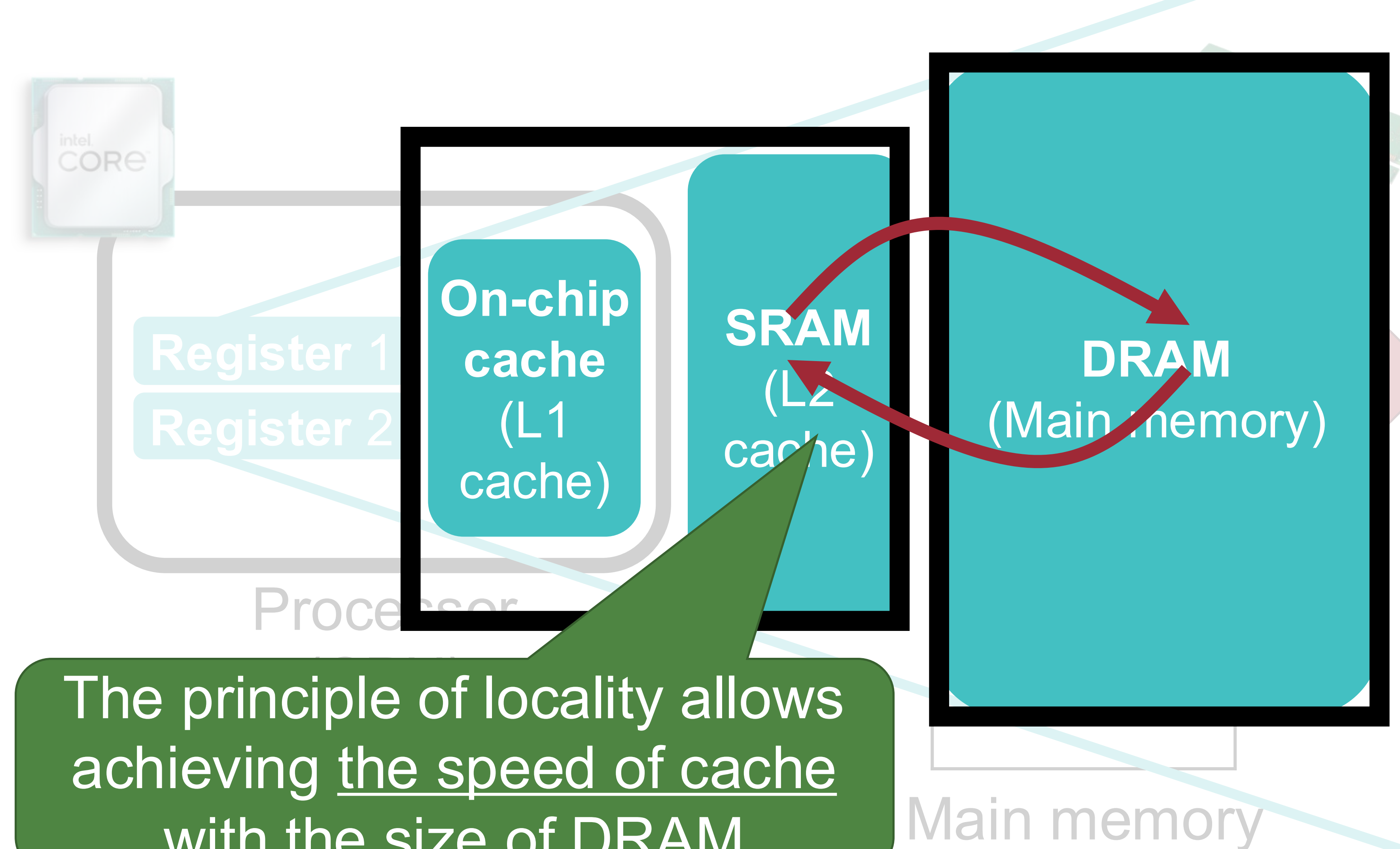
CSE251: System Programming

17. Dynamic Memory Allocation (1)

Seongil Wi

Department of Computer Science and Engineering

Where Are We?: Cache



Where Are We?: Virtual Memory

We want to give
the speed of DRAM with
the size of hard disk!

Register 1
Register 2
On-chip cache (L1 cache)
Processor (CPU) (e.g., Intel I7)
SRAM (L2 cache)

DRAM
(Main memory)

Secondary storage
(Hard disk)

Main memory

Today's Topic: Dynamic Memory Allocation

- Programmers use dynamic memory allocators (e.g., `malloc`) to acquire virtual memory **at runtime**

Why do we need this?

Problems of Statically Declared Variables


```c
int main(void) {
  int scores[100];
  int n;

  printf("Number of students: ");
  scanf("%d", &n);


  for (int i = 0; i < n; i++) {
    scanf("%d", &scores[i]);
  }

  return 0;
}
```

The array size is fixed before runtime

- If n > 100, the program may access memory out of bounds
- If n < 100, unused space is wasted

n is known only at runtime

Motivation of Dynamic Memory Allocation


```c
int main(void) {
  int scores[100];
  int n;

  printf("Number of students: ");
  scanf("%d", &n);


  for (int i = 0; i < n; i++) {
    scanf("%d", &scores[i]);
  }

  return 0;
}
```

```c
int main(void) {

  int n;

  printf("Number of students: ");
  scanf("%d", &n);

  int *scores = malloc(n*sizeof(int));
  for (int i = 0; i < n; i++) {
    scanf("%d", &scores[i]);
  }

  return 0;
}
```

Motivation of Dynamic Memory Allocation


```
int main(void) {
  int s
  int n

  printf("Number of student
  scanf("%d", &n);


  for (int i = 0; i < n; i++) {
    scanf("%d", &scores[i]);
  }
```

malloc allocates exactly enough space for n integers

```
int main(void) {

  int n;

  printf("Number of students: ");
  scanf("%d", &n);

  int *scores = malloc(n*sizeof(int));
  for (int i = 0; i < n; i++) {
    scanf("%d", &scores[i]);
  }
```

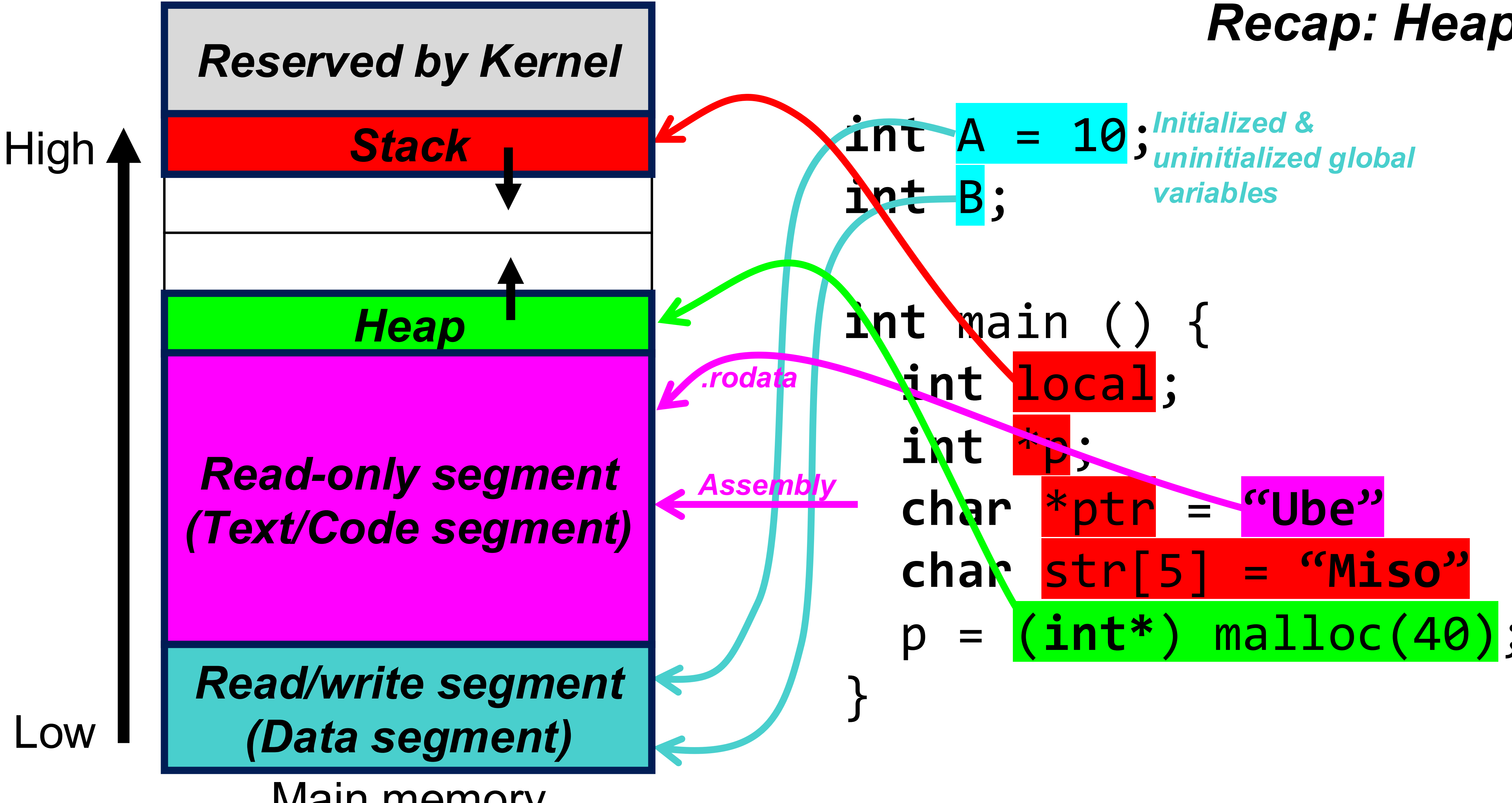
Dynamic memory allocation is needed when the required memory size is ***known only at runtime***

Dynamic Memory Allocation

Dynamic Memory Allocation

- Programmers use dynamic memory allocators (e.g., `malloc`) to acquire virtual memory **at runtime**
  - For data structures whose size is only known at runtime (It is very common to receive runtime data from user input via `stdin`, files, or the network)

- Dynamic memory allocators manage an area of process virtual memory known as the ***heap***



Recap: Heap
Reserved by Kernel
Stack
Heap
Read-only segment
(Text/Code segment)
Read/write segment
(Data segment)
High
Low
Main memory
.rodata
Assembly
int A = 10;
int B;
Initialized & uninitialized global variables
int main () {
int local;
int *p;
char *ptr = “Ube”
char str[5] = “Miso”
p = (int*) malloc(40);
}

Heap

- Heap is a memory area where objects are dynamically allocated and freed

High

Low

Stack

Heap

Read-only segment (Text/Code segment)

Read/write segment (Data segment)

Main memory

`rsp` register: point to the top of the stack

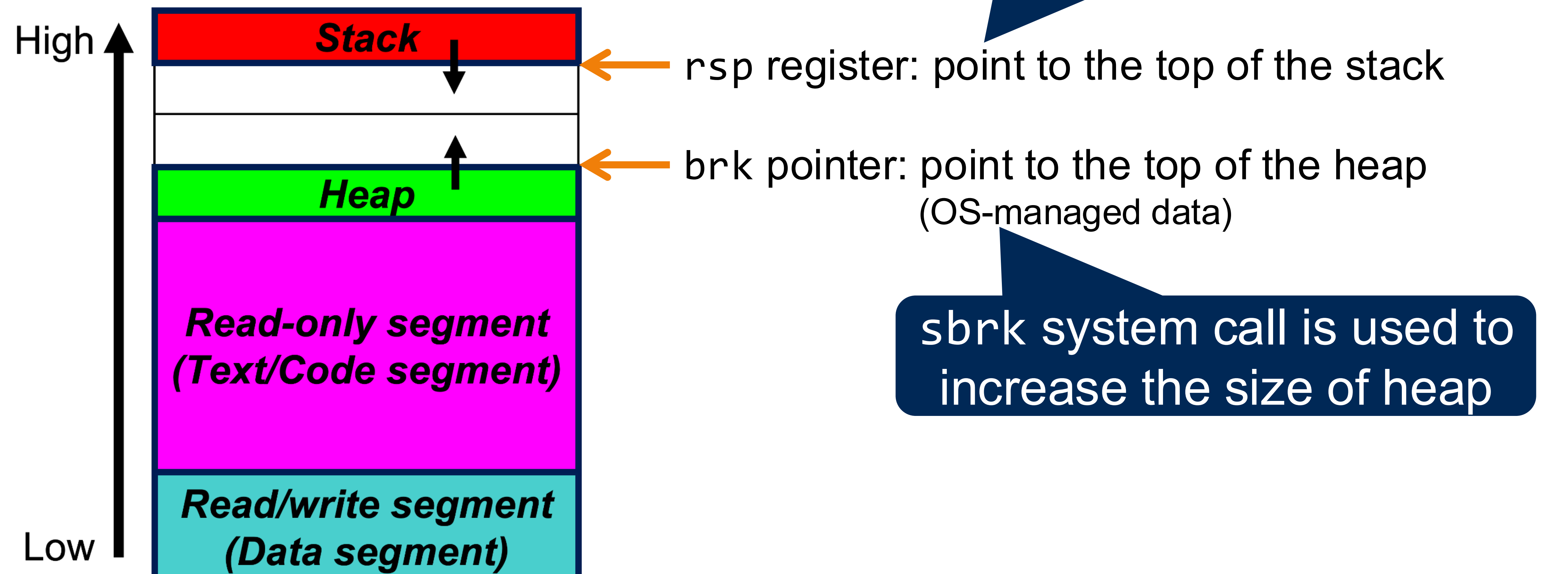
`brk` pointer: point to the top of the heap

(OS-managed data)

Heap

- Heap is a memory area where ob ted and freed

push instruction is used to increase the size of stack



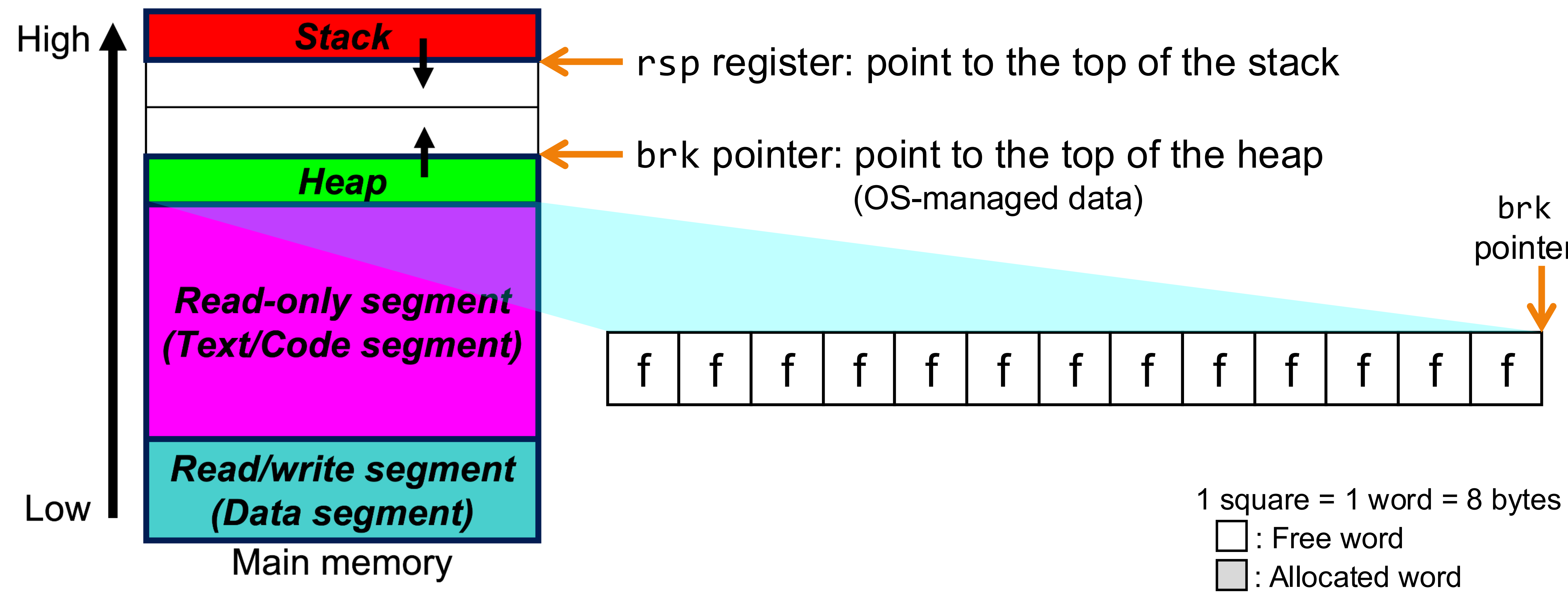
Main memory

Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***

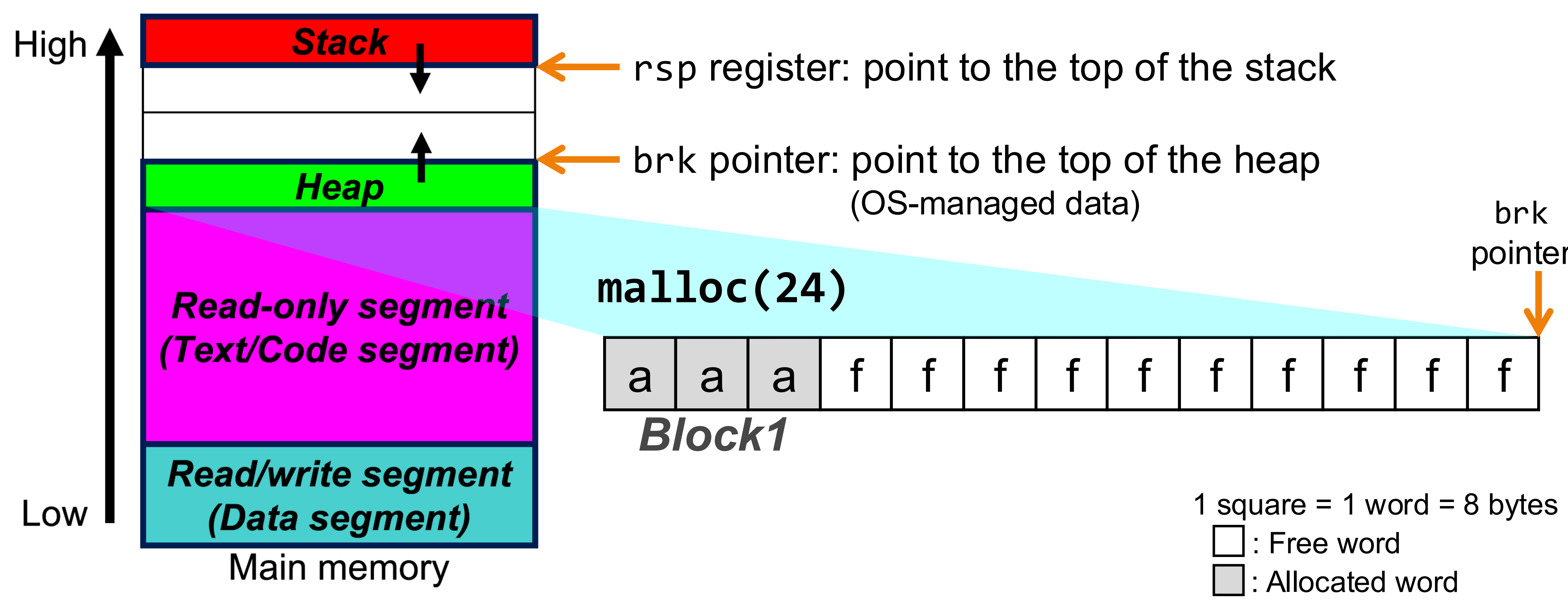
Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***



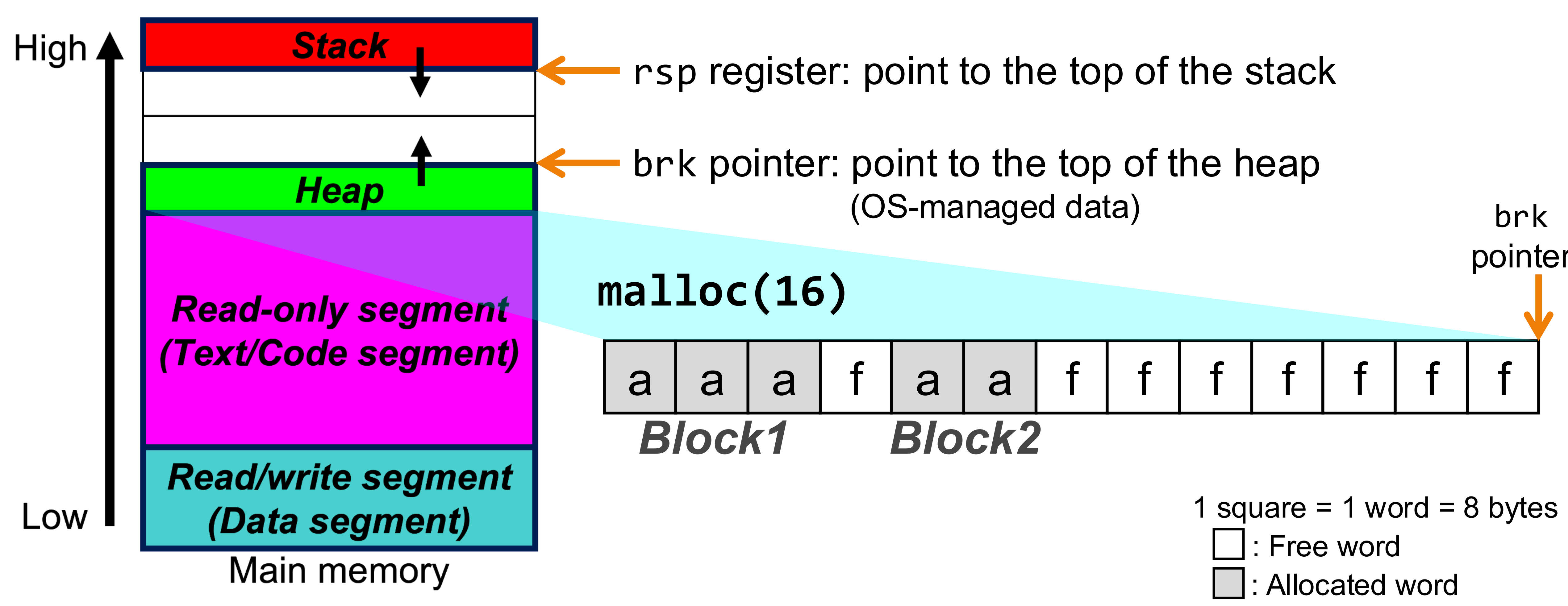
Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***



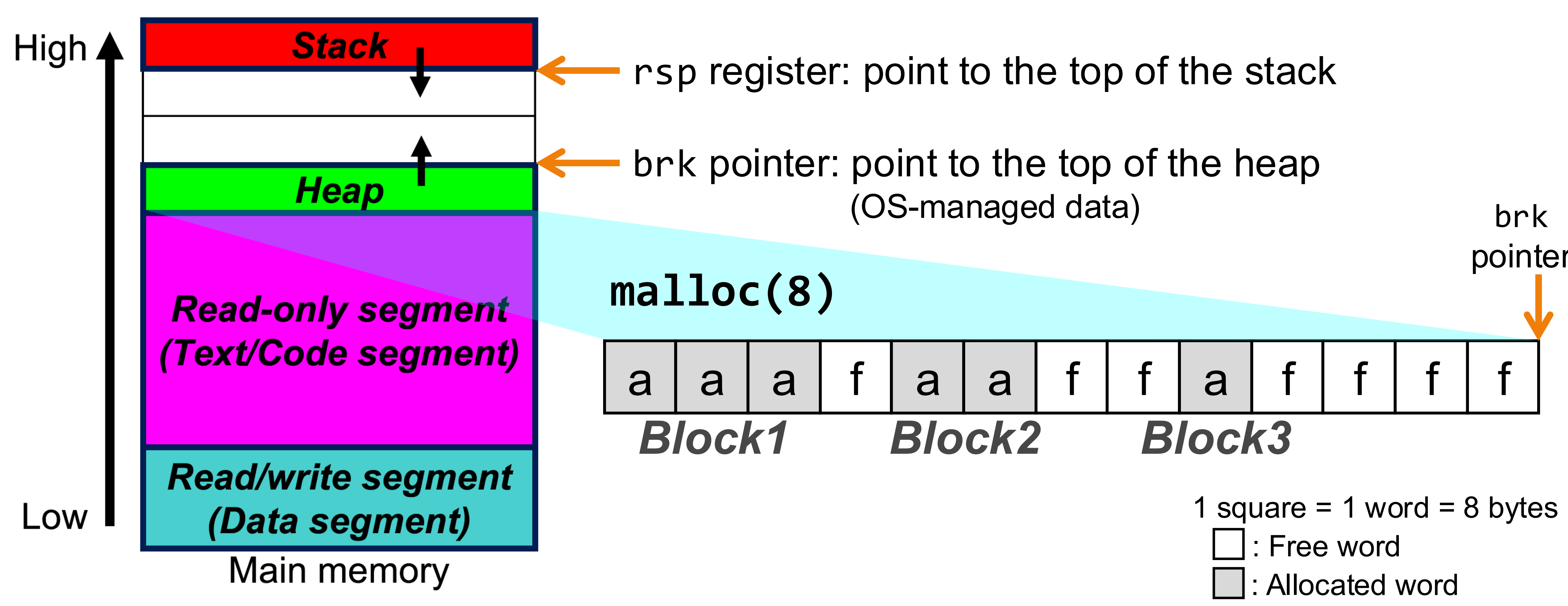
Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***



Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***



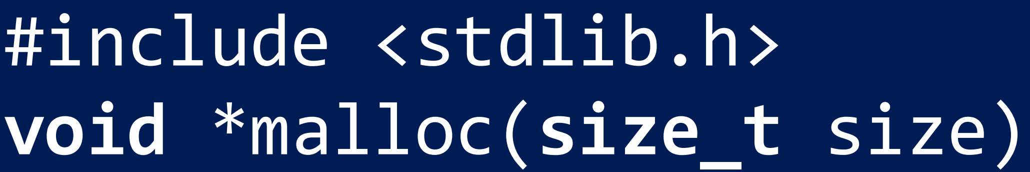
Heap Allocator (a.k.a, Dynamic Memory Allocator, Heap Manager)

- Maintains heap as collection of variable sized ***blocks***, which are either ***allocated*** or ***free***

- Types of allocators:
  - **Explicit allocator**: program allocates and frees space
    - e.g., malloc and free in C
  - **Implicit allocator**: program allocates, but does not free space
    - e.g., new and garbage collection in Java

- Will discuss simple explicit memory allocation today

The malloc Package

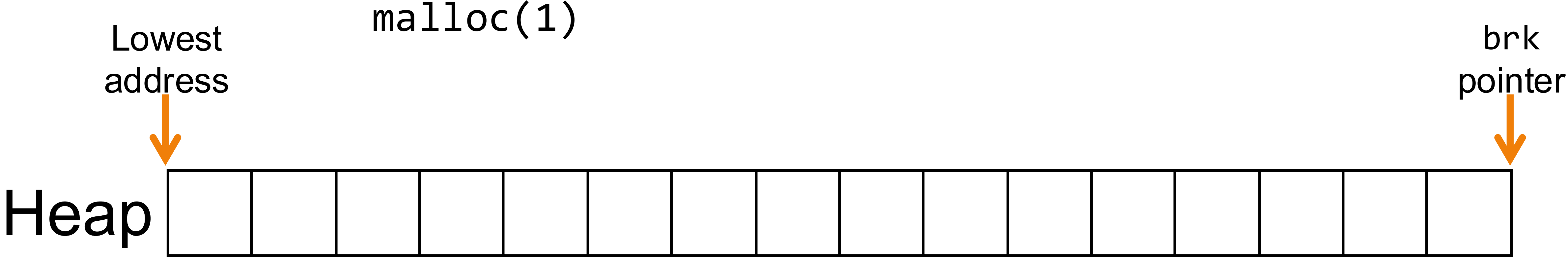


- Successful:
  - Returns a pointer to a memory block of at least size bytes aligned to a 16-byte boundary (on x86-64)

The malloc Package

```
#include <stdlib.h>
void *malloc(size_t size)
```

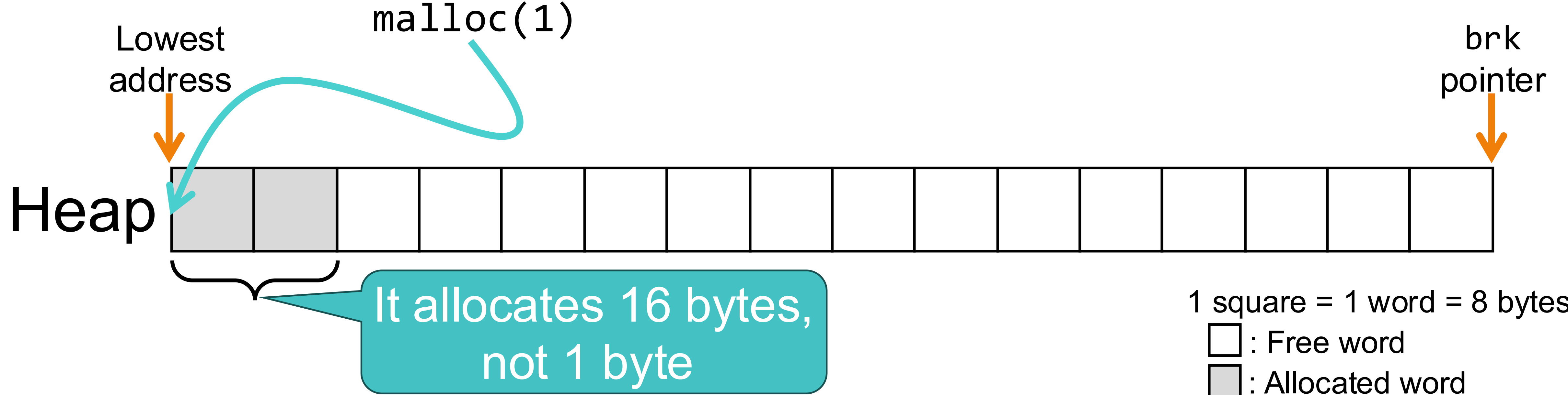
- Successful:
  - Returns a pointer to a memory block of at least size bytes <u>aligned to a 16-byte boundary (on x86-64)</u>



The malloc Package

```
#include <stdlib.h>
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least size bytes aligned to a 16-byte boundary (on x86-64)



The malloc Package

```
#include <stdlib.h>
void *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least size bytes aligned to a 16-byte boundary (on x86-64)
- Unsuccessful:
  - returns NULL (0) and sets `errno`

```
void free(void *p)
```

- Returns the block pointed at by `p` to pool of available memory
- `p` must come from a previous call to `malloc`, `calloc`, or `realloc`

Other Functions

- `calloc`: Version of `malloc` that initializes allocated block to zero
- `realloc`: Changes the size of a previously allocated block
- `sbrk`: Used internally by allocators to grow or shrink the heap

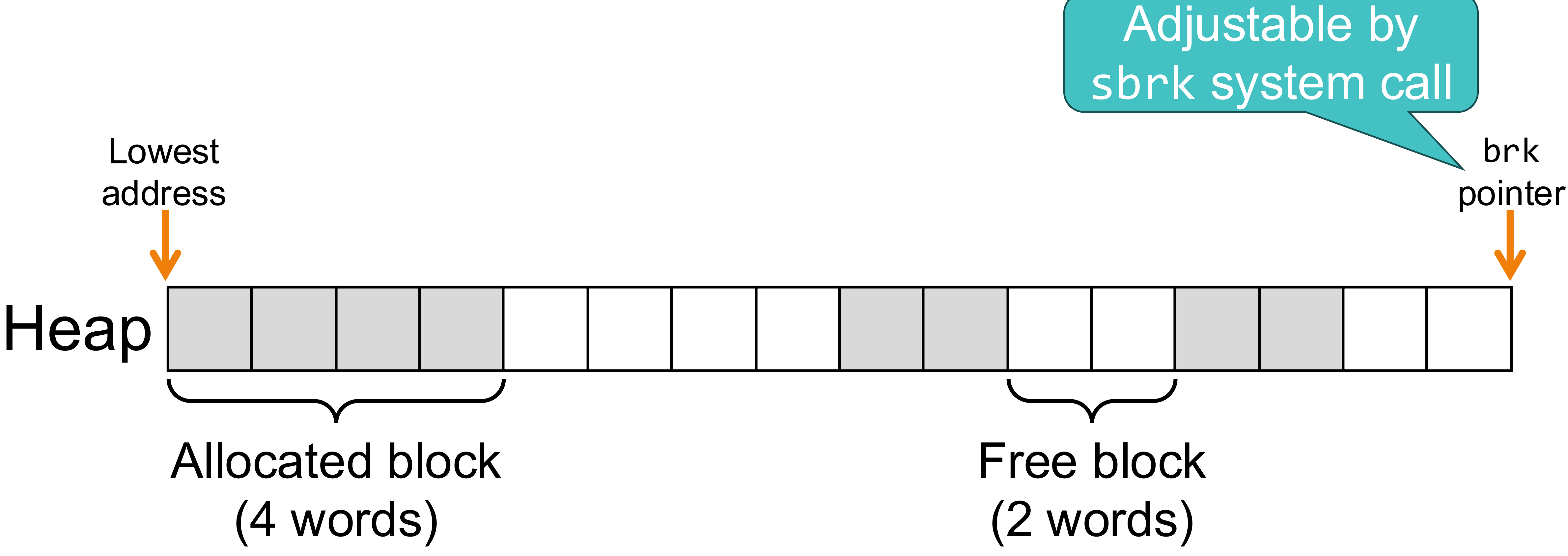
Malloc Example

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

void foo(long n) {
    long i, *p;

    p = (long *) malloc(n * sizeof(long));

    for (i=0; i<n; i++)
        p[i] = i;

    free(p);
}
```

Allocate a block of n longs

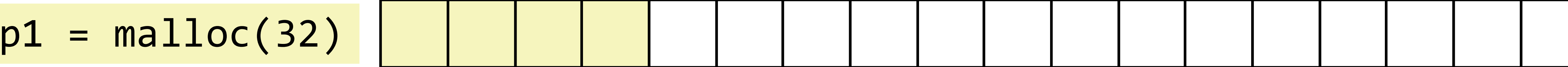
Return allocated block to the heap

Heap Visualization Convention

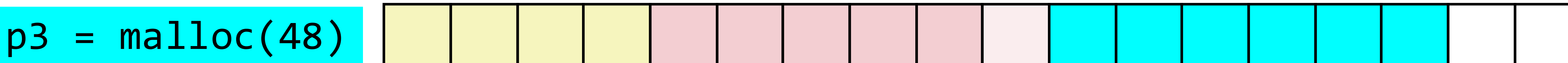


1 square = 1 word = 8 bytes
: Free word
: Allocated word

Allocation Example



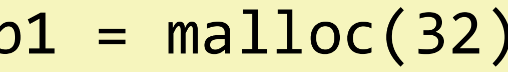
p2 = malloc(40)



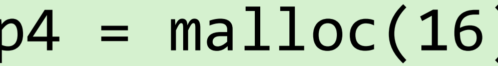
free(p2)

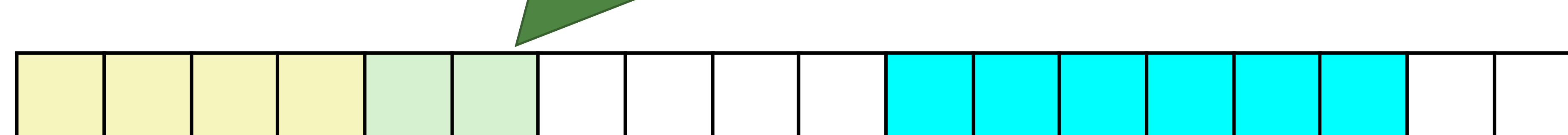
p4 = malloc(16)

Allocation Example



free(p2)





(Traditional) Constraints of Explicit Allocator

- Programs can issue arbitrary sequence of `malloc` and `free` requests

- Allocators handle only the current request as best as it can (It does not know what requests will come in the future)
  - Must respond *immediately* to `malloc` requests
  - Must allocate blocks from free memory

- Must align blocks so they satisfy all alignment requirements
  - 16-byte (x86-64) alignment on 64-bit systems

Design Goals of Heap Allocator

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

- Goals: maximize <u>throughput</u> and <u>peak memory utilization</u>

Throughput

- Measures the number of completed requests per unit time

- Example:
  - 5,000 `malloc` and 5,000 `free` calls in 10 seconds
  - Throughput =1,000 operations/second

Peak Memory Utilization

- Measures the allocator's space efficiency

- **Definition #1**: Aggregated payload (P_k)
  - The sum of the allocated payloads after request R_k

- **Definition #2**: Current heap size (H_k)
  - The heap size after request R_k (heap only grows when allocator uses `sbrk`)

- **Peak memory utilization U_k** $= max(P_0, P_1, \ldots, P_k) / H_k$
  - Peak memory utilization after request R_k

Peak Memory Utilization

- Measures the allocator's space efficiency

- **Definition #1**: Aggregated payload (P_k)
  - The sum of the allocated payloads after request R

- **Definition #2**: Current
  - The heap size after requ

Peak aggregate payload: the maximum aggregate payload at any point in the sequence up to request

- **Peak memory utilization U_k** = $max(P_0, P_1, \ldots, P_k) / H_k$
  - Peak memory utilization after request R_k

Peak Memory Utilization


```
R0 a = malloc(16)
R1 b = malloc(32)
R2 free(a)
R3 c = malloc(48)
R4 free(c)
```

P_0 = 16 bytes

P_1 = 48 bytes

P_2 = 32 bytes

P_3 = 80 bytes

P_4 = 32 bytes

- Peak aggregate payload = max(P_0, P_1, P_2, P_3, P_4) = 80 bytes
- Current heap size H_4 = 128 bytes (Assumption)

- Peak memory utilization U_4 = 80 / 128 = 62.5%

Design Goals of Heap Allocator

- Given some sequence of `malloc` and `free` requests:
  - $R_0, R_1, \ldots, R_k, \ldots, R_{n-1}$

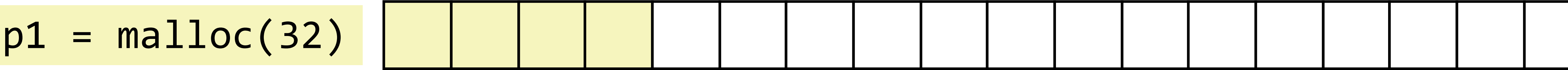
- Goals: maximize throughput and peak memory utilization

Measures the number of completed requests per unit time

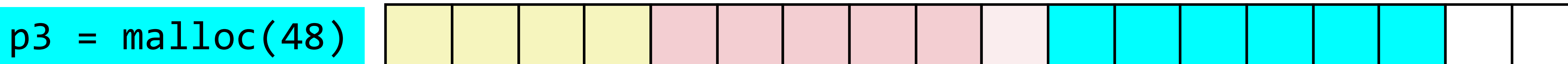
Measures the allocator's space efficiency

These goals are often conflicting

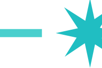
Any Problems?



```
p1 = malloc(32)

p2 = malloc(40)

p3 = malloc(48)

free(p2)

p4 = malloc(64)
```

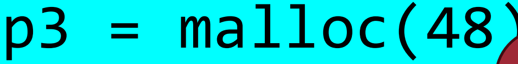


Any Problems?



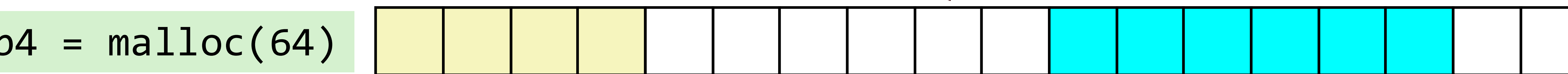
p1 = malloc(32)

p2 = malloc(40)



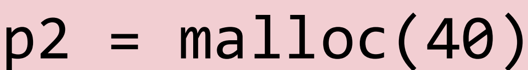
Oops! There is enough aggregate heap memory, but no single free block is large enough

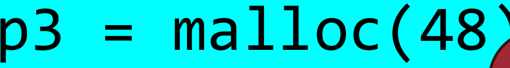
free(p2)



Any Problems?

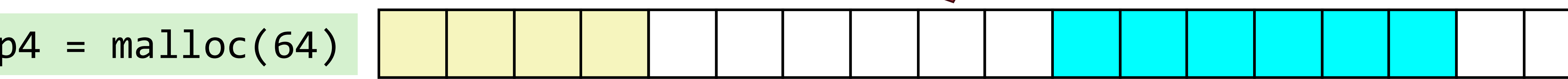
```
p1 = malloc(32)
```






```
free(p2)
```

Fragmentation (단편화, 파편화)

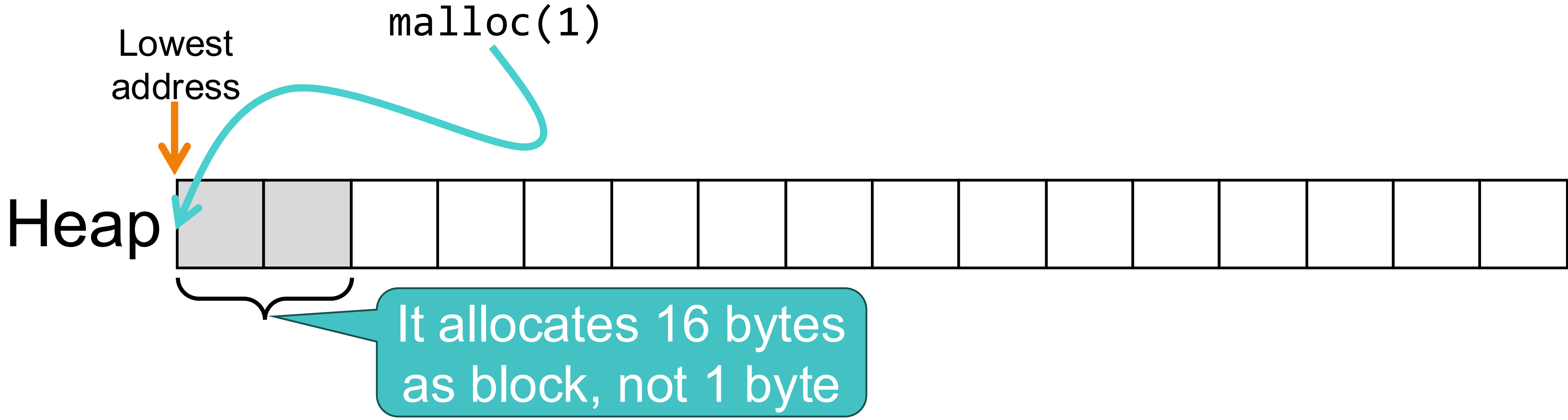


Fragmentation (단편화, 파편화)

- Poor memory utilization caused by *fragmentation*
  - ***Internal*** fragmentation
  - ***External*** fragmentation

Internal Fragmentation

Occurs if payload is smaller than block size



Internal Fragmentation

Occurs if payload is smaller than block size

`malloc(1)`

Lowest address

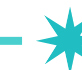
Heap

1 byte: used

15 bytes: not used

→ ***Internal fragmentation***

Internal Fragmentation



Occurs if payload is smaller than block size

- Caused by
  - Overhead of maintaining heap data structure (e.g., header, footer ← we will cover about it later)
  - Padding for alignment purposes
  - Explicit policies (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of ***previous*** requests
  - Thus, easy to measure

External Fragmentation

- Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(32)`

`p2 = malloc(40)`

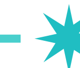
`p3 = malloc(48)`

`free(p2)`

External fragmentation

`p4 = malloc(64)`

External Fragmentation



- Occurs when there is enough aggregate heap memory, but no single free block is large enough

- Depends on the pattern of ***future*** requests
  - Thus, difficult to measure

Question?