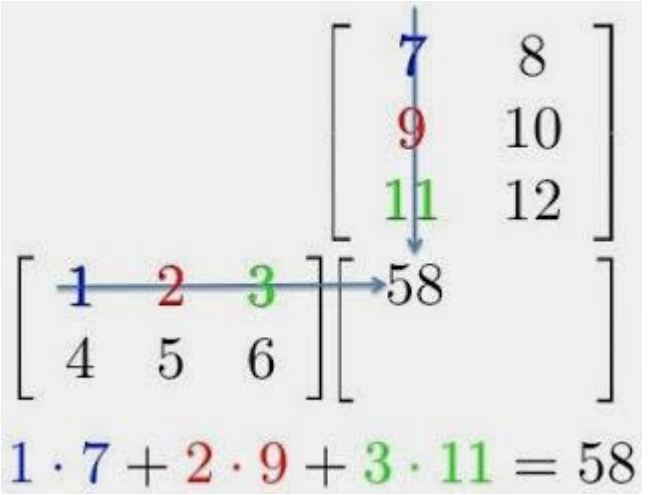
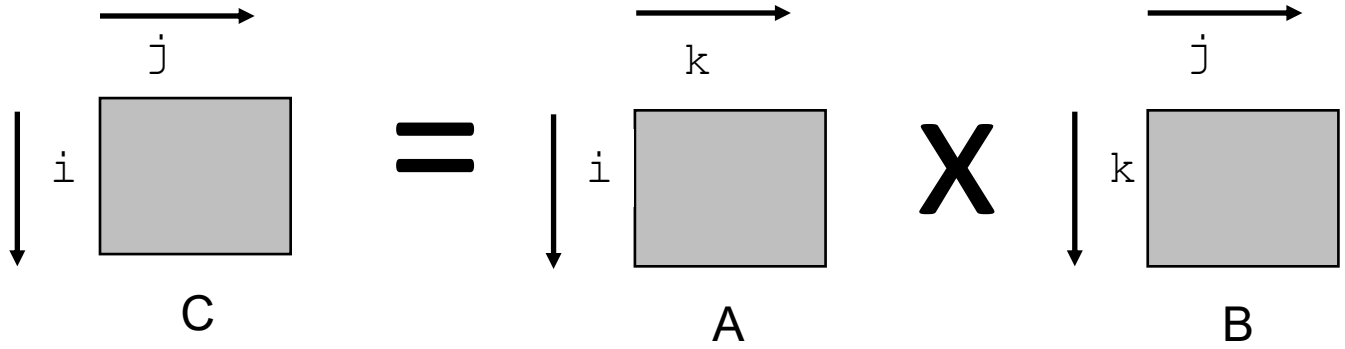
Cache Performance and Dynamic Memory

CMPU 224 – Computer Organization
Jason Waterman

Miss Rate Analysis for Matrix Multiply

- Assume:
  - Block size = 32B (big enough for **four** doubles)
  - Matrix dimension (N) is very large
  - Cache is not even big enough to hold multiple rows
- Analysis Method:
  - Look at access pattern of inner loop



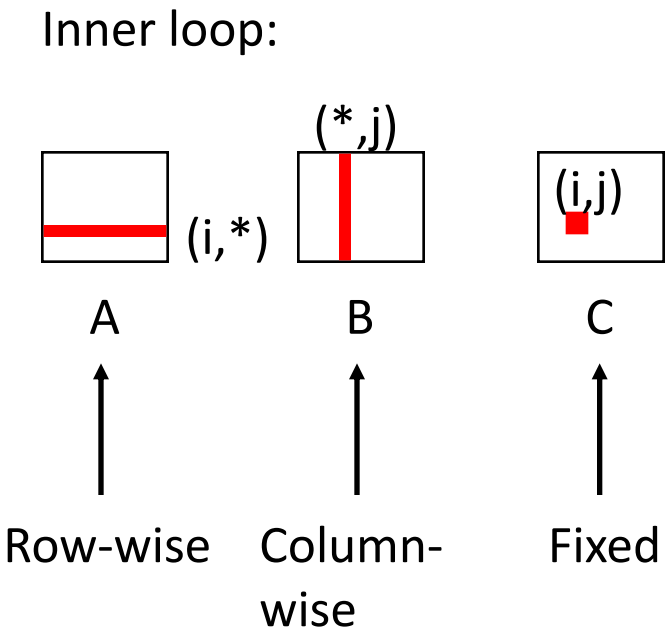


Layout of C Arrays in Memory

- C arrays allocated in row-major order
  - Each row in contiguous memory locations
- Stepping through columns in one row:
  - `for (i = 0; i < N; i++)`
    `sum += a[0][i];`
  - Accesses successive elements to exploit spatial locality
    - Miss rate = 8 (sizeof(double)) / 32 (block size) = 1/4 (25%)
- Stepping through rows in one column:
  - `for (i = 0; i < n; i++)`
    `sum += a[i][0];`
  - Accesses distant elements
  - No spatial locality!
    - Miss rate = 1 (100%)

Matrix Multiplication (ijk)

```
/* ijk */
for (i=0; i<n; i++)  {
  for (j=0; j<n; j++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

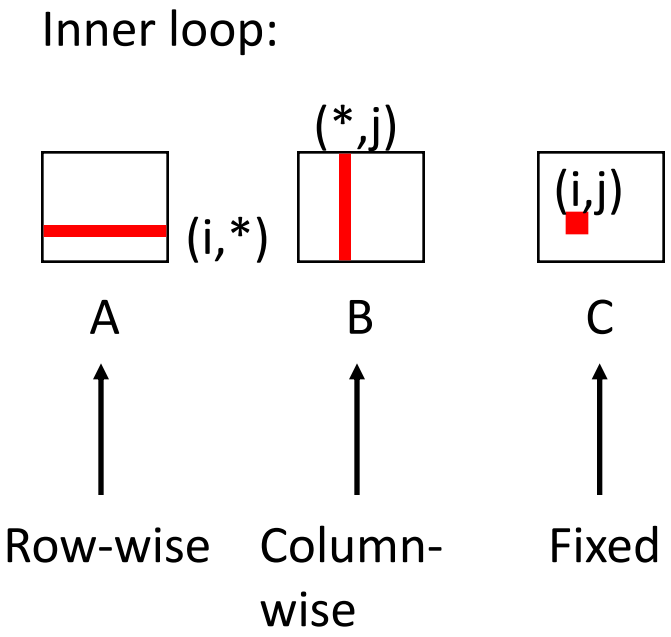


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (jik)

```
/* jik */
for (j=0; j<n; j++)  {
  for (i=0; i<n; i++) {
    sum = 0.0;
    for (k=0; k<n; k++)
      sum += a[i][k] * b[k][j];
    c[i][j] = sum;
  }
}
```

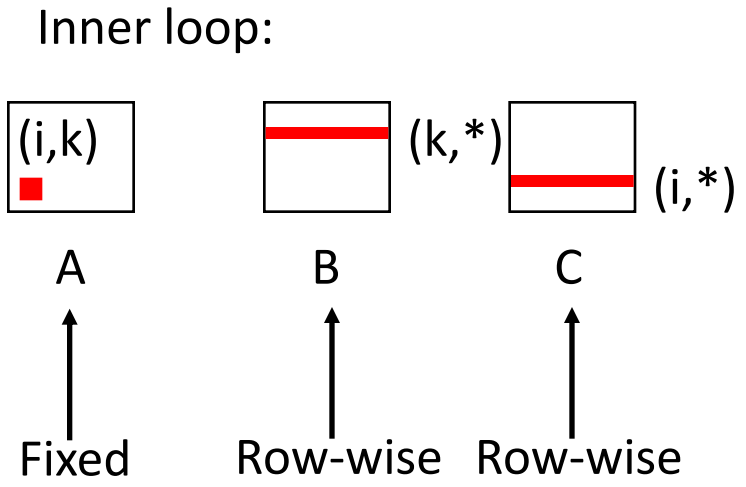


Misses per inner loop iteration:

A	B	C
0.25	1.0	0.0

Matrix Multiplication (kij)

```
/* kij */
for (k=0; k<n; k++) {
  for (i=0; i<n; i++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

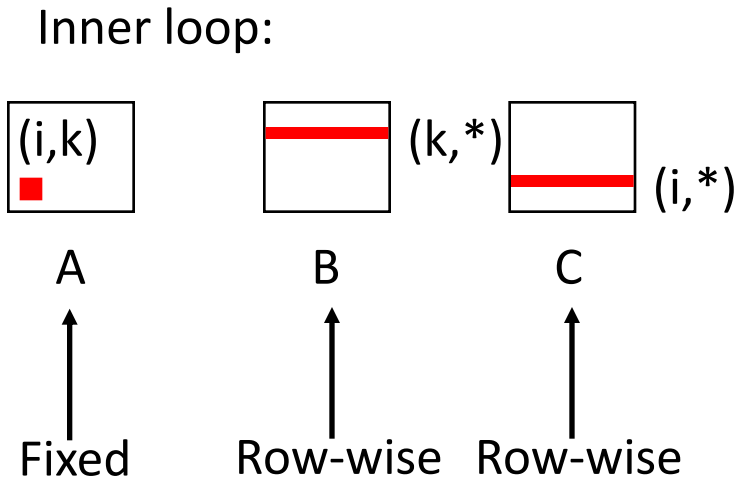


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (ikj)

```
/* ikj */
for (i=0; i<n; i++) {
  for (k=0; k<n; k++) {
    r = a[i][k];
    for (j=0; j<n; j++)
      c[i][j] += r * b[k][j];
  }
}
```

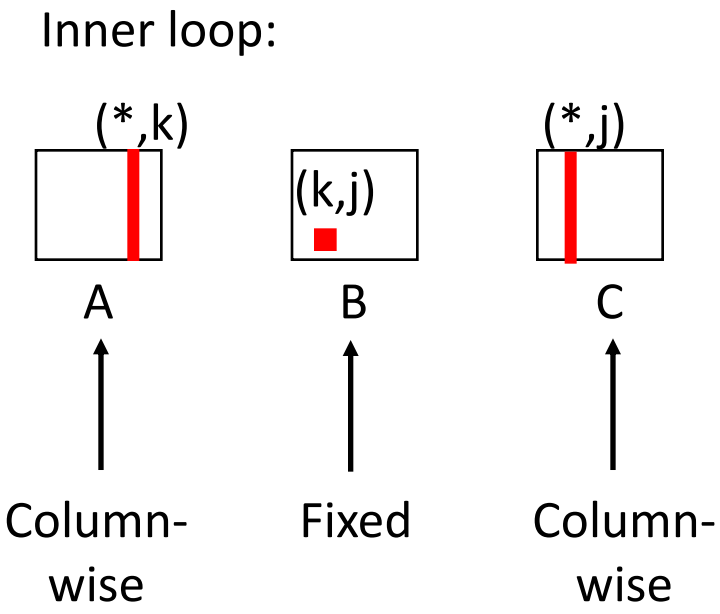


Misses per inner loop iteration:

A	B	C
0.0	0.25	0.25

Matrix Multiplication (jki)

```
/* jki */
for (j=0; j<n; j++) {
  for (k=0; k<n; k++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```

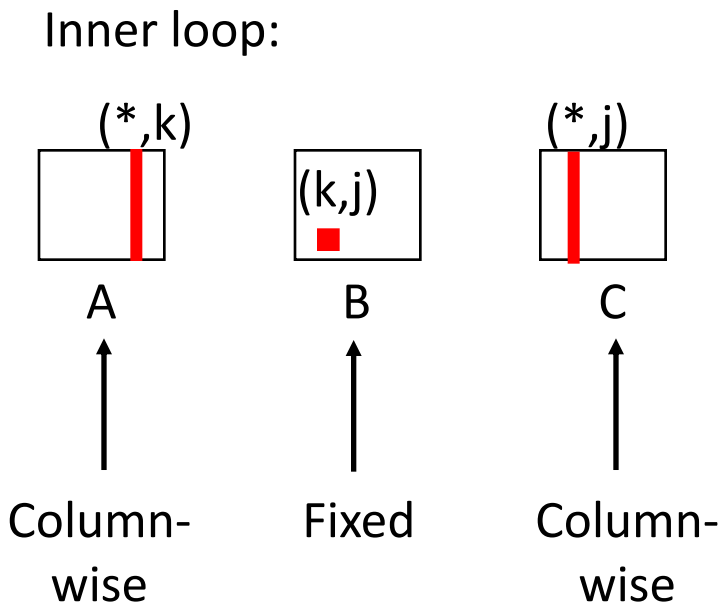


Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Matrix Multiplication (kji)

```
/* kji */
for (k=0; k<n; k++) {
  for (j=0; j<n; j++) {
    r = b[k][j];
    for (i=0; i<n; i++)
      c[i][j] += a[i][k] * r;
  }
}
```



Misses per inner loop iteration:

A	B	C
1.0	0.0	1.0

Summary of Matrix Multiplication

```
for (i=0; i<n; i++) {
  for (j=0; j<n; j++) {
   sum = 0.0;
   for (k=0; k<n; k++)
     sum += a[i][k] * b[k][j];
   c[i][j] = sum;
 }
}
```

```
for (k=0; k<n; k++) {
 for (i=0; i<n; i++) {
  r = a[i][k];
  for (j=0; j<n; j++)
   c[i][j] += r * b[k][j];
 }
}
```

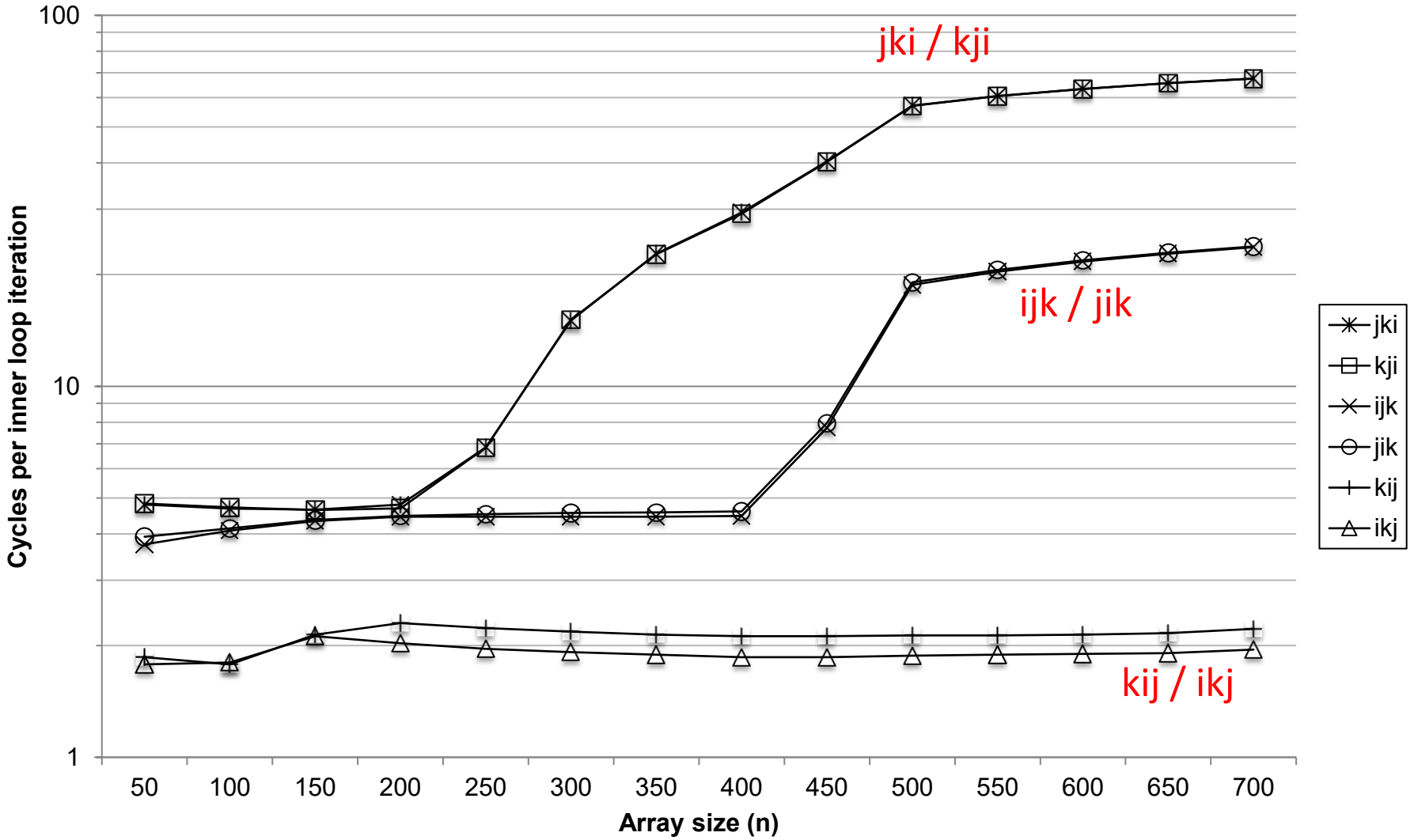
```
for (j=0; j<n; j++) {
 for (k=0; k<n; k++) {
   r = b[k][j];
   for (i=0; i<n; i++)
    c[i][j] += a[i][k] * r;
 }
}
```

ijk (& jik):
- 2 loads, 0 stores
- misses/iter = 1.25

kij (& ikj):
- 2 loads, 1 store
- misses/iter = 0.5

jki (& kji):
- 2 loads, 1 store
- misses/iter = 2.0

Core i7 Matrix Multiply Performance

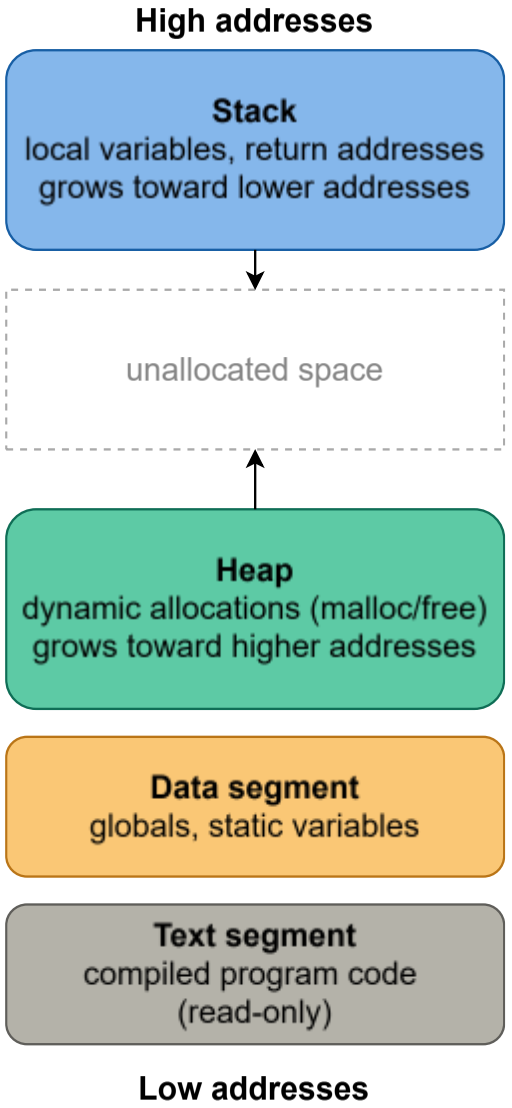


Dynamic Memory

- **Global / Static Variables**
  - Allocated at compile time in the **data segment**
  - Size and lifetime fixed for the entire program run
- **Local Variables — The Stack**
  - Allocated automatically when a function is called, freed when it returns
  - Fast: just moving the stack pointer
  - But: size must be known at compile time, and lifetime is tied to the call frame
- **We need memory that outlives the function that created it**
  - Where the size of memory isn’t known at compile time
  - Mechanism for allocating memory at **run-time**

The Heap

- **A region of memory for runtime-managed allocation**
  - Sits between the data segment and the stack
  - Grows upward (toward higher addresses) as allocations are made
  - Not automatically managed — your program is responsible for it
- **Key properties**
  - Memory allocated on the heap **persists** until explicitly released
  - Any part of the program with the pointer can access it
  - Lifetime is completely independent of any function call or scope
- **The tradeoff**
  - Stack allocation is just a pointer increment — essentially free
  - Heap allocation requires bookkeeping: finding free blocks, tracking sizes
  - With manual control comes manual responsibility — the runtime won't clean up for you



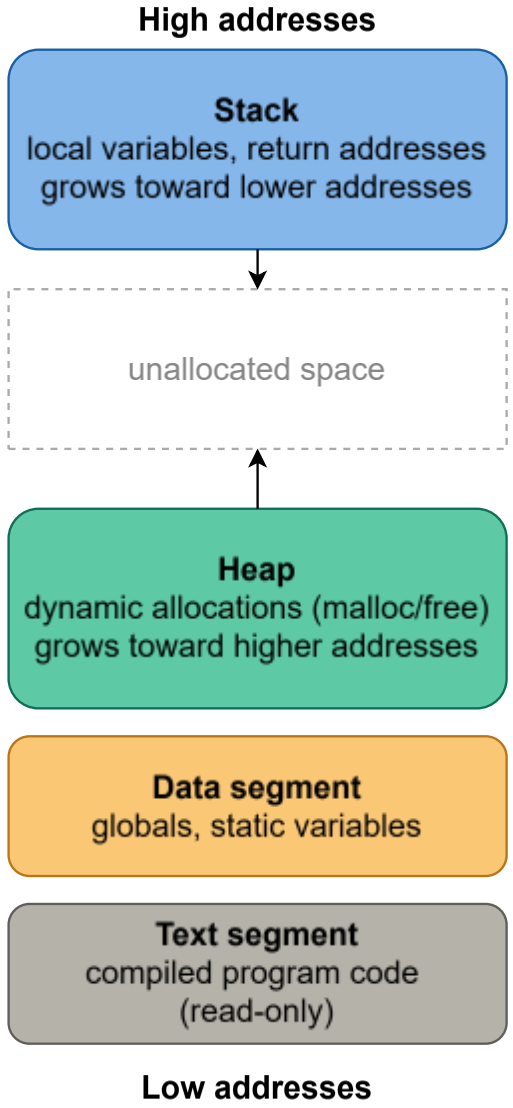
Memory Addresses

- **Where does code, stack and dynamic memory live?**

```
#include <stdio.h>
#include <stdlib.h>

int x = 1; // global

int main(void) {
  int y = 2; // allocated in the stack frame of main()

  printf("location of code:    %p\n", (void *)main);
  printf("location of globals: %p\n", (void *)&x);
  printf("location of heap:    %p\n", (void *)malloc(1));
  printf("location of stack:   %p\n", (void *)&y);

  return x + y; // so compilier doesn't optimize x and y away
}
```

A simple program that prints out addresses

```
$ ./print_address
location of code:    0x101b0
location of globals: 0x22834
location of heap:    0x23410
location of stack:   0x2b2aaf9c
```



Memory API: `malloc()`

```
#include <stdlib.h>

void* malloc(size_t size)
```

- Allocate a memory region on the heap
  - Argument
    - `size_t size` : size of the memory block (in bytes)
    - `size_t` is an unsigned integer type capable of holding the size of any valid memory request
  - Return
    - Success: a void type pointer to the memory block allocated by `malloc`
    - Fail: a `null` pointer

Memory API: `sizeof()`

- Good coding style is to use `sizeof` in `malloc` instead of requesting the number of bytes directly
- Two types of results using `sizeof` with variables
  - The actual size of `'x'` is known at compile-time

```
int x[10];
printf("%d\n", sizeof(x));
```

```
40
```

  - The actual size of `'x'` is known at run-time

```
int *x = malloc(10 * sizeof(int));
printf("%d\n", sizeof(x));
```

```
4
```

Memory API: `free()`

```
#include <stdlib.h>

void free(void* ptr)
```

- Free a memory region allocated by a call to `malloc`
  - Argument
    - `void *ptr` : a pointer to a memory block allocated with `malloc`
  - Returns
    - Nothing

Forgetting To Allocate Memory

- Incorrect code

```
char *src = "hello"; //character string constant
char *dst;           //unallocated
size_t BUFSIZE = strlen(src)+1;
strncpy(dst, src, BUFSIZE);    //segfault and die
```

- Correct code

```
char *src = "hello";      //character string constant
char *dst;
size_t BUFSIZE = strlen(src)+1;
dst = (char *)malloc(BUFSIZE); // allocated
strncpy(dst, src, BUFSIZE);    //works properly
```

Not Allocating Enough Memory

- Incorrect code, but “works” properly

```
char *src = “hello”;      //character string constant
char *dst;
size_t BUFSIZE = strlen(src);
dst = (char *)malloc(BUFSIZE); // allocated
strncpy(dst, src, BUFSIZE);    // may seem to work, but dangerous
```

Other Errors and Issues

- Uninitialized read

```
int *x = (int *)malloc(sizeof(int)); // allocated
printf("*x = %d\n", *x); // uninitialized memory access
```

- Memory Leak
  - Forgetting to free memory after you are done with it
  - If this pattern repeats, system may slowly run out of memory
- Double Free
  - Freeing memory that was freed already

```
int *x = (int *)malloc(sizeof(int)); // allocated
free(x); // free memory once, OK
free(x); // free repeatedly, bad!  Behavior undefined
```

Other Memory APIs: `calloc()`

```
#include <stdlib.h>

void *calloc(size_t nmemb, size_t size)
```

- Allocate memory on the heap and zeroes it before returning
  - Arguments
    - `nmemb`: number of elements to allocate
    - `size`: size of one element
  - Returns
    - Success: a void type pointer to the memory block allocated by `calloc`
    - Fail: a null pointer

Other Memory APIs: realloc()

```
#include <stdlib.h>

void *realloc(void *ptr, size_t size)
```

- Change the size of memory block
  - A pointer returned by `realloc` may be either the same as `ptr` or a new
  - Argument
    - `void *ptr`: Pointer to memory block allocated with `malloc`, `calloc`, or `realloc`
    - `size_t size`: New size for the memory block(in bytes)
  - Return
    - Success: Void type pointer to the memory block
    - Fail: Null pointer

System Calls

- `malloc` is just a library call
  - It uses the `brk` system call to request more memory
  - `brk` expands (or shrinks) the program's *break*
    - *break*: The address of **the end of the heap** in address space
  - `sbrk` increases the break by increment
    - Can be a negative increment to reduce the size of the heap
  - Programmers **should never directly call** either `brk` or `sbrk`
- **Start** with **a small-sized heap** and then **request more** memory from the OS when needed

```
#include <unistd.h>

int brk(void *addr)
void *sbrk(intptr_t increment);
```

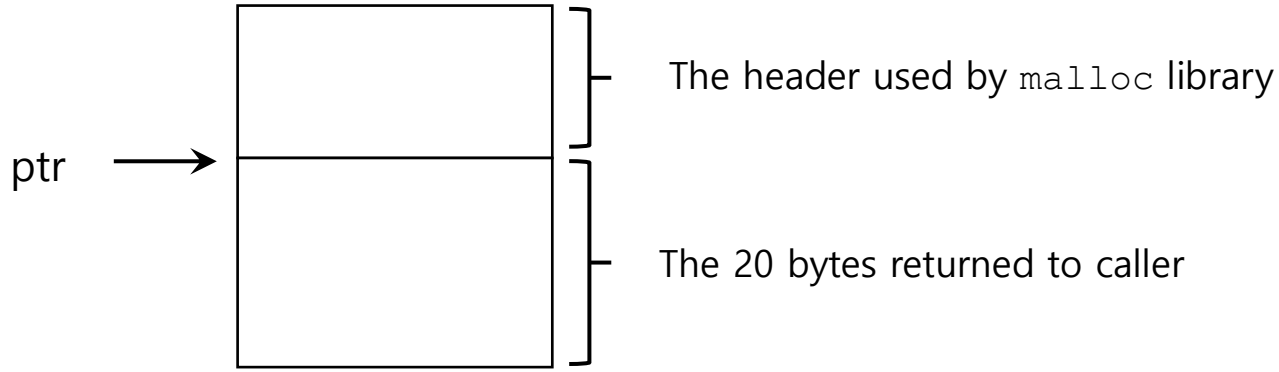
Free space management

- How should free space be managed?
  - Unpredictable, variable-sized requests
- Try to minimize fragmentation
  - **External** – total amount free memory available, but it's scattered in small, disconnected pieces, leaving no *single continuous block* large enough
  - **Internal** – wasted space *inside* an allocated memory block, when allocated block is larger than the specific size you requested.
- Time and space requirements for allocation algorithms

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

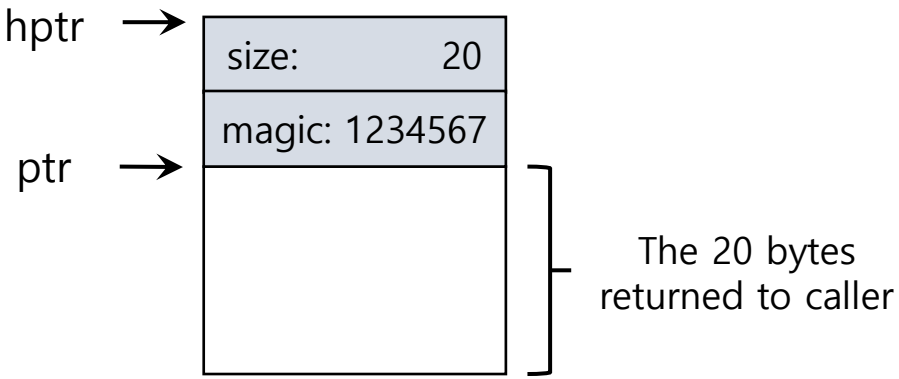
```
ptr = malloc(20);
```



An Allocated Region Plus Header

The Header of Allocated Memory Chunk

- 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



Specific Contents Of The Header

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

A Simple Header

The Header of Allocated Memory Chunk(Cont.)

- 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 = (char *)ptr - sizeof(header_t);
}
```

Embedding a Free List

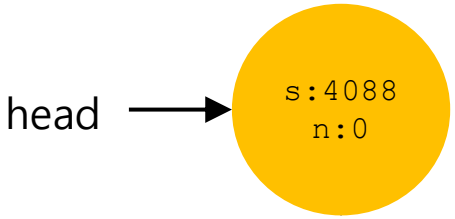
- The memory-allocation library **initializes** the heap and **puts** the first element of **the free list** in the **free space**
  - The library **can't use** `malloc()` to build a list **within itself**
- Description of a node in the list

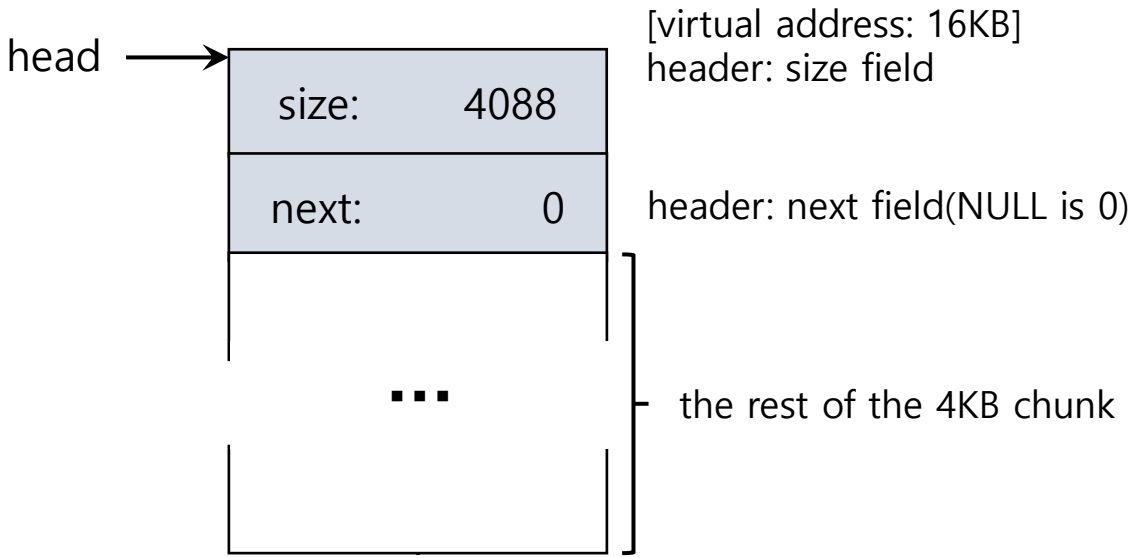
```
typedef struct __node_t {
        int size;
        struct __node_t *next;
} node_t;
```

A Heap With One Free Chunk

- Building the heap and creating a free list
  - For this example, the heap is built with the `mmap()` system call
  - `malloc()` would use `brk()`

```
// 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;
```





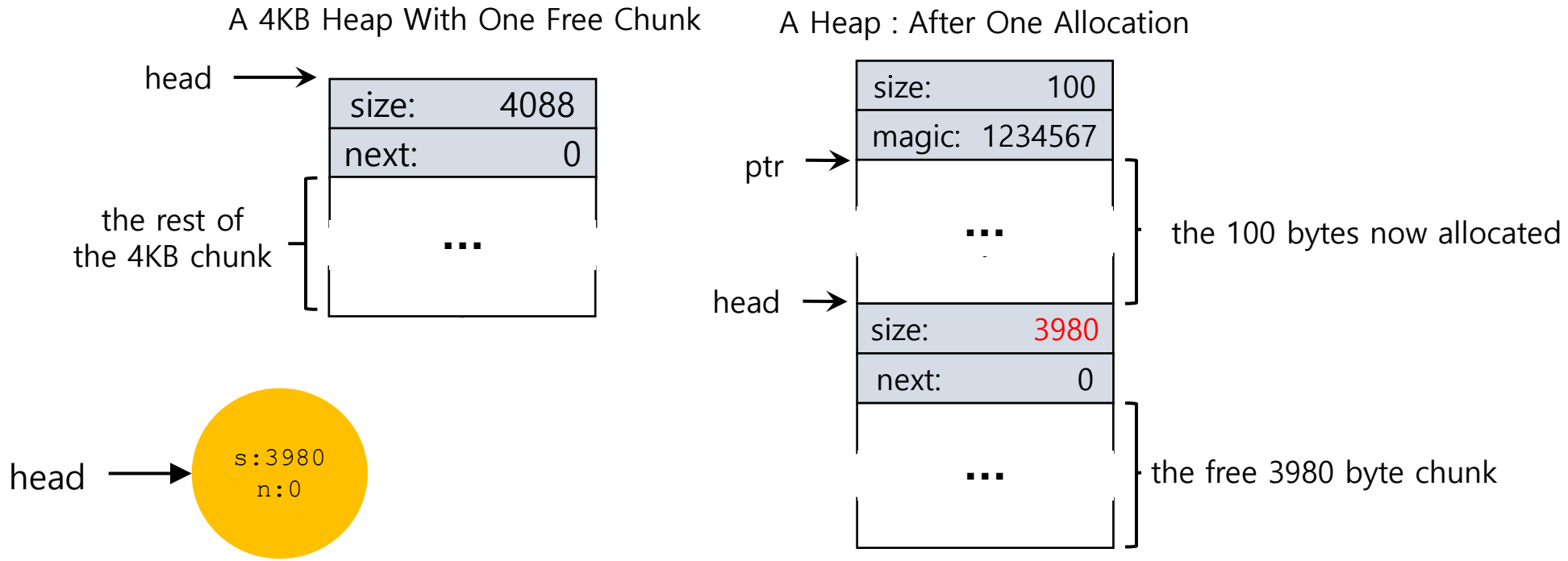
Embedding A Free List: Allocation

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

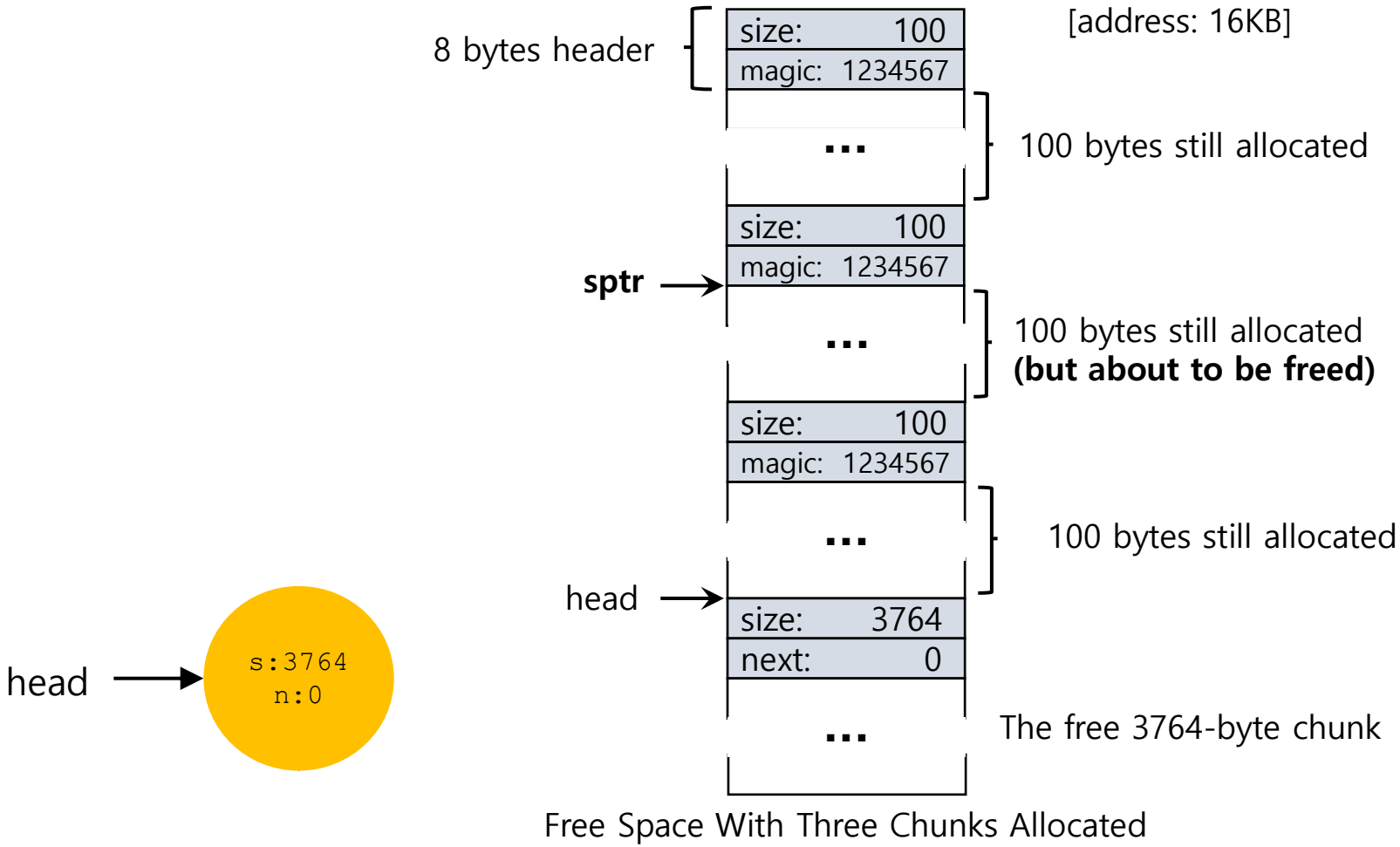
- The library will
  - **Split** the large free chunk into two
    - **One** for the **request** and the **remaining** free chunk
  - **Shrink** the size of free chunk in the list

Embedding A Free List: Allocation and Splitting

- Example: a request for 100 bytes by `ptr = malloc(100)`
  - **Splitting** the free block
  - Allocate 108 bytes out of the existing free chunk
  - Shrink the free chunk to 3980 (4088 minus 108)

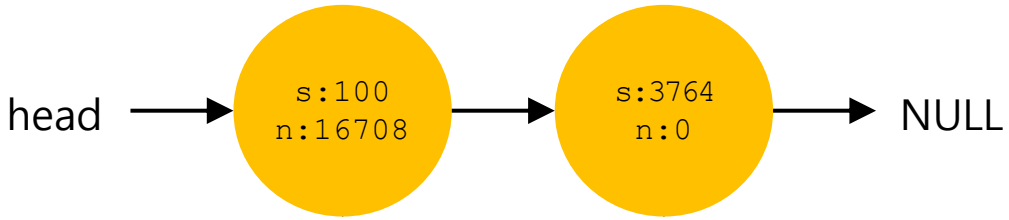


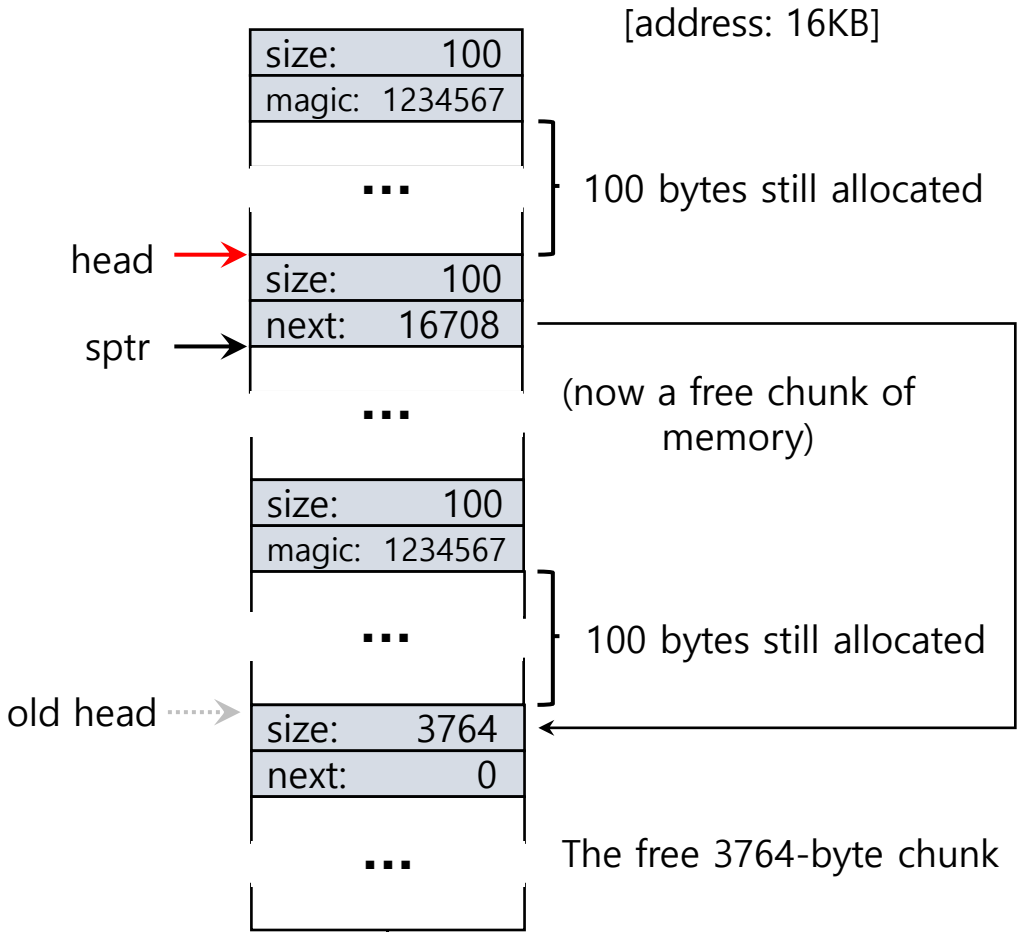
Freeing a pointer



Free Space With `free()`

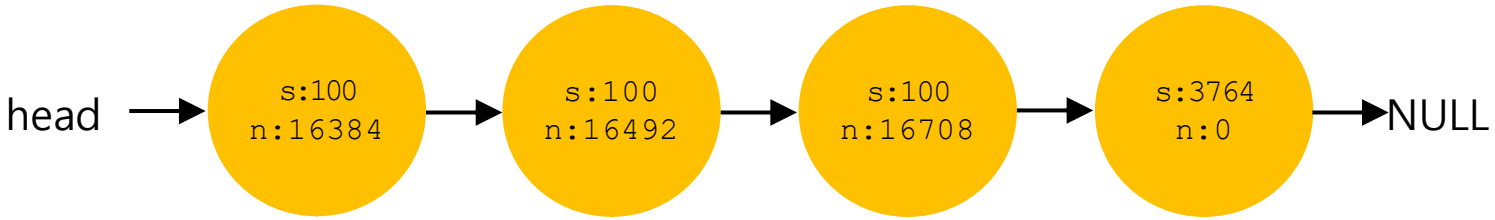
- Example: `free(sptr)`
  - The 100 byte chunk is **put back into** the free list
  - sptr->next == old head
  - It is now the head of the free list
  - The free list will **start** with **the small chunk**

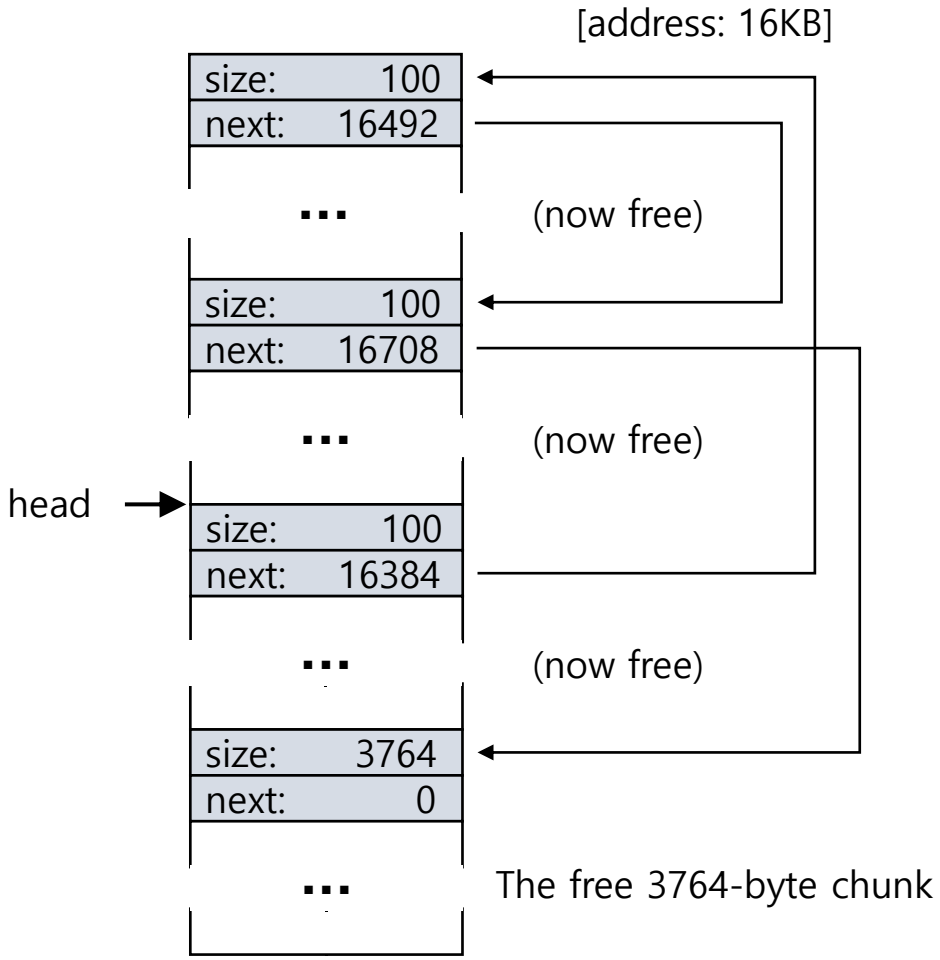




Free Space With Freed Chunks

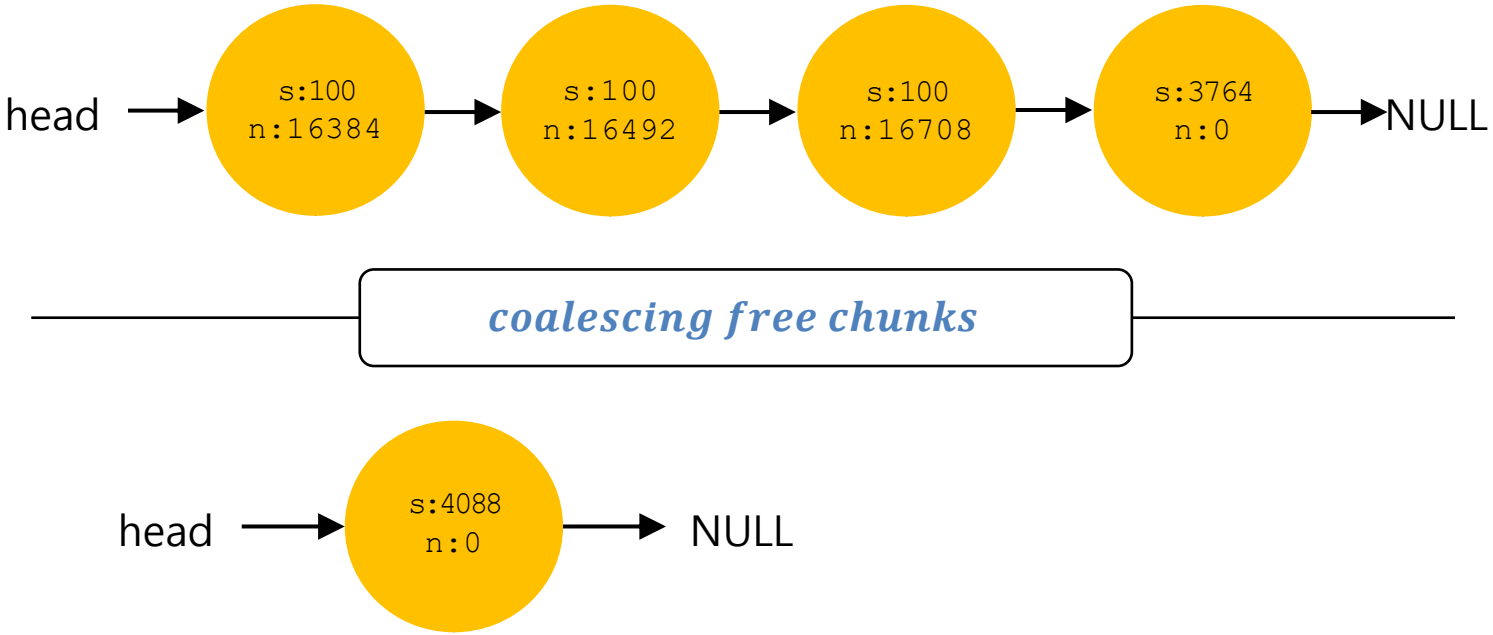
- Let's assume that the last two in-use chunks are freed
- **External Fragmentation** occurs
  - **Coalescing** is needed in the list



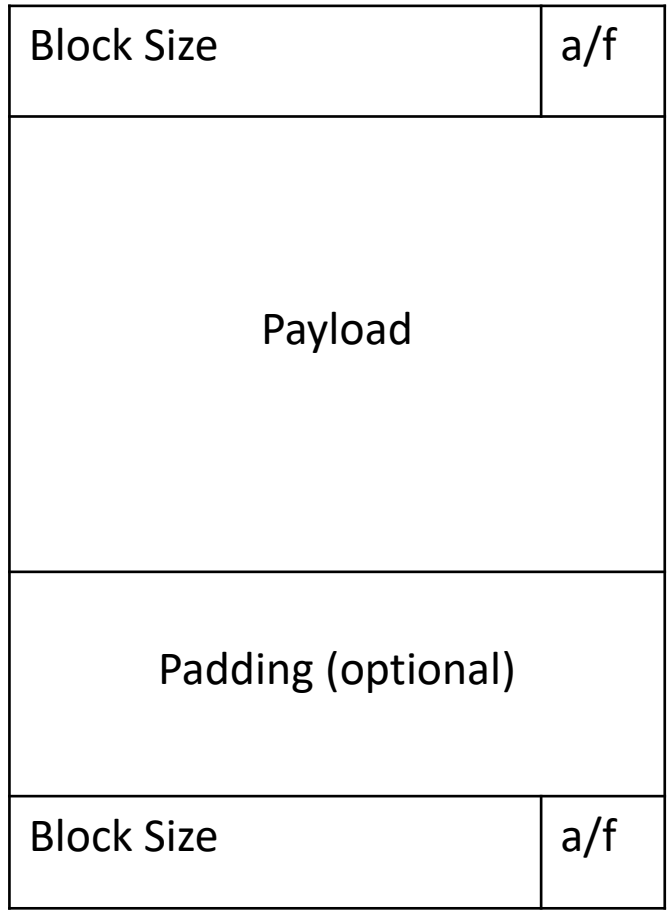


Coalescing

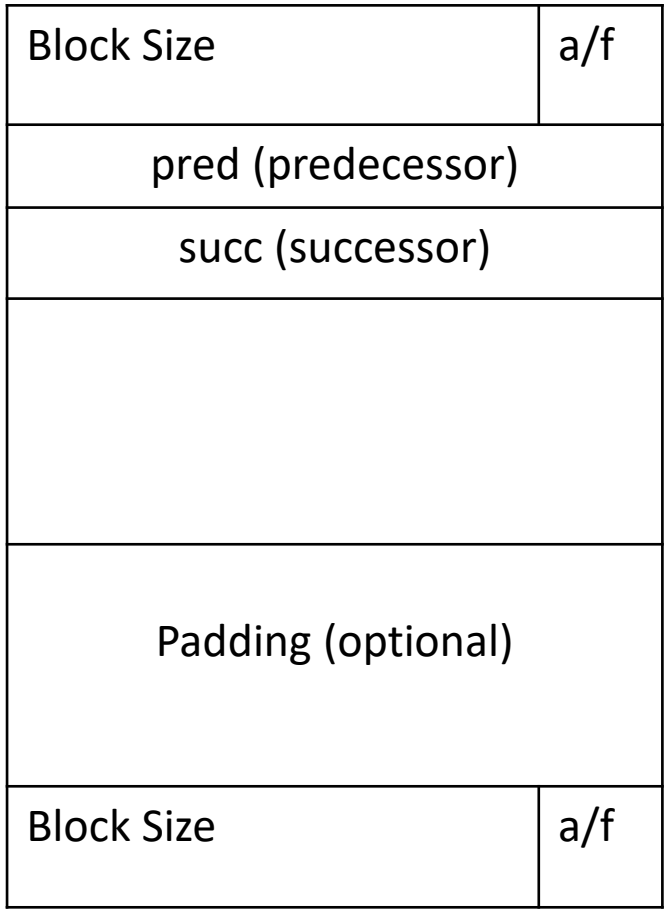
- If a user requests memory that is **bigger than the free chunk size**, the list will **not find** such a free chunk
- Coalescing: **Merge** returning a free chunk with existing chunks into a large single free chunk if **addresses** of them are **nearby**



Constant time Coalescing with an Explicit Free List



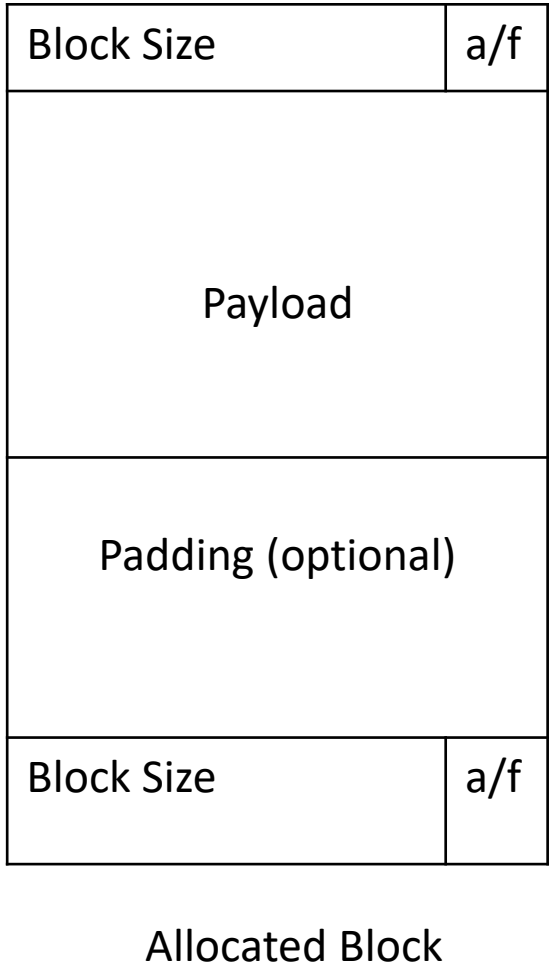
Allocated Block



Free Block

Constant time Coalescing with an Explicit Free List

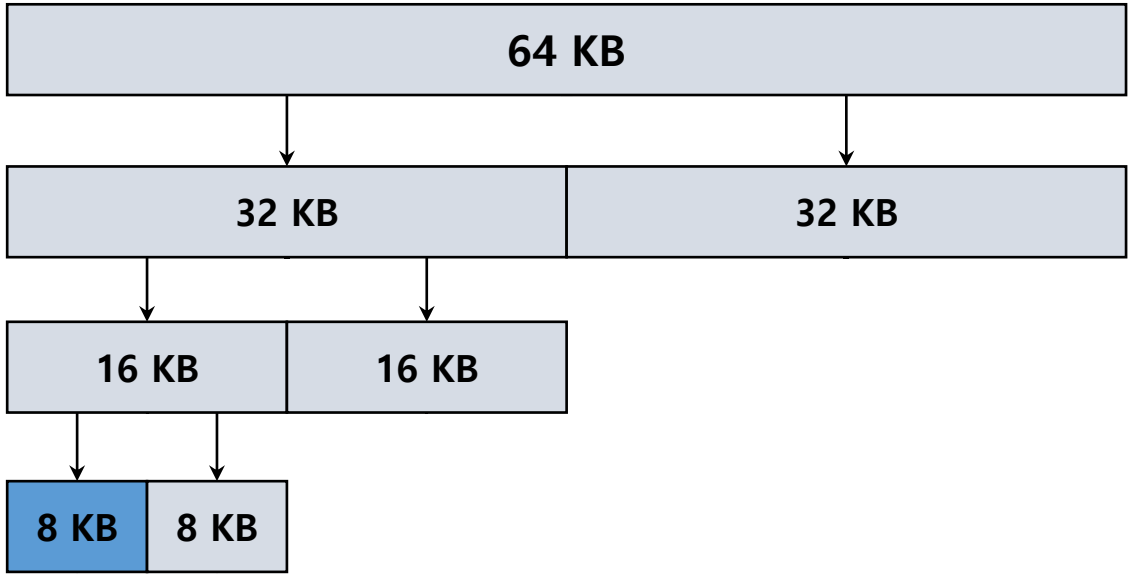
- Case 1: A (Alloc) | **B (Freed)** | C (Alloc)
  - No coalescing; block B is simply added to the front of the explicit free list
- Case 2: A (Alloc) | **B (Freed)** | **C (Free)**
  - Coalesce B and C; since C is already free, it *must* be in the explicit list
  - Block C is **removed** from the explicit list
  - The new, larger block (B+C) is added to the front of the list
- Case 3: **A (Free)** | **B (Freed)** | C (Alloc)
  - Coalesce A and B; block A is *already* in the explicit list
  - No list pointers need to change. Block A's header and new footer (at the end of B) are simply updated to reflect the new, larger size (A+B)
- Case 4: **A (Free)** | **B (Freed)** | **C (Free)**
  - Coalesce all three; both A and C are in the explicit list
  - Block C is **removed** from the list
  - Block A's header and new footer (at the end of C) are updated to reflect the new size (A+B+C)



Allocated Block

Other Approaches

- Segregated Free List
  - Have multiple lists based on size
  - Free chunk goes in the appropriate list
- Binary Buddy Allocation
  - The allocator **divides free space** by two **until a block** that is big enough to accommodate the request is **found**
  - Can suffer from **internal fragmentation**
  - Buddy system makes **coalescing** simple



64KB free space for 7KB request