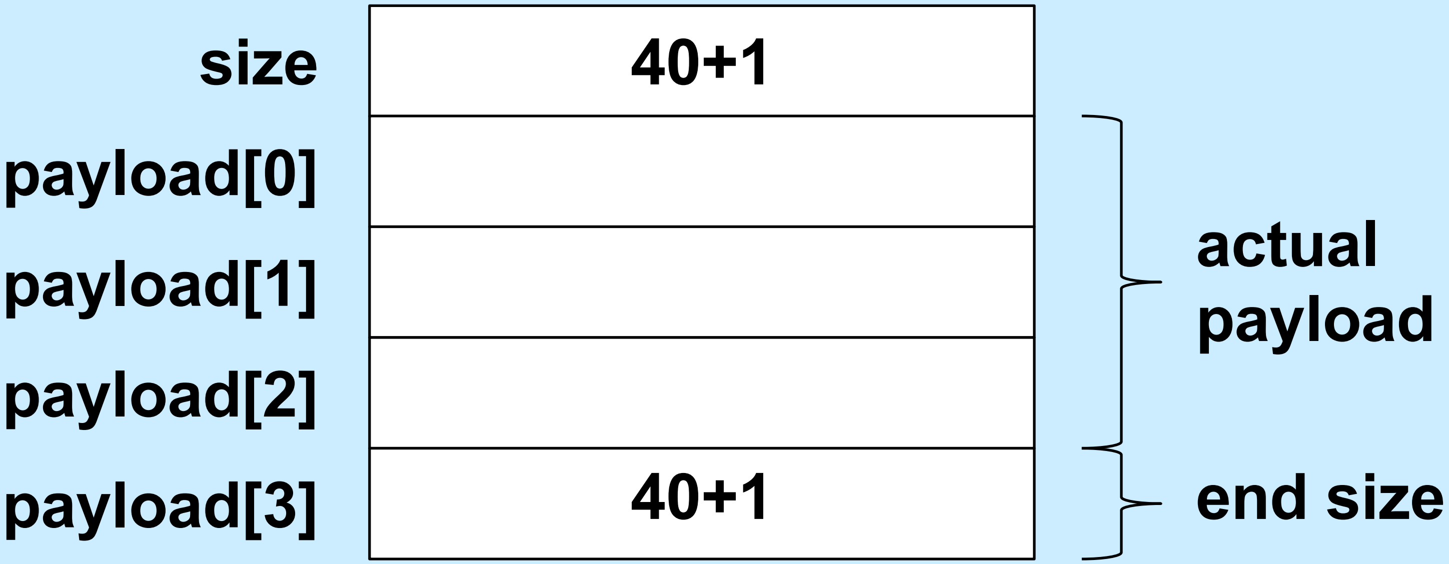
```c
typedef struct free_block {
  long size;
  struct free_block *flink;
  struct free_block *blink;
  long filler[size/8 - 4];
  long end_size;
} free_block_t;
```

# Overcoming C

- Think objects
  - a block is an object
    - opaque to the outside world
  - define accessor functions to get and set its contents

```
typedef struct block {
  size_t size;
  size_t payload[0];
} block_t;
```

# Allocated Block



size 40+1

payload[0]

payload[1]

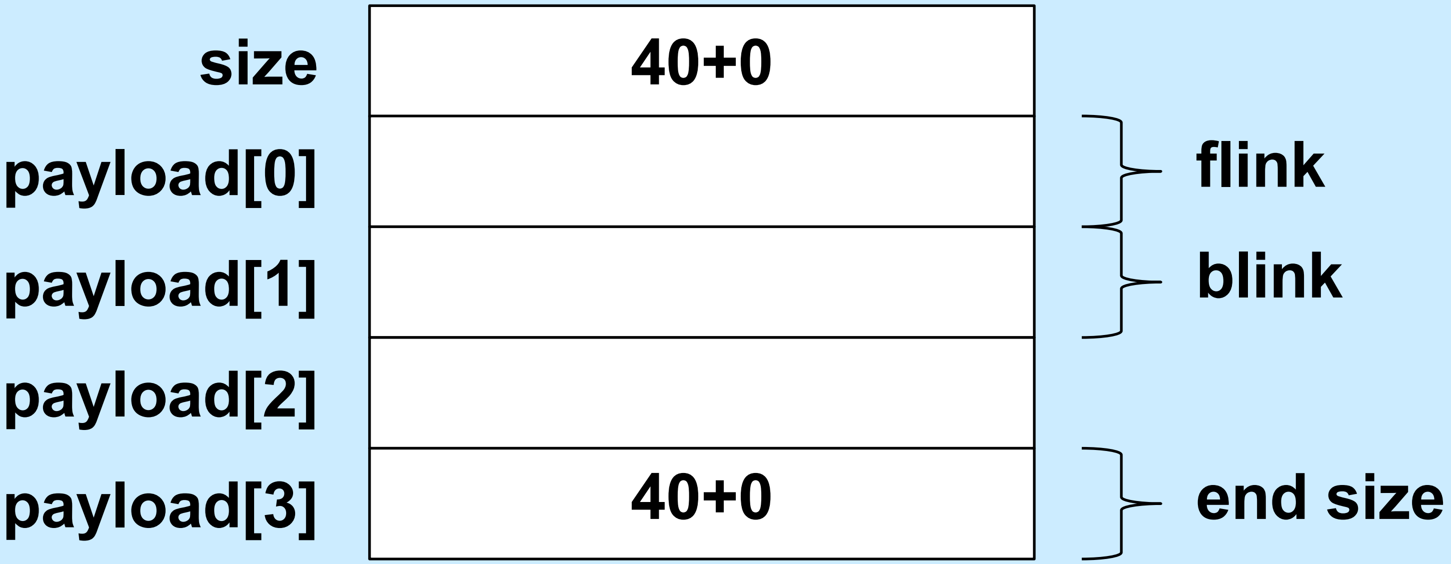
payload[2]

payload[3] 40+1

actual payload

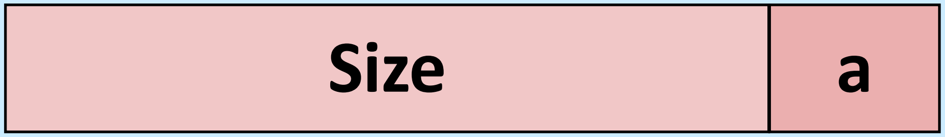
end size

# Free Block



- In general, end size is at *payload[size/8 – 2]*

# Overloading Size

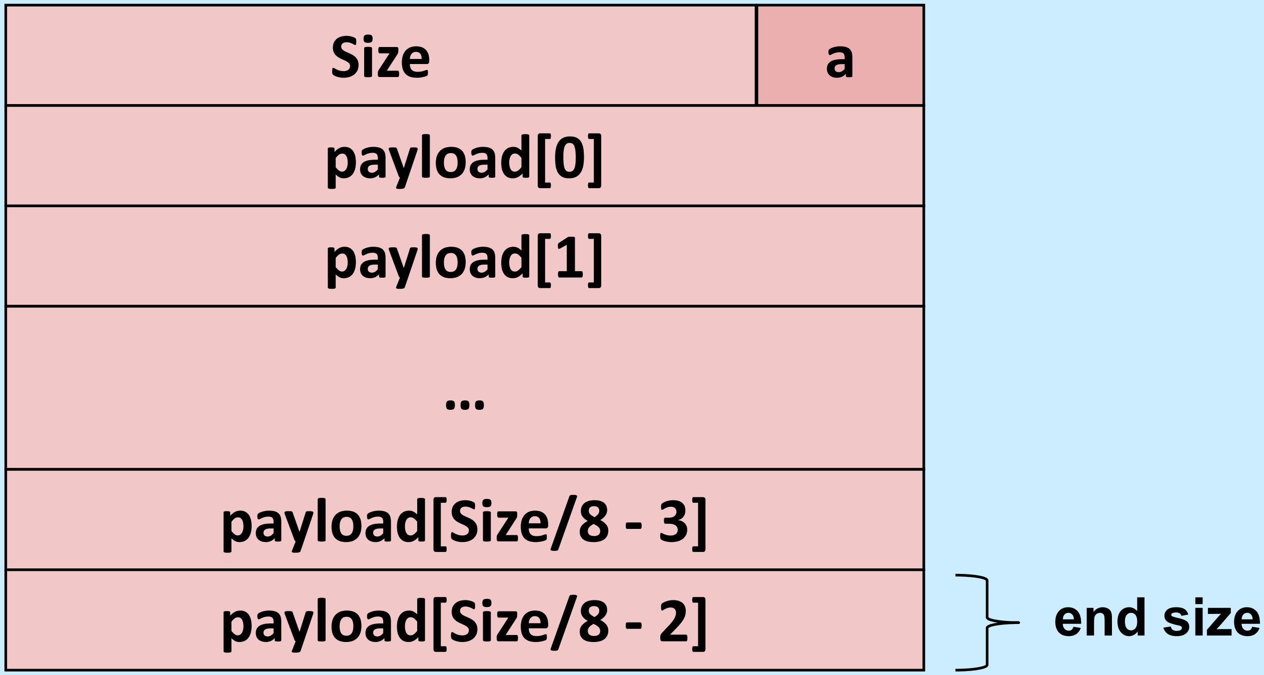


Size

a

```
size_t block_allocated(block_t *b) {
  return b->size & 1;
}

size_t block_size(block_t *b) {
  return b->size & -2;
}
```

# End Size



Size | a

payload[0]

payload[1]

…

payload[Size/8 - 3]

payload[Size/8 - 2]

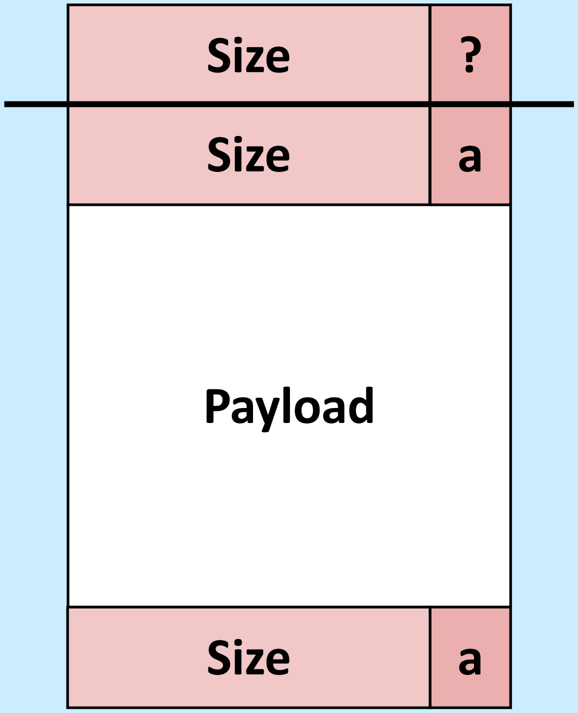
end size

```
size_t *block_end_tag(block_t *b) {
  return &b->payload[b->size/8 - 2];
}
```

# Setting the Size

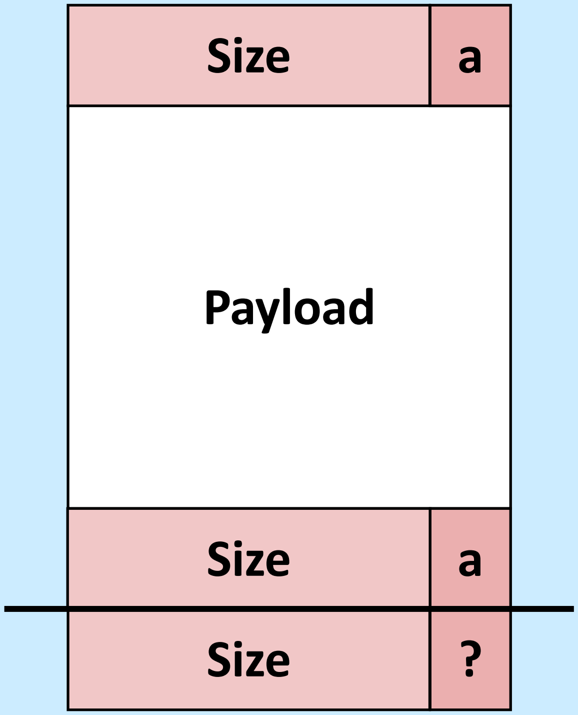
```
void block_set_size(block_t *b, size_t size) {
  assert(!(size & 7));          // multiple of 8
  size |= block_allocated(b);   // preserve alloc bit
  b->size = size;
  *block_end_tag(b) = size;
}

void block_set_allocated(block_t *b, size_t a) {
  assert((a == 0) || (a == 1));
  if (a) {
    b->size |= 1;
    *block_end_tag(b) |= 1;
  } else {
    b->size &= -2;
    *block_end_tag(b) &= -2;
  }
}
```

# Is Previous Adjacent Block Free?



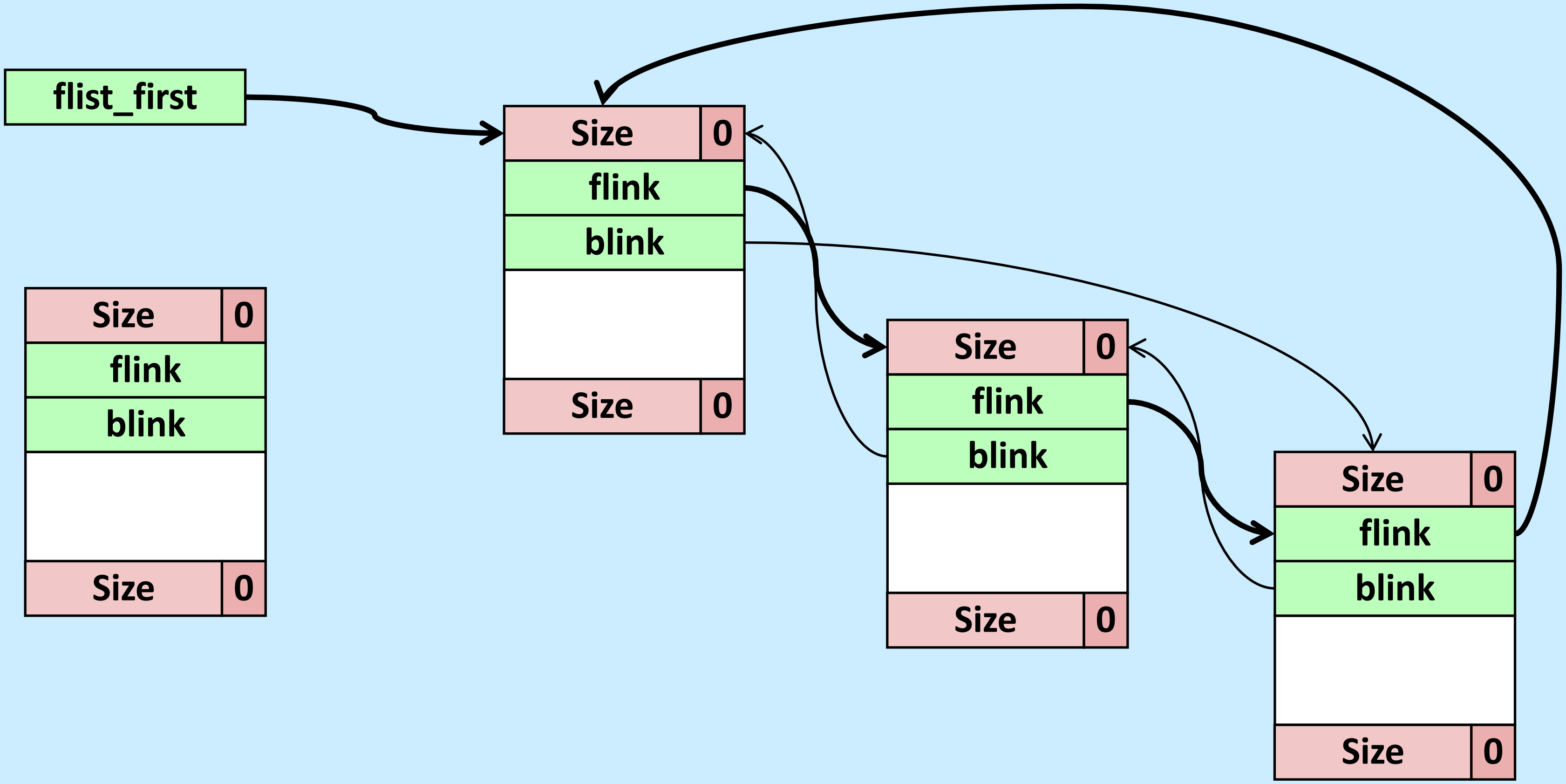
```
size_t block_prev_allocated(
    block_t *b) {
  return b->payload[-2] & 1;
}
```

# Is Next Adjacent Block Free?

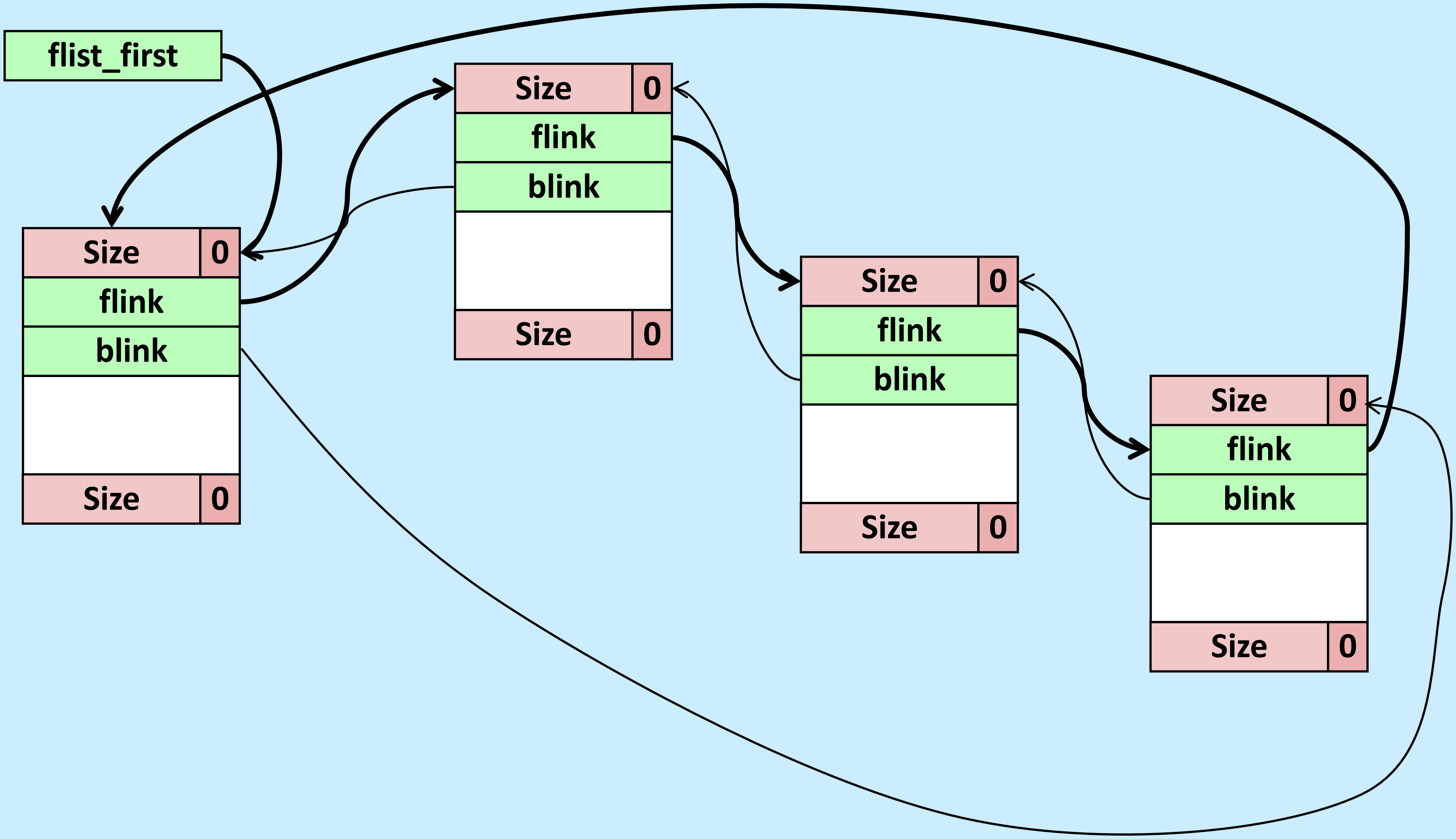


```
block_t *block_next(
    block_t *b) {
  return (block_t *)
    ((char *)b + block_size(b));
}

size_t block_next_allocated(
    block_t *b) {
  return block_allocated(
    block_next(b));
}
```

# Adding a Block to the Free List (1)



# Adding a Block to the Free List (2)



# Accessing the Object

```
block_t *block_flink(block_t *b) {
  return (block_t *)b->payload[0];
}

void block_set_flink(block_t *b, block_t *next) {
  b->payload[0] = (size_t)next;
}

block_t *block_blink(block_t *b) {
  return (block_t *)b->payload[1];
}

void block_set_blink(block_t *b, block_t *next) {
  b->payload[1] = (size_t)next;
}
```

# Insertion Code

```
void insert_free_block(block_t *fb) {
  assert(!block_allocated(fb));
  if (flist_first != NULL) {
    block_t *last =
      block_blink(flist_first);
    block_set_flink(fb, flist_first);
    block_set_blink(fb, last);
    block_set_flink(last, fb);
    block_set_blink(flist_first, fb);
  } else {
    block_set_flink(fb, fb);
    block_set_blink(fb, fb);
  }
  flist_first = fb;
}
```

# Performance

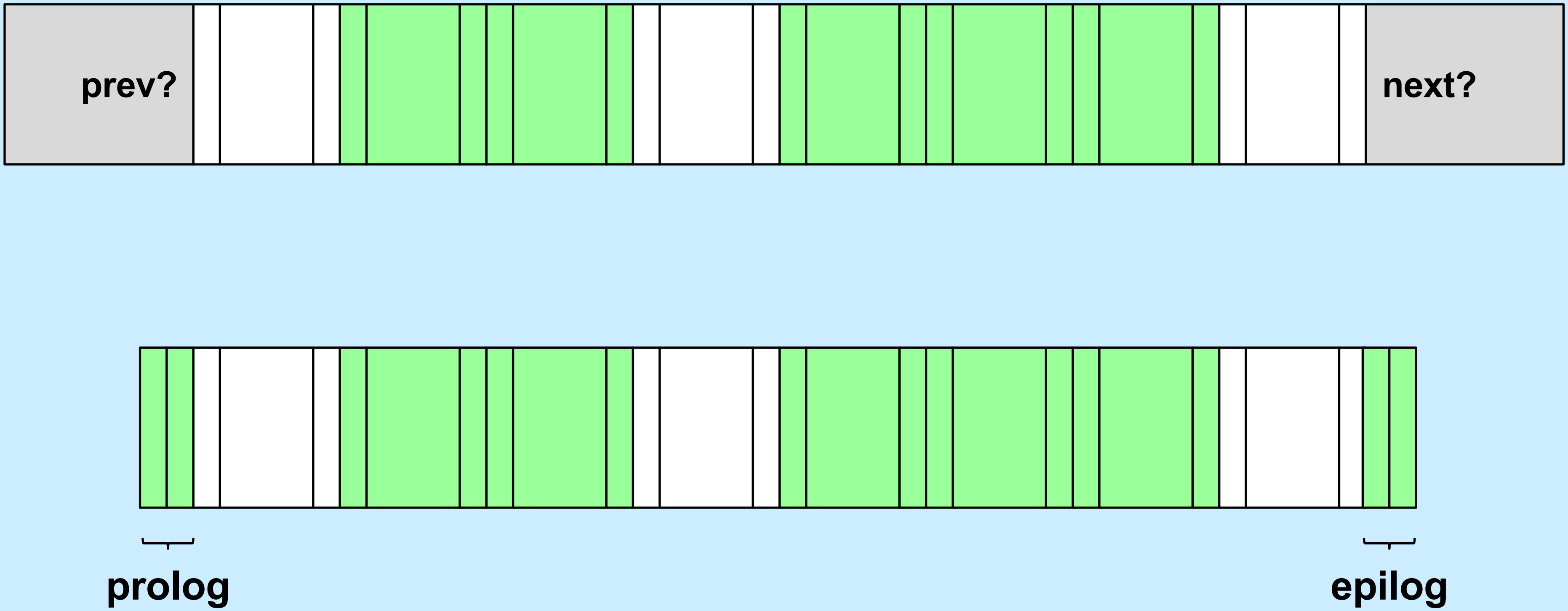
- Won't all the calls to the accessor functions slow things down a lot?
  - yes — not just a lot, but tons
- Why not use macros (#define) instead?
  - the textbook does this
  - it makes the code impossible to debug
    - » gdb shows only the name of the macro, not its body
- What to do????

# Inline Functions

```
static inline size_t block_size(
      block_t *b) {
   return b->size & -2;
}
```

- when debugging (–O0), the code is implemented as a normal function
  - » easy to debug with gdb
- when optimized (–O1, –O2), calls to the function are replaced with the body of the function
  - » no function-call overhead

# Prolog and Epilog



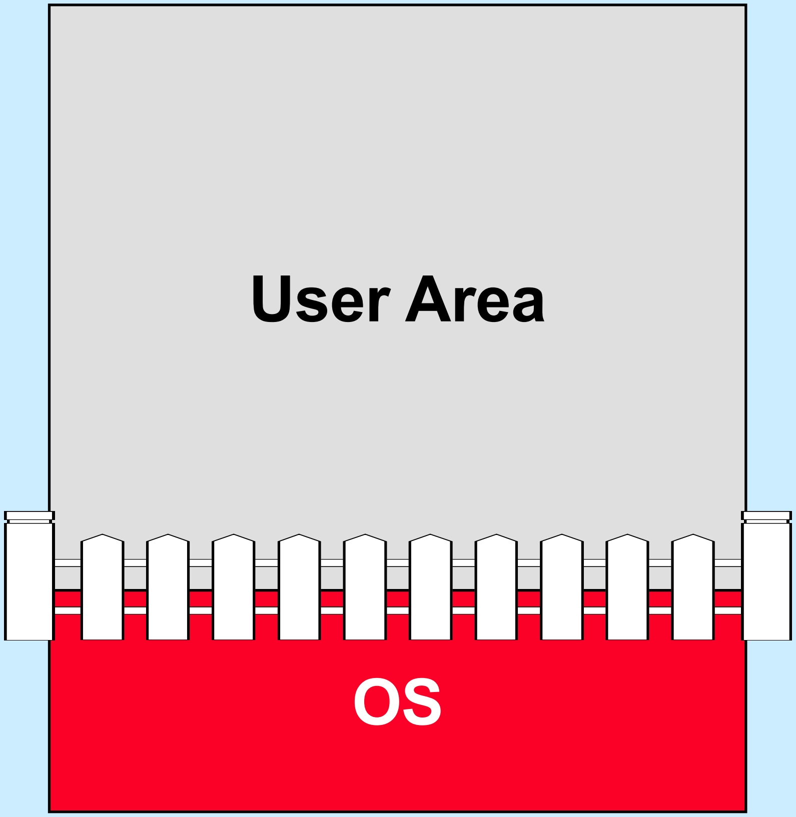
# CS 33

## Virtual Memory

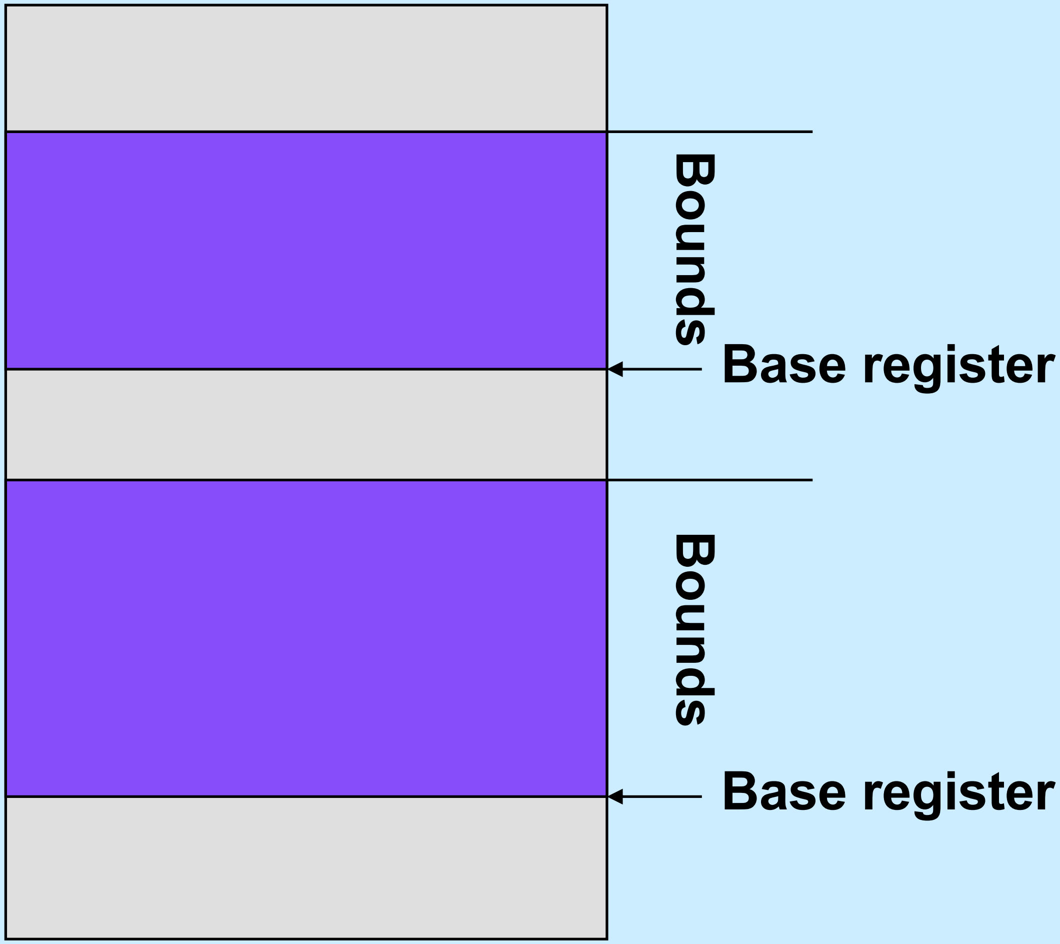
# The Address-Space Concept

- Protect processes from one another
- Protect the OS from user processes
- Provide efficient management of available storage

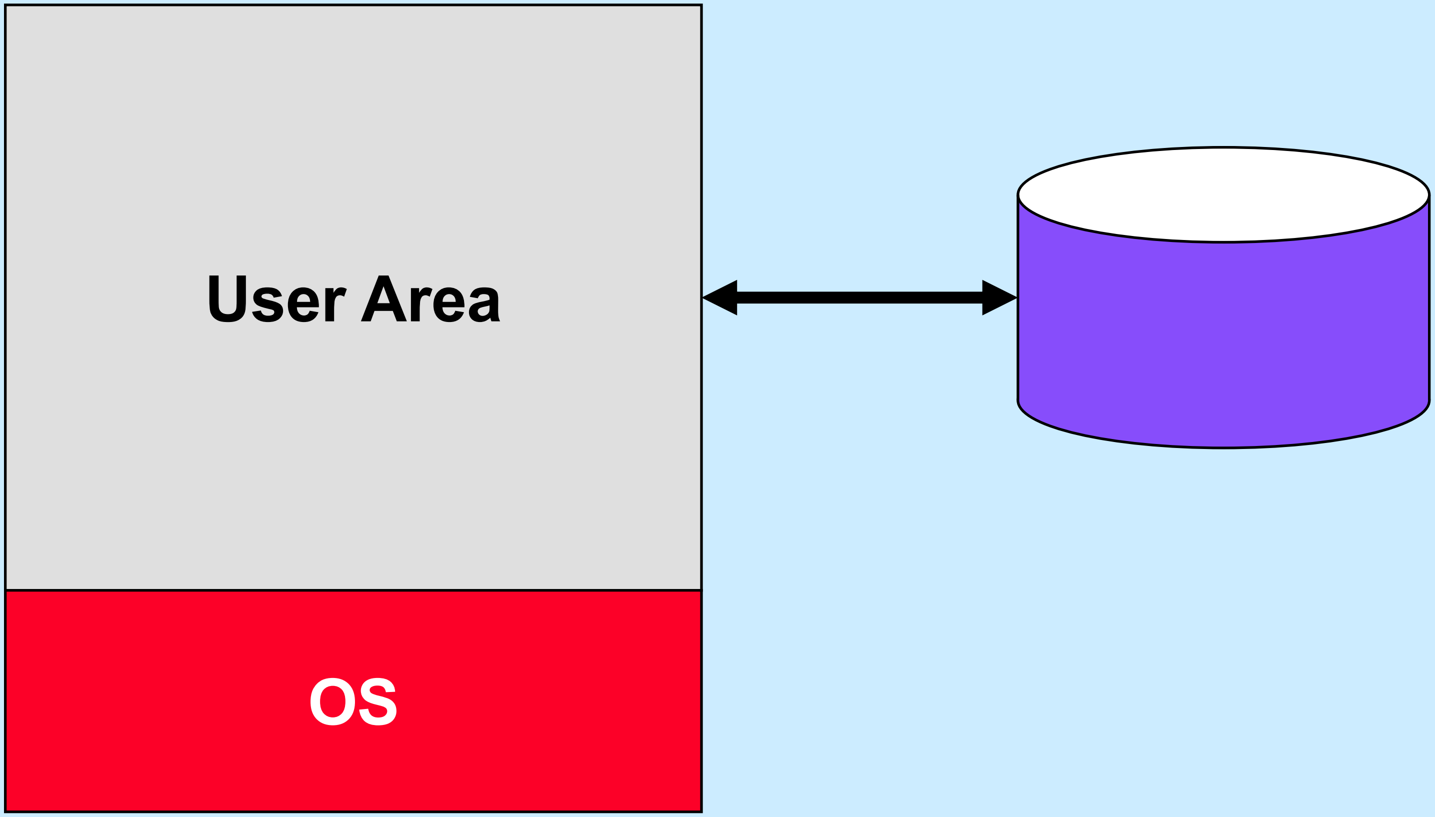
# Memory Fence



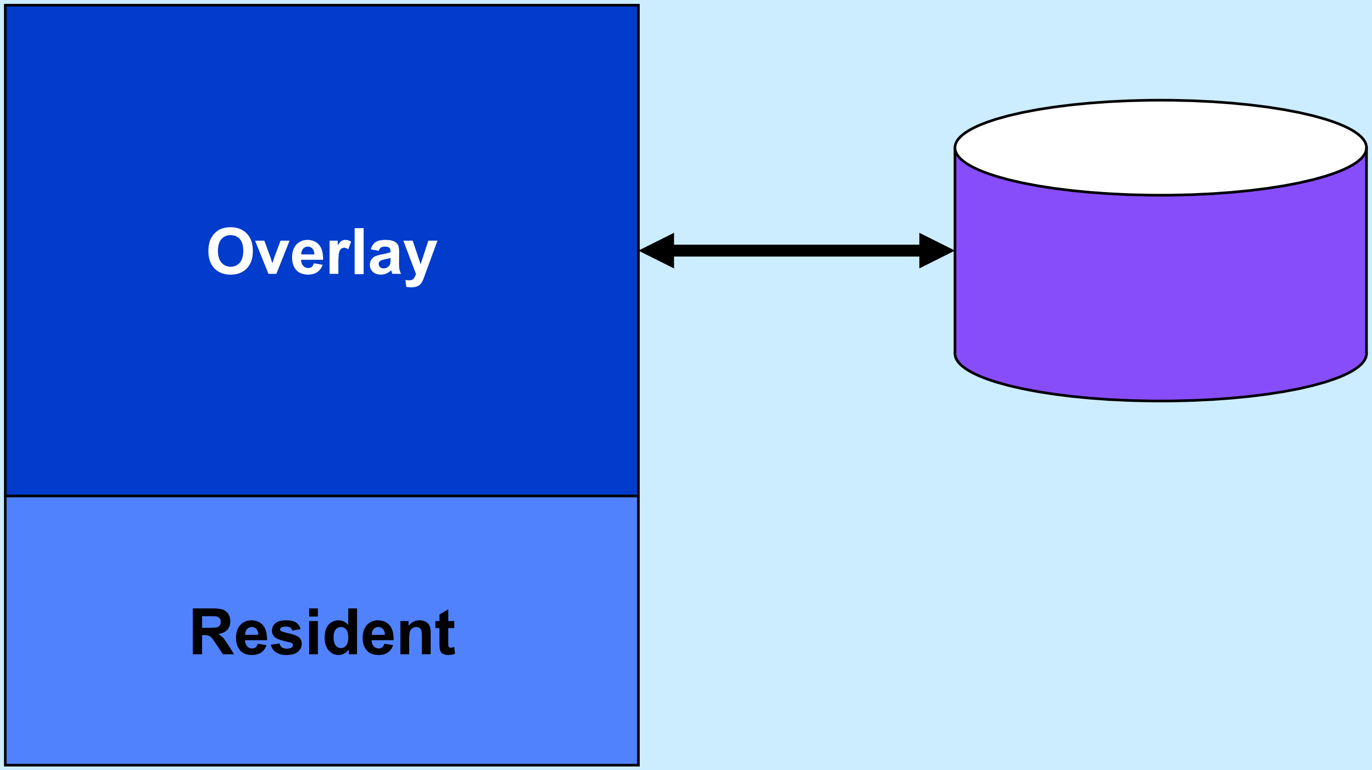
# Base and Bounds Registers



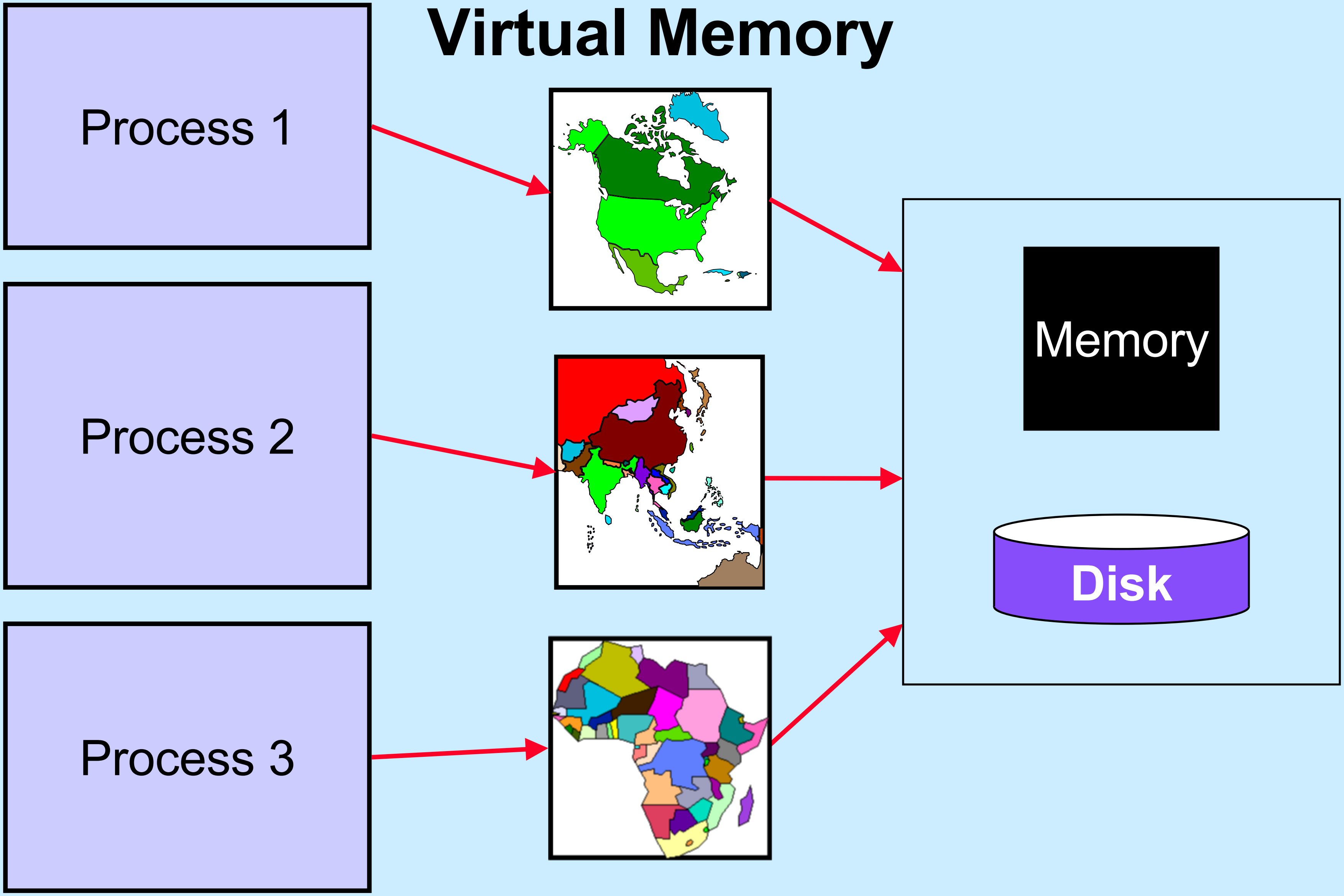
# Swapping



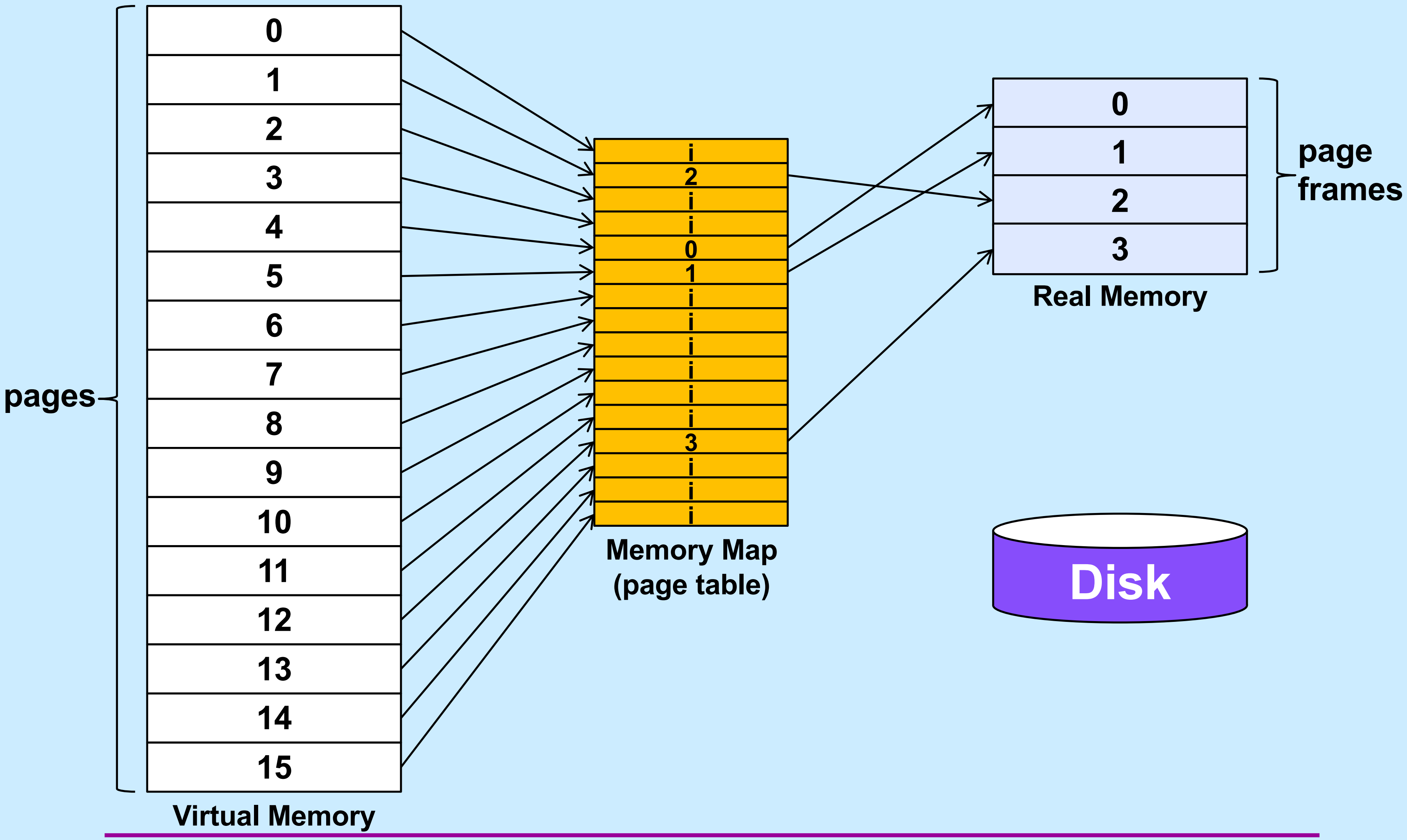
# Overlays



# Virtual Memory

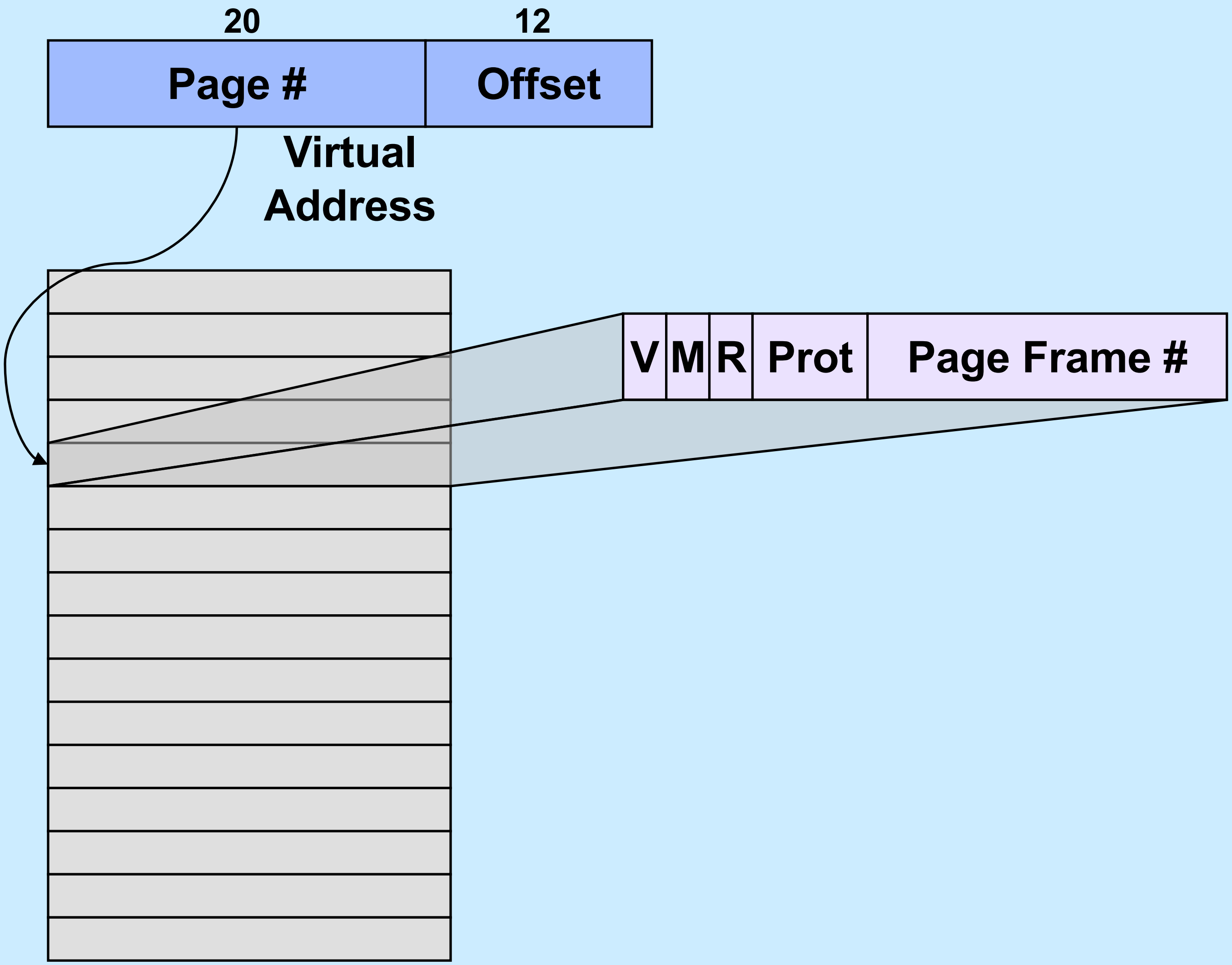


# Memory Maps



pages
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
Virtual Memory
i
2
i
i
0
1
i
i
i
i
i
i
3
i
i
i
Memory Map
(page table)
0
1
2
3
page
frames
Real Memory
Disk

# Page Tables


20
12
Page #
Offset
Virtual Address
V
M
R
Prot
Page Frame #