System Level Programming

Software College of SCU

Week07

Unit 6. Memory Layout and Allocation

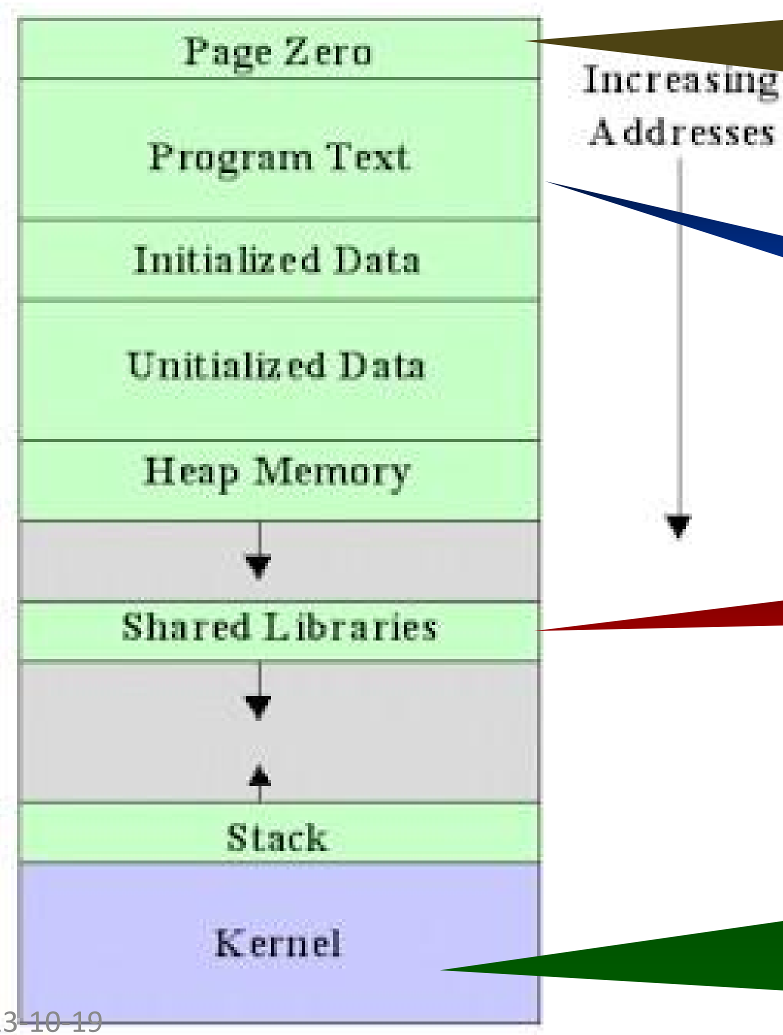
- <u>6.1 Several Uses of Memory</u>
- 6.2 Memory Bugs
- 6.3 Garbage Collection

6.1 Several Uses of Memory

- <u>6.1.1 Memory Management</u>
- 6.1.2 Static Allocation
- 6.1.3 Dynamic Allocation
- 6.1.4 Memory Management Mechanism

6.1.1 Memory Management(1/2)

Program Memory Layout



address zero is reserved and made unreadable to catch the common bug of trying to use a NULL pointer to access data

Machine code instruction

dynamically linked shared libraries

in some systems, the operating system or operating system kernel becomes part of an application's memory during system calls.

6.1.1 Memory Management(2/2)

- Variable lifetimes correspond to one of three memory allocation mechanisms:
  - Static: Absolute address retained throughout program's execution
  - Stack: Allocated & deallocated in last-in, first-out order, with function calls and returns
  - Heap: Allocated and deallocated at arbitrary times according to user's requirements.
- Stack and heap are used for dynamic allocation

6.1 Several Uses of Memory

- 6.1.1 Memory Management
- <u>6.1.2 Static Allocation</u>
- 6.1.3 Dynamic Allocation
- 6.1.4 Memory Management Mechanism

6.1.2 Static Allocation(1/11)

- The word static refers to things that happen at compile time and link time when the program is constructed.

- For example, you can define a global variable

  char a[9] =“12345678”; //assign 9 bytes for array a

- The compiler will assign 9 bytes during compilation.
  - You cannot change it even you think you need 10 bytes while running this program.
  - Of course, you can change it by modifying the program and re-compile it.

6.1.2 Static Allocation(2/11)

- Static Allocation of Variables
  - all global variables, regardless of whether or not they have been declared as static;
  - local variables explicitly declared to be static.
  - explicit constants (including strings, sets, etc)

6.1.2 Static Allocation(3/11)

```
int my_var[128]; // a statically allocated variable
static bool my_var_initialized = false; //static declaration

int my_fn(int x) {
    static int a = 5; // static local variable
    if (my_var_initialized) return;
    my_var_initialized = true;
    for (int i = 0; i < 128; i++)
        my_var[i] = 0;
}
```

functions declared in other files cannot access my_var_initialized

variable a is statically allocated and only visible within the function

6.1.2 Static Allocation(4/11)

- The following program has three different kinds of statically allocated variable.
- The program is built from three source files. There is a header file ...

```
/* foo.h */
void foo(void);
```

6.1.2 Static Allocation(5/11)

```
/* foo.c */
#include <stdio.h>
#include "foo.h“
/* Tell compiler evil_glob is defined in another .c file. */
extern int evil_glob;
void foo(void) {
  int on_stack = 100;
  static int not_on_stack = 200;
  evil_glob--;
  on_stack++;
  not_on_stack++;
  printf("In foo ...\n");
  printf(" value of on_stack: %d\n", on_stack);
  printf(" value of not_on_stack: %d\n", not_on_stack); }
```

6.1.2 Static Allocation(6/11)

```c
/* main.c */
#include <stdio.h>
#include "foo.h"
int evil_glob = 999; /* accessible from all .c files */
static int good_glob = 333; /* private to this .c file */
int main(void) {
    evil_glob--;
    good_glob++;
    foo();
    foo();
    printf("At end, evil_glob is %d and good_glob is %d.\n",
                evil_glob, good_glob);
    return 0;
}
```

6.1.2 Static Allocation(7/11)

- Result

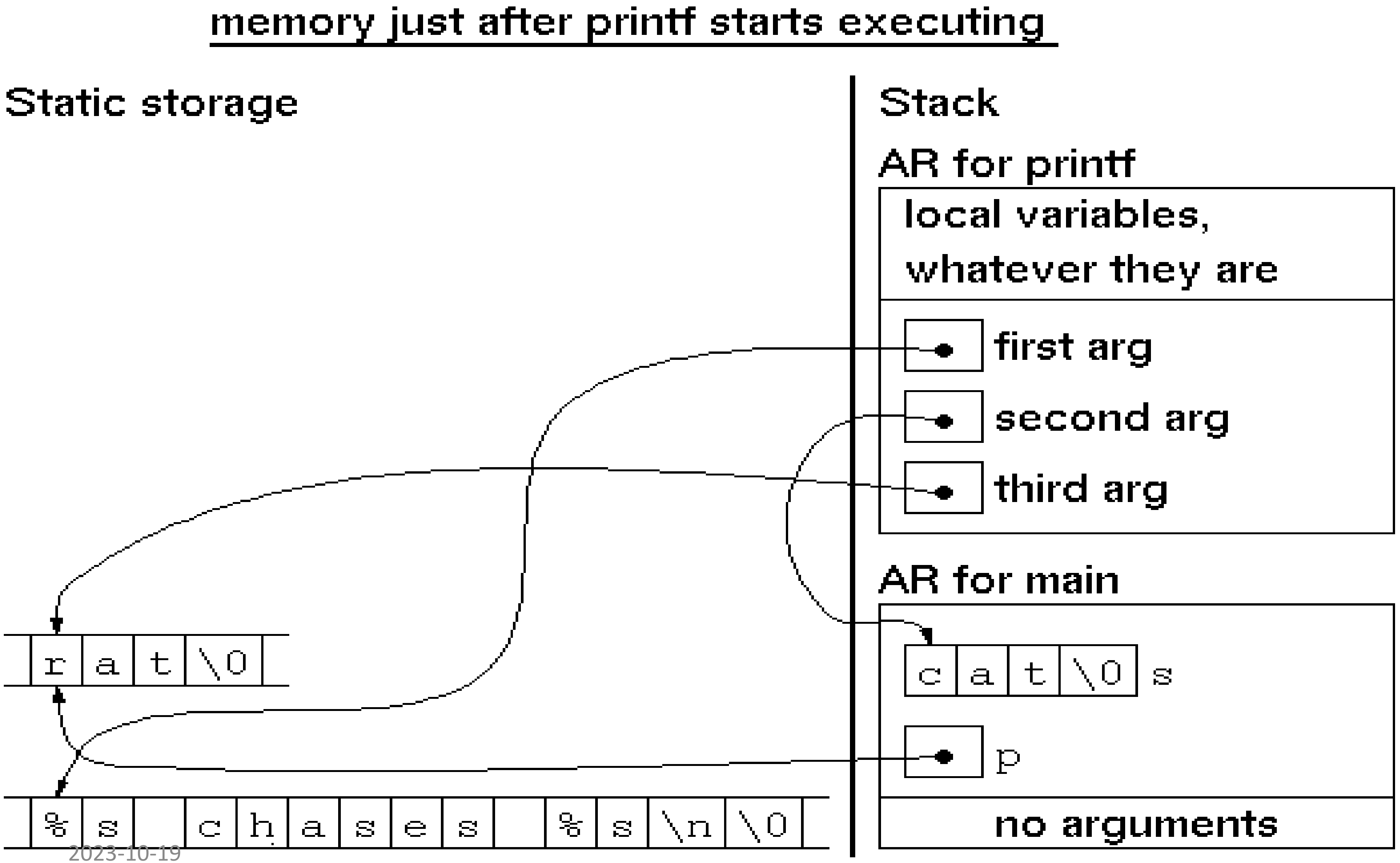
```
In foo ...
value of on_stack: 101
value of not_on_stack: 201
In foo ...
value of on_stack: 101
value of not_on_stack: 202
At end, evil_glob is 996 and good_glob is 334.
```

6.1.2 Static Allocation(8/11)

- String constants are statically allocated.
  - In most expressions a string constant generates a pointer to the first char in the string

```c
#include <stdio.h>
int main(void) {
  char s[] = "cat";
  char *p = "rat";
  printf("%s chases %s\n", s, p);
  return 0;
}
```

6.1.2 Static Allocation(9/11)



6.1.2 Static Allocation(10/11)

```
00401017 56                   push      esi
00401018 57                   push      edi
00401019 8D 7D B8             lea       edi,[ebp-48h]
0040101C B9 12 00 00 00       mov       ecx,12h
00401021 B8 CC CC CC CC       mov       eax,0CCCCCCCCh
00401026 F3 AB                rep stos  dword ptr [edi]
5:        char s[] = "cat";
00401028 A1 30 00 42 00       mov       eax,[string "cat" (00420030)]
0040102D 89 45 FC             mov       dword ptr [ebp-4],eax
6:        char *p = "rat";
00401030 C7 45 F8 2C 00 42 00 mov       dword ptr [ebp-8],offset string "rat" (0042002c)
7:        printf("%s chases %s\n", s, p);
00401037 8B 4D F8             mov       ecx,dword ptr [ebp-8]
0040103A 51                   push      ecx
0040103B 8D 55 FC             lea       edx,[ebp-4]
0040103E 52                   push      edx
0040103F 68 1C 00 42 00       push      offset string "%s chases %s\n" (0042001c)
00401044 E8 37 00 00 00       call      printf (00401080)
00401049 83 C4 0C             add       esp,0Ch
8:        return 0;
0040104C 33 C0                x
9:    }
0040104E 5F                   p
0040105D 5D                   p
0040105E C3                   r
```

Debug

Watch

Name	Value
⊞ &s[0]	0x0013ff7c "cat"
⊞ s	0x0013ff7c "cat"

Memory

Address: 0x0013ff7c

```
0013FF7C  63 61 74 00  cat.
0013FF80  C0 FF 13 00  ....
0013FF84  29 12 40 00  ).@.
0013FF88  01 00 00 00  ....
0013FF8C  80 0E 43 00  ..C.
0013FF90  90 0D 43 00  ..C.
0013FF94  00 00 00 00  ....
0013FF98  E8 F8 23 03  杪#.
```

Memory

Address: 0x00420030

```
0042001C  25 73 20 63  %s c
00420020  68 61 73 65  hase
00420024  73 20 25 73  s %s
00420028  0A 00 00 00  ....
0042002C  72 61 74 00  rat.
00420030  63 61 74 00  cat.
00420034  70 72 69 6E  prin
00420038  74 66 2E 63  tf.c
```

6.1.2 Static Allocation(11/11)

- Limitations of static allocation
  - Naming gets to be a problem
  - Programs do not always know how much storage is required until run time, so static allocation is inaccurate.
    - E.g., you allocated 5 bytes, but later you find that it needs 6 bytes.
  - Static allocation reserves memory for the duration of the program, but often a data structure is only needed temporarily.
  - Recursive procedures are impossible.

6.1 Several Uses of Memory

- 6.1.1 Memory Management
- 6.1.2 Static Allocation
- <u>6.1.3 Dynamic Allocation</u>
- 6.1.4 Memory Management Mechanism

6.1.3 Dynamic Allocation

- <u>6.1.3.1 Dynamic Memory Allocation Concept</u>
- 6.1.3.2 Stack Allocation
- 6.1.3.3 Heap Allocation
- 6.1.3.4 Stack vs. Heap

6.1.3.1 Dynamic Memory Allocation Concept(1/1)

- Dynamic memory allocation is the allocation of memory storage during the runtime of program.

  - This is in contrast to static memory allocation, which has a fixed duration.

6.1.3 Dynamic Allocation

- 6.1.3.1 Dynamic Memory Allocation Concept
- <u>6.1.3.2 Stack Allocation</u>
- 6.1.3.3 Heap Allocation
- 6.1.3.4 Stack vs. Heap

6.1.3.2 Stack Allocation(1/9)

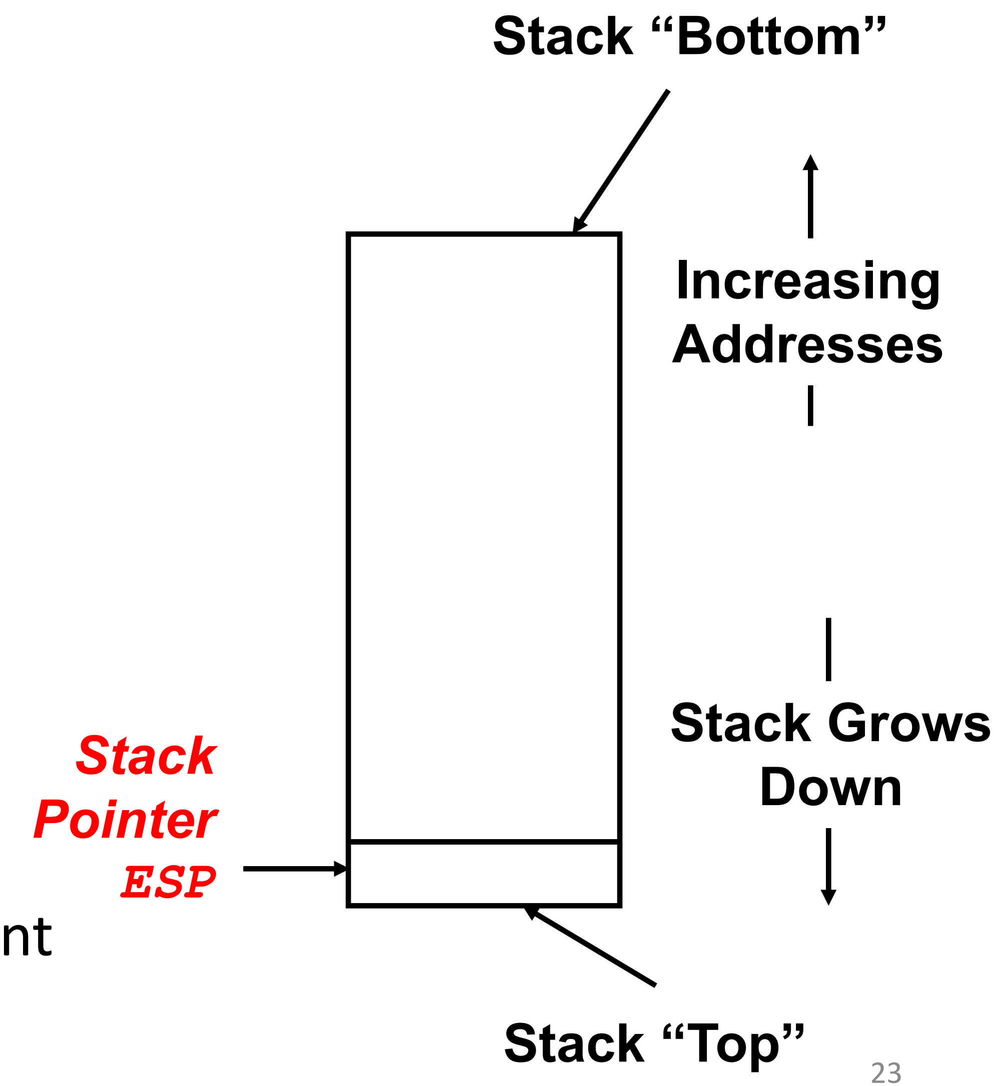
- It is the most common form of dynamic allocation.

- Allocated & deallocated in last-in, first-out order, with functions calls and returns.

- It is a standard feature of modern programming languages, including C and C++, as stacks support recursion.

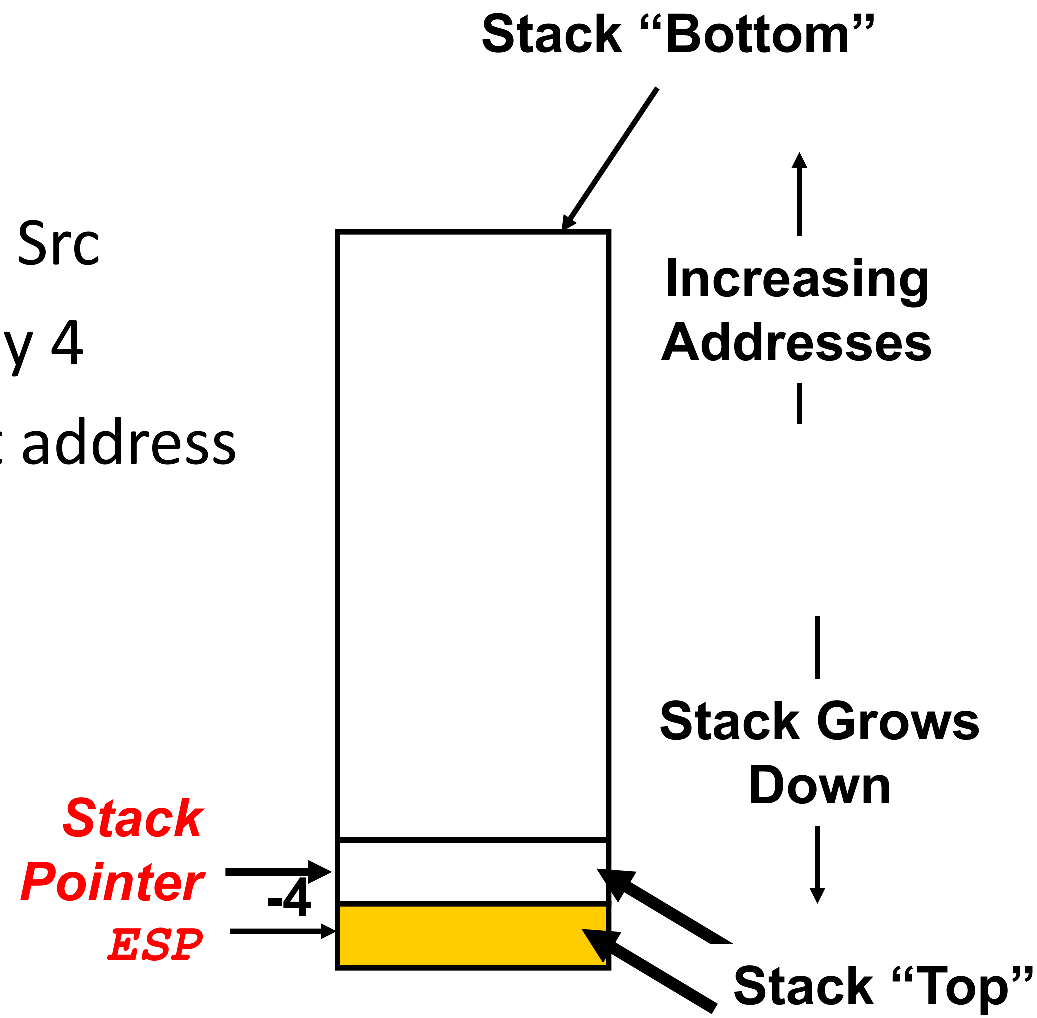
6.1.3.2 Stack Allocation(2/9)

- IA32 Stack
  - Region of memory managed with stack discipline
  - Grows toward lower addresses
  - Register EBP indicates highest stack address
  - Register ESP indicates lowest stack address
    - address of top element



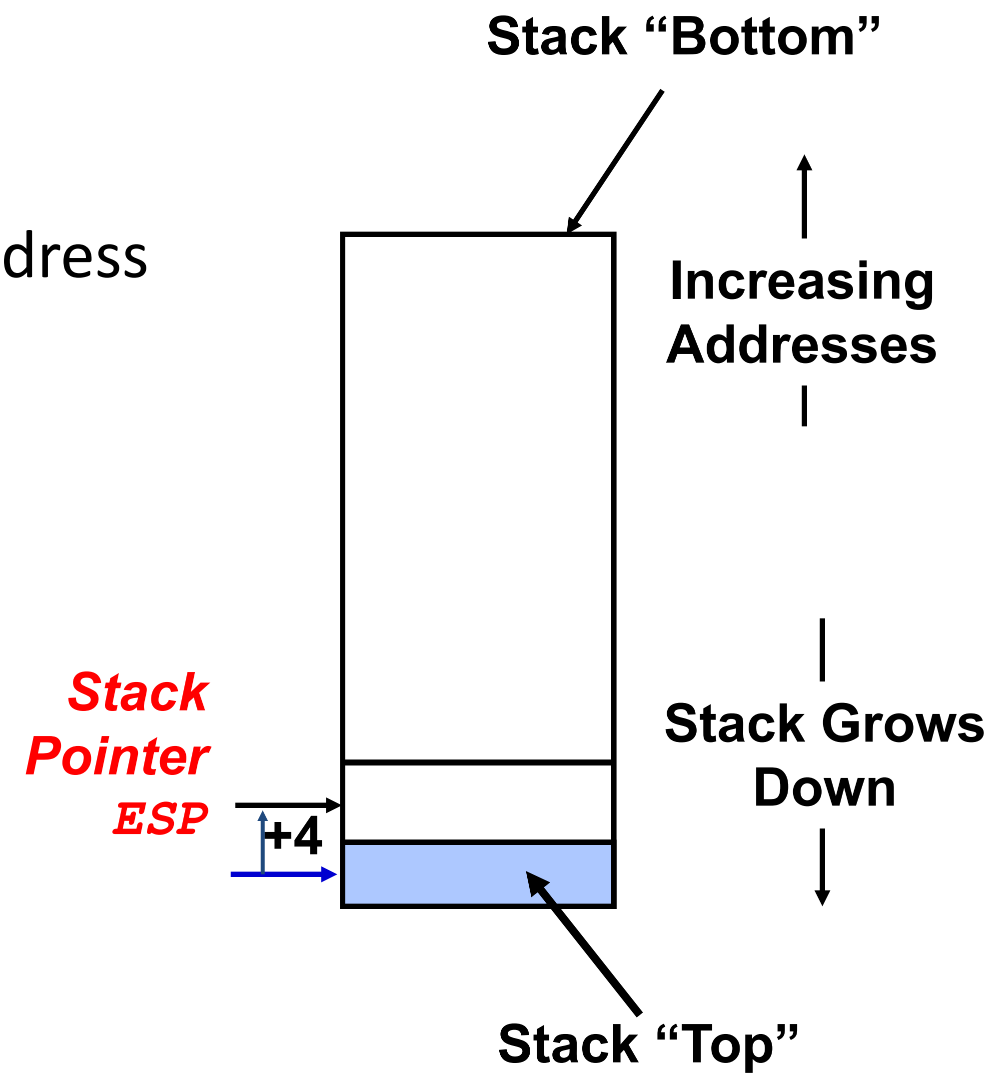
6.1.3.2 Stack Allocation(3/9)

- Pushing
  - **push Src**
  - Fetch operand at Src
  - Decrement esp by 4
  - Write operand at address given by ESP

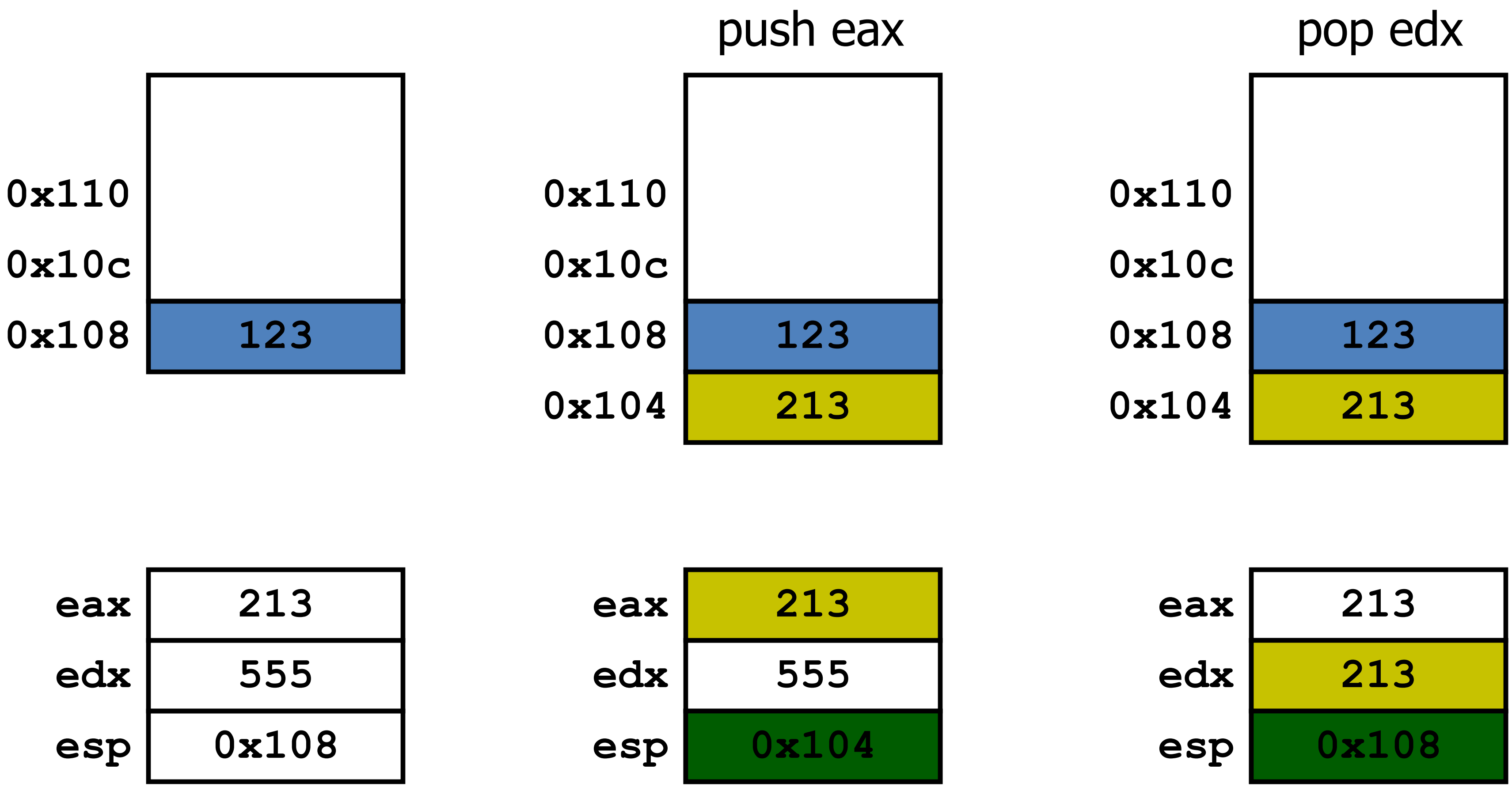


6.1.3.2 Stack Allocation(4/9)

- Popping
  - **pop Dest**
  - Read operand at address given by ESP
  - Increment esp by 4
  - Write to Dest



6.1.3.2 Stack Allocation(5/9)



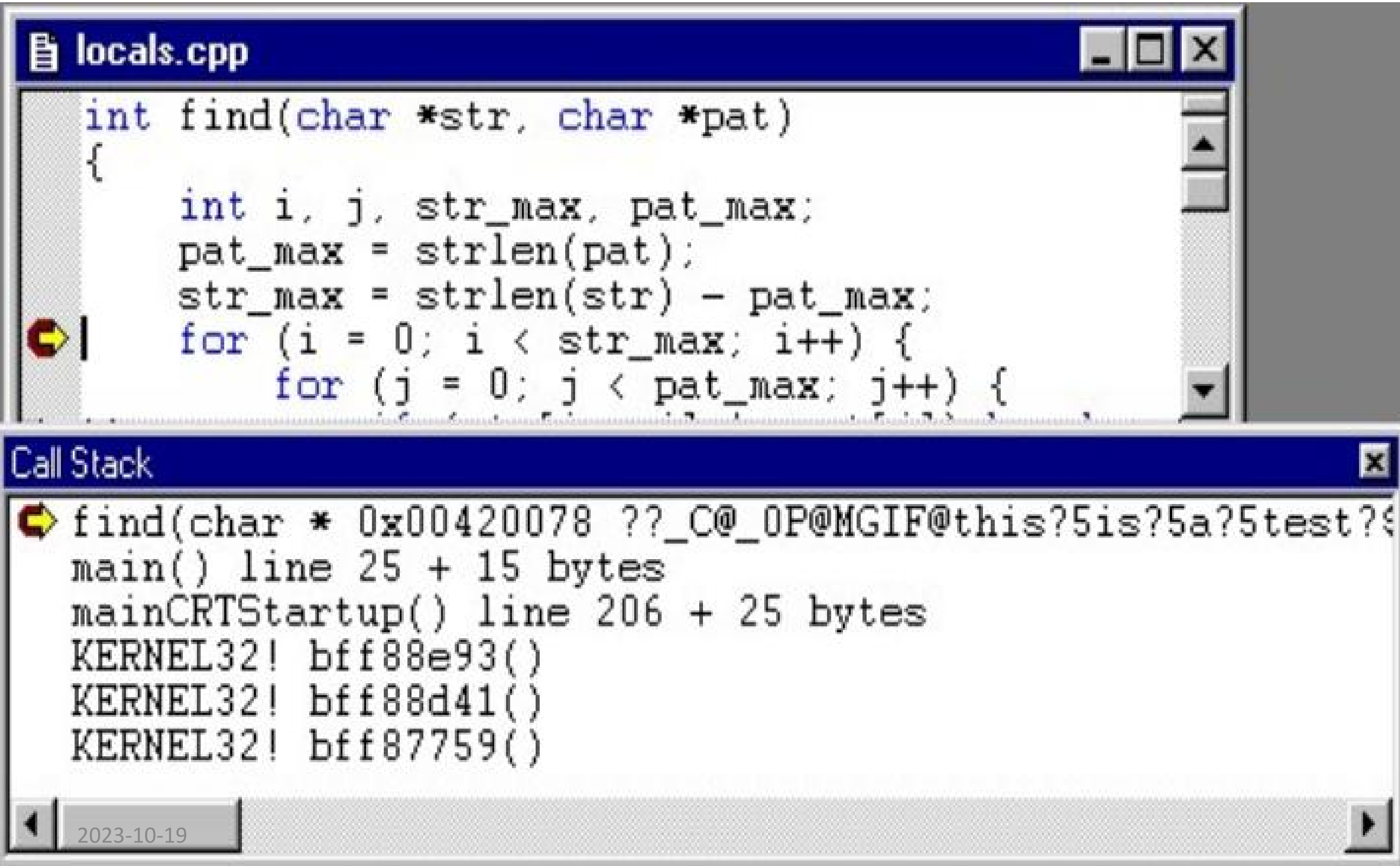
6.1.3.2 Stack Allocation(6/9)

- Stack Allocation For Local Variables

```
…
int find(char *str, char *pat) {
    int i, j, str_max, pat_max;
    pat_max = strlen(pat);
    str_max = strlen(str) - pat_max;
    for (i = 0; i < str_max; i++) {
      for (j = 0; j < pat_max; j++) {
            if (str[i + j] != pat[j]) break;
      }
      if (j == pat_max) return i;
    }
    return -1;  }
```

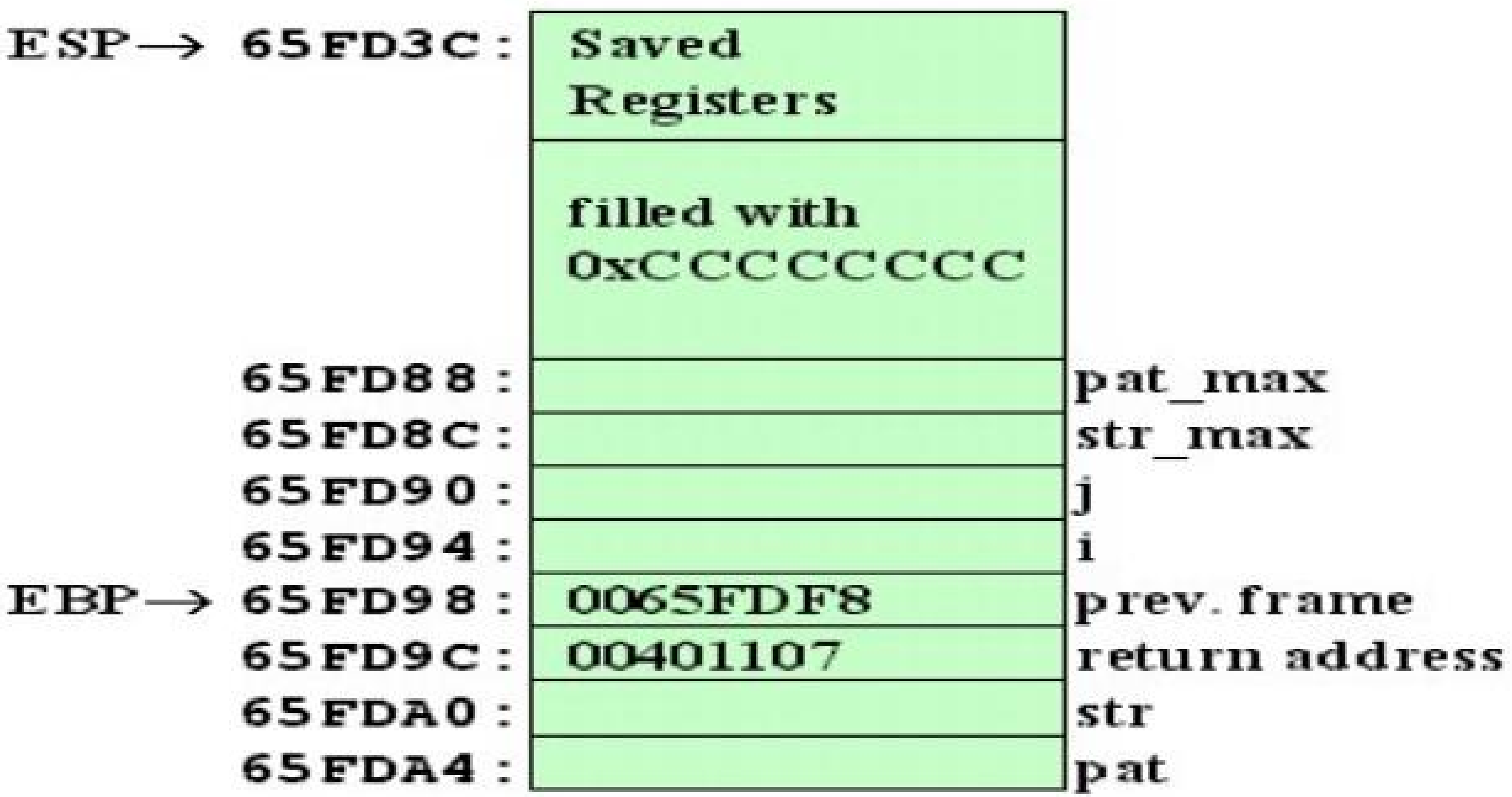
```
void main() {
    printf("find(\"this is a test\",
    \"is\") -> %d\n", find("this is
    a test", "is"));

   printf("find(\"this is a test\",
    \"IS\" -> %d\n", find("this is
    a test", "IS"));
}
```

6.1.3.2 Stack Allocation(7/9)



6.1.3.2 Stack Allocation(8/9)

- Local Variables on Stack



6.1.3.2 Stack Allocation(9/9)

- What's the problem of returning local variable?

```
int* ptr_to_zero() {
            int i[5]={1,2,3,4,5};
           return &i;
}
```

6.1.3 Dynamic Allocation

- 6.1.3.1 Dynamic Memory Allocation Concept
- 6.1.3.2 Stack Allocation
- <u>6.1.3.3 Heap Allocation</u>
- 6.1.3.4 Stack vs. Heap

6.1.3.3 Heap Allocation(1/12)

- Heap in memory space
  - A chunk of memory used for dynamic memory allocation
  - Allocated and deallocated memory space at arbitrary times.

6.1.3.3 Heap Allocation(2/12)

- To grab memory, use *malloc*(size).
- To free this pointer to the memory, we use *free*(ptr).

6.1.3.3 Heap Allocation(3/12)

- **malloc()** is short for “memory allocation”
  - Takes as a parameter the number of bytes to take from the heap
  - Returns a pointer to the memory that was just allocated

```
int  main() {
     int* myInt =  (int*) malloc(sizeof(int));
     if  ( myInt  != NULL )  {
          *myInt = 5;
          free(myInt);
      }
      return 0;
}
```

6.1.3.3 Heap Allocation(4/12)

- **free()** releases memory we aren't using anymore
  - Takes a pointer to the memory to be released
  - Must have been allocated with malloc()

```
int main() {
    int *myInt = (int*) malloc(sizeof(int));
    *myInt = 5;
    free(myInt);    //use free() to release memory
    myInt = NULL;
    return 0;
}
```

6.1.3.3 Heap Allocation(5/12)

- Do not
  - free() memory on the stack
  - Lose track of malloc()'d memory
  - free() the same memory twice

```
int main() {
    int myInt;
    free(&myInt);  //very bad
    return 0;
}
```

```
int main() {
    int *A = malloc(sizeof(int));
    int *B = A;
    free(A);
    free(B);  //very bad
    return 0;
}
```

```
int main() {
    int *myInt = malloc(sizeof(int));
    myInt = 0;
    //how can you free it now?
    return 0;
}
```

memory leak

6.1.3.3 Heap Allocation(6/12)

- MSDN malloc
  - Allocates memory blocks.
  - void *malloc(size_t size);

```
#include <stdlib.h> /* For _MAX_PATH definition */
#include <stdio.h>
#include <malloc.h>
void main( void ) {
    char* string; /* Allocate space for a path name */
    string = malloc( _MAX_PATH );
    if( string == NULL )  printf( "Insufficient memory available\n" );
    else {
          printf( "Memory space allocated for path name\n" );
          free( string );
          printf( "Memory freed\n" );  }
}
```

6.1.3.3 Heap Allocation(7/12)

- MSDN calloc
  - Allocates an array in memory with elements initialized to 0.
  - void *calloc(size_t num, size_t size);

```
/* CALLOC.C: This program uses calloc to allocate space for
 * 40 long integers. It initializes each element to zero. */
#include <stdio.h>
#include <malloc.h>
void main( void ) {
          long* buffer;
          buffer = (long*) calloc( 40, sizeof( long ) );
          if( buffer != NULL )
               printf( "Allocated 40 long integers\n" );
          else
               printf( "Can't allocate memory\n" );
          free( buffer );
}
```

6.1.3.3 Heap Allocation(8/12)

- MSDN realloc
  - Reallocate memory blocks.
  - void *realloc(void *memblock, size_t size);

```
/* REALLOC.C: This program allocates a block of memory for
 * buffer and then uses _msize to display the size of that  block.
 * Next, it uses realloc to expand the amount of  memory used
 * by buffer and then calls _msize again to  display the new
 * amount of memory allocated to buffer. */
```

```c
#include <stdio.h>
#include <malloc.h>
#include <stdlib.h>
void main( void ) {
    long* buffer;
    size_t size;
    if( (buffer = (long *)malloc( 1000 * sizeof( long ) )) == NULL )
            exit( 1 );
    size = _msize( buffer );
    printf( "Size of block after malloc of 1000 longs: %u\n", size );
    /* Reallocate and show new size: */
    if( (buffer = realloc( buffer, size + (1000 * sizeof( long )) )) == NULL )
            exit( 1 );
    size = _msize( buffer );
   printf( "Size of block after realloc of 1000 more longs: %u\n", size );
    free( buffer );
    exit( 0 );
}
```

```
Size of block after malloc of 1000 longs: 4000
Size of block after realloc of 1000 more longs: 8000
```

6.1.3.3 Heap Allocation(10/12)

- MSDN free
  - Deallocates or frees a memory block.
  - void free(void *memblock);

6.1.3.3 Heap Allocation(11/12)

- Use malloc to allocate space on the heap for a string

```
int main() {
   int n = rand() % 100;        // random number between 0 and 99
   char *string = (char*)malloc(n);  // a char is one byte
   for(int i = 0; i < n-1; i++)   string[i] = ‘A’;
    string[n–1] = ‘\0’;               // a string must be null terminated!
    free(string);                     // make sure to free the string!
   string = NULL;
   return 0;
}
```

6.1.3.3 Heap Allocation(12/12)

- Use malloc to allocate space on the heap for a struct

```
typedef struct foo {
   int  value;
   int[10] array;
} foo_t;
int main() {
   foo_t *fooStruct = (foo_t*) malloc(sizeof(foo_t));
   fooStruct->value = 5;   //  （*fooStruct).value = 5;
   for(int i = 0; i < 10; i++)  fooStruct->array[i] = 0;
   free(fooStruct);  // free the struct. DON'T need to also free the array
   fooStruct = NULL;
   return 0;
}
```

6.1.3 Dynamic Allocation

- 6.1.3.1 Dynamic Memory Allocation Concept
- 6.1.3.2 Stack Allocation
- 6.1.3.3 Heap Allocation
- <u>6.1.3.4 Stack vs. Heap</u>

6.1.3.4 Stack vs. Heap(1/6)

- Stack vs. Heap
  - Both are sources from which memory is allocated
  - Stack is automatic (implicit)
    - Created when "in scope"
    - Destroyed when "out of scope"
  - Heap is manual (explicit)
    - Created upon request
    - Destroyed upon request

6.1.3.4 Stack vs. Heap(2/6)

- Two ways to get an int
  - 1. On the Stack:

```
void main() {
    int myInt;    // declare an int on the stack
    myInt = 5;   // set the memory to five
}
```

  - 2. On the Heap:

```
void main() {
    int *myInt = (int*)malloc(sizeof(int));
    // allocate mem. from heap
   *myInt = 5;  // set the memory to five
    free(myInt);
}
```

6.1.3.4 Stack vs. Heap(3/6)

- Static Array
  - Size determined at compile time.

```
int main() {
    int array[5];         // static array of size 5
    for(int i = 0; i < 5; i++)
        array[i] = 0;      // initialize the array to 0
   return 0;
}
```

6.1.3.4 Stack vs. Heap(4/6)

- Dynamic Array
  - use malloc to allocate enough space for the array at run-time

```
int main() {
   int n = rand() % 100;      // random number between 0 and 99
   int *array = malloc(n * sizeof(int));
   // allocate array of size 'n' using malloc
   for(int i = 0; i < n; i++) {
      array[i] = 0;
   }
   free(array);  // make sure to free the array!
   array = NULL;
   return 0;
}
```

6.1.3.4 Stack vs. Heap(5/6)

- Stack vs. Heap
  - Stack
    - very fast access
    - don't have to explicitly de-allocate variables
    - space is managed efficiently by CPU, memory will not become fragmented
    - local variables only
    - limit on stack size (VC default stack size: 1M)
    - variables cannot be resized

6.1.3.4 Stack vs. Heap(6/6)

- Stack vs. Heap
  - Heap
    - variables can be accessed globally
    - no limit on memory size
    - (relatively) slower access
    - no guaranteed efficient use of space, memory may become fragmented over time as blocks of memory are allocated, then freed
    - you must manage memory (you're in charge of allocating and freeing variables)
    - variables can be resized using realloc()

6.1 Several Uses of Memory

- 6.1.1 Memory Management
- 6.1.2 Static Allocation
- 6.1.3 Dynamic Allocation
- <u>6.1.4 Memory Management Mechanism</u>

6.1.4 Memory Management Mechanism

- 6.1.4.1 Free block organization
- 6.1.4.2 Placement Policy
- 6.1.4.3 Coalescing

6.1.4.1 Free block organization(1/8)

- Implicit list using lengths
  - links all blocks

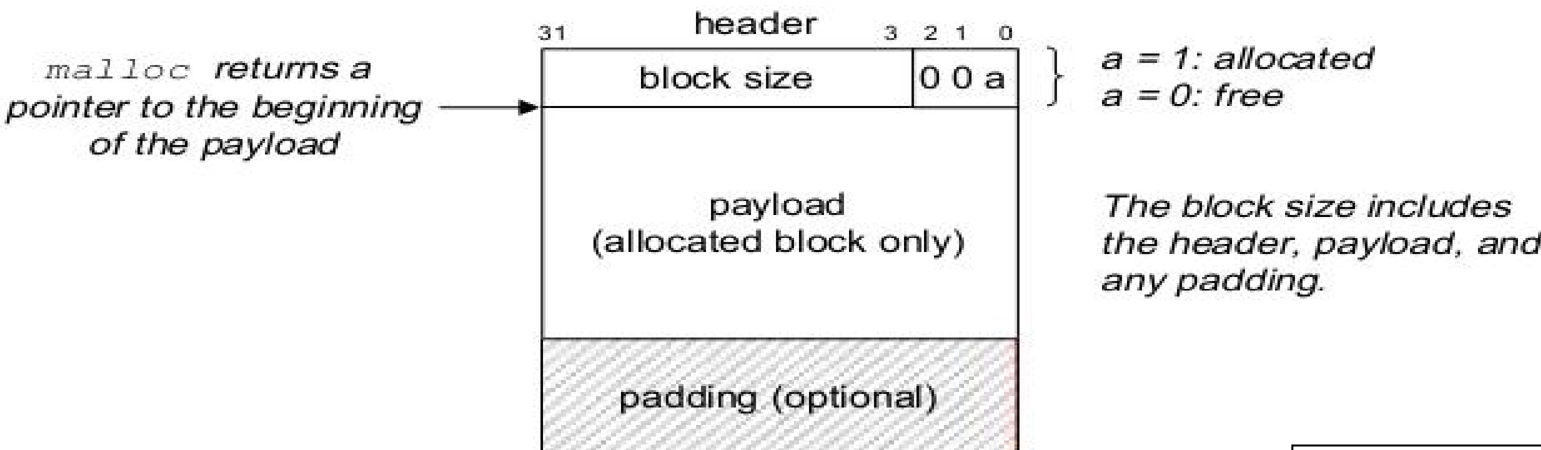
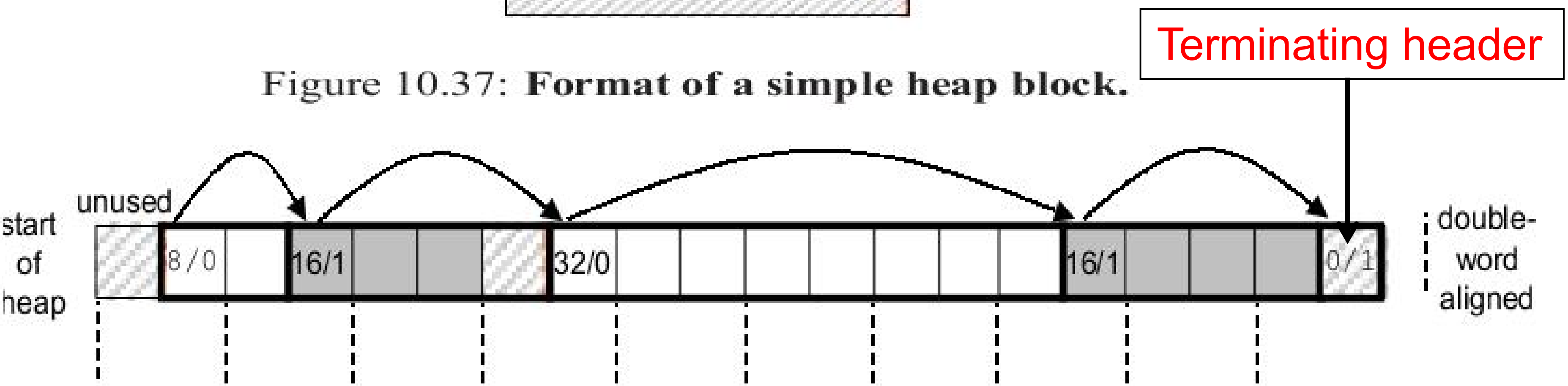


Figure 10.37: **Format of a simple heap block.**



6.1.4.1 Free block organization(2/8)

- Explicit list among the free blocks using pointers within the free blocks

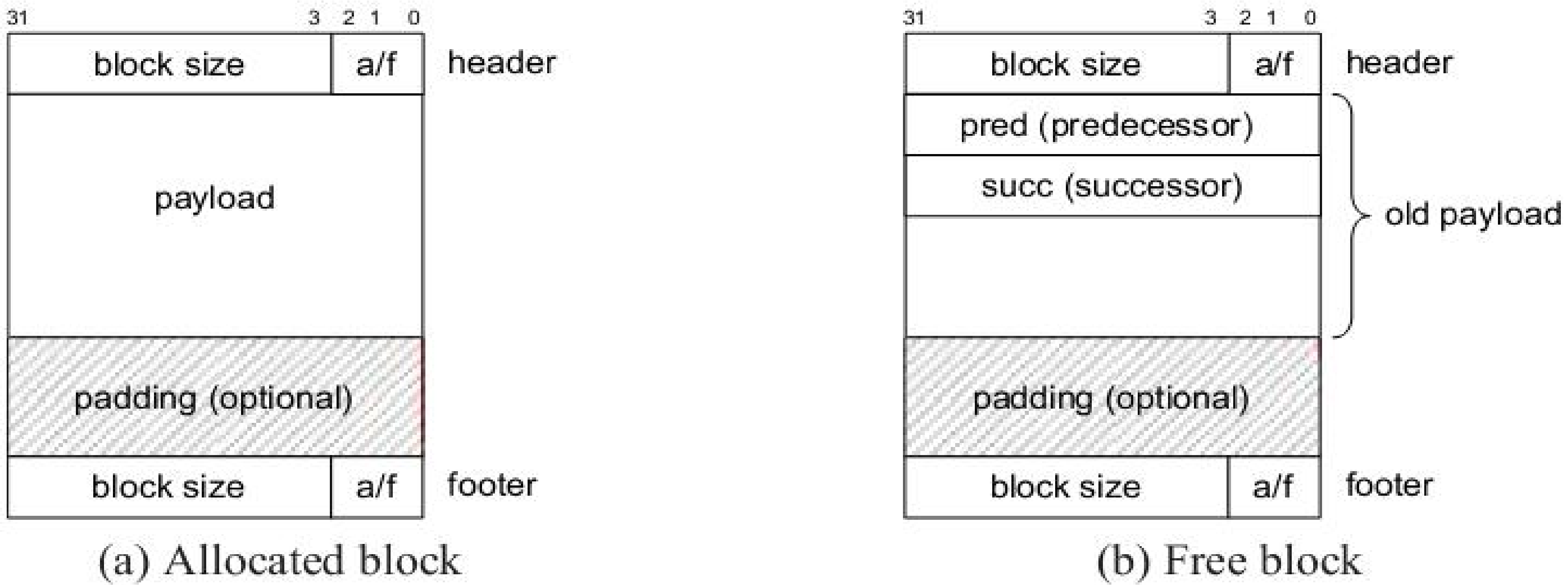
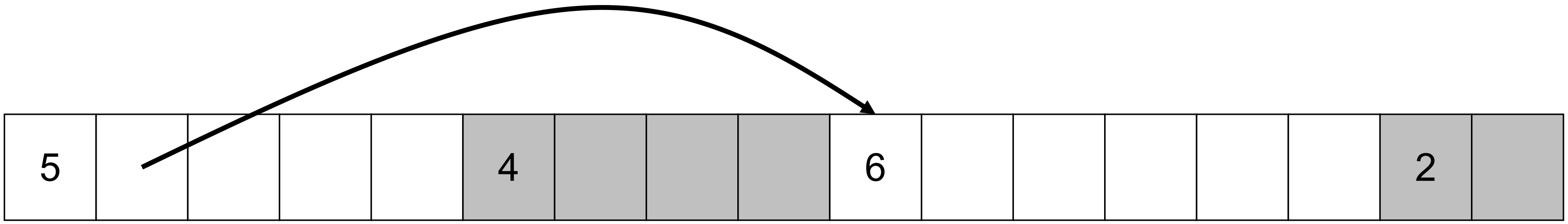


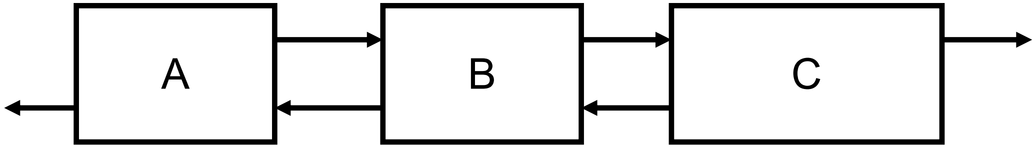
Figure 10.50: **Format of heap blocks that use doubly-linked free lists.**

6.1.4.1 Free block organization(3/8)

- Explicit list among the free blocks using pointers within the free blocks

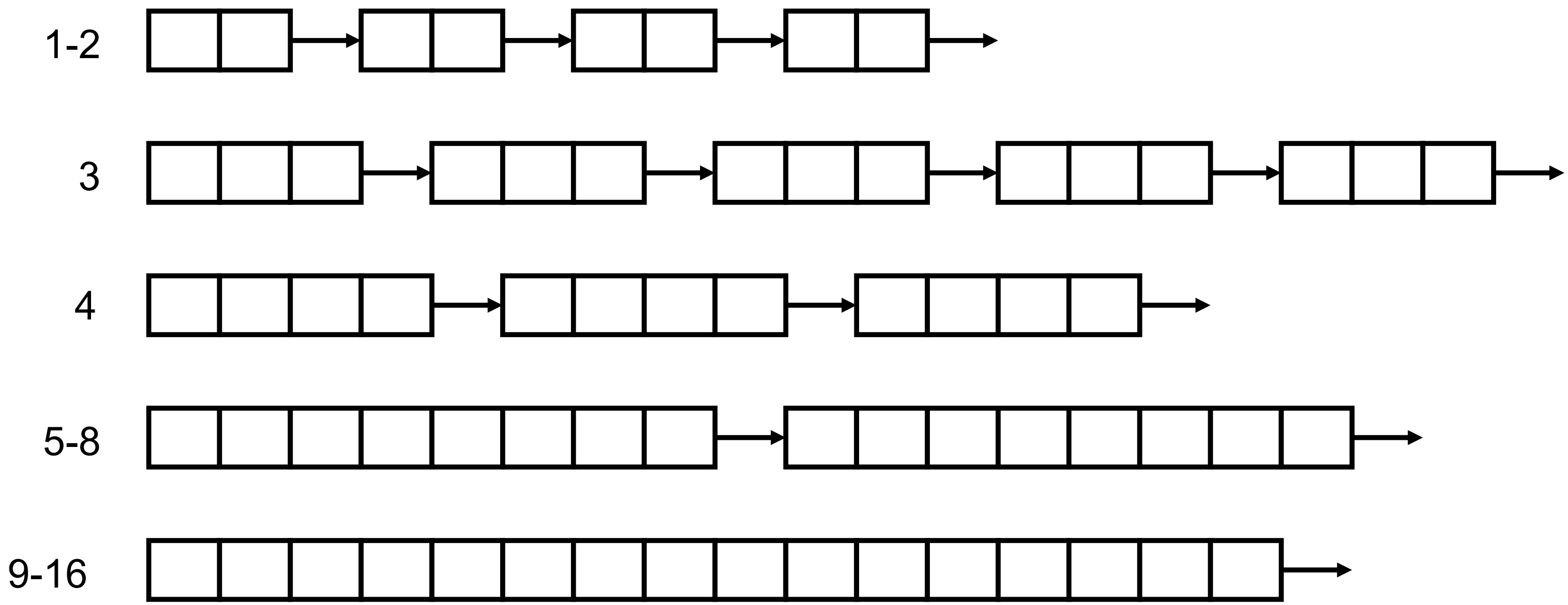


  - Typically doubly linked



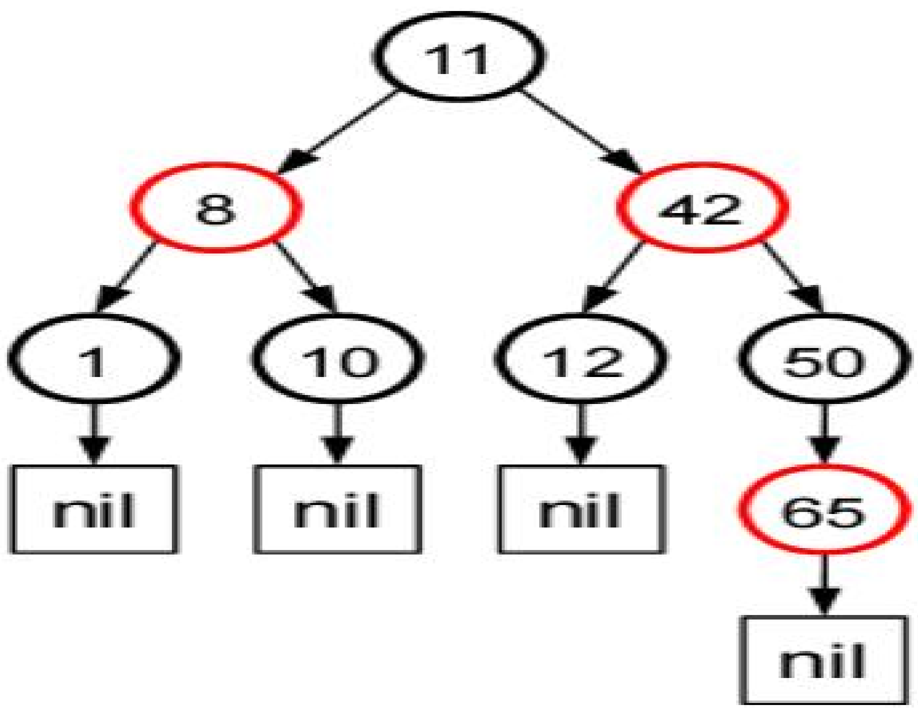
6.1.4.1 Free block organization(4/8)

- Segregated free lists
  - Different free lists for different size classes



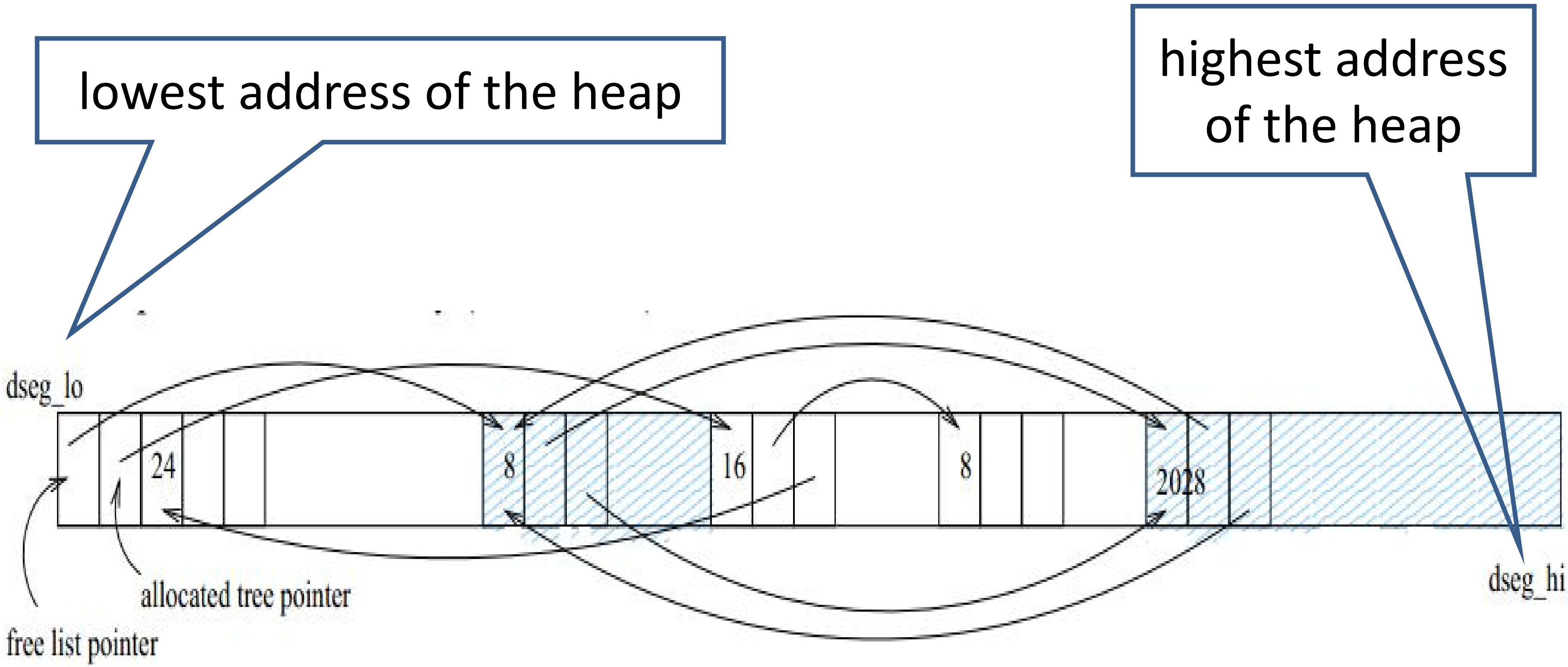
6.1.4.1 Free block organization(5/8)

- Blocks sorted by size
  - Can use a balanced tree (e.g. Red-Black tree) with pointers within each free block, and the length used as a key



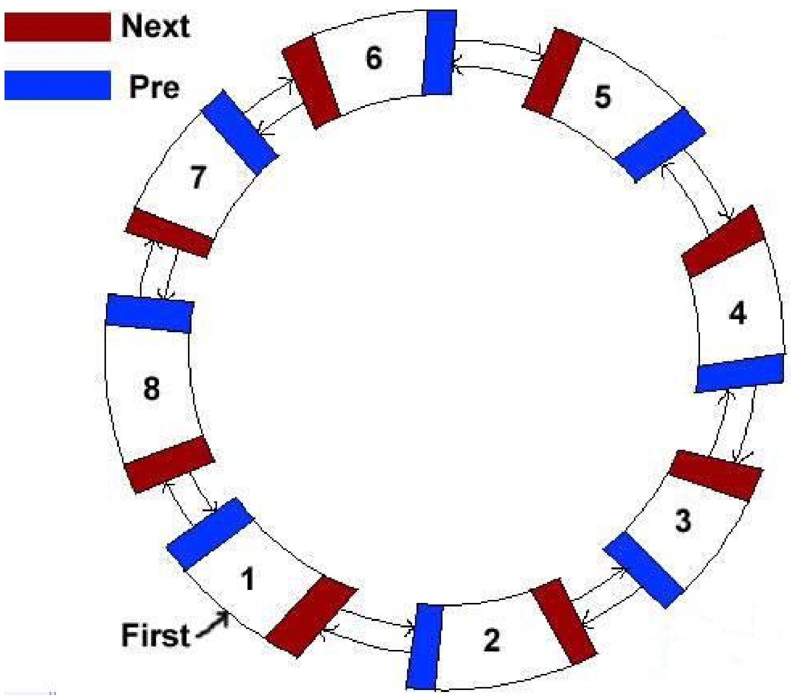
6.1.4.1 Free block organization(6/8)

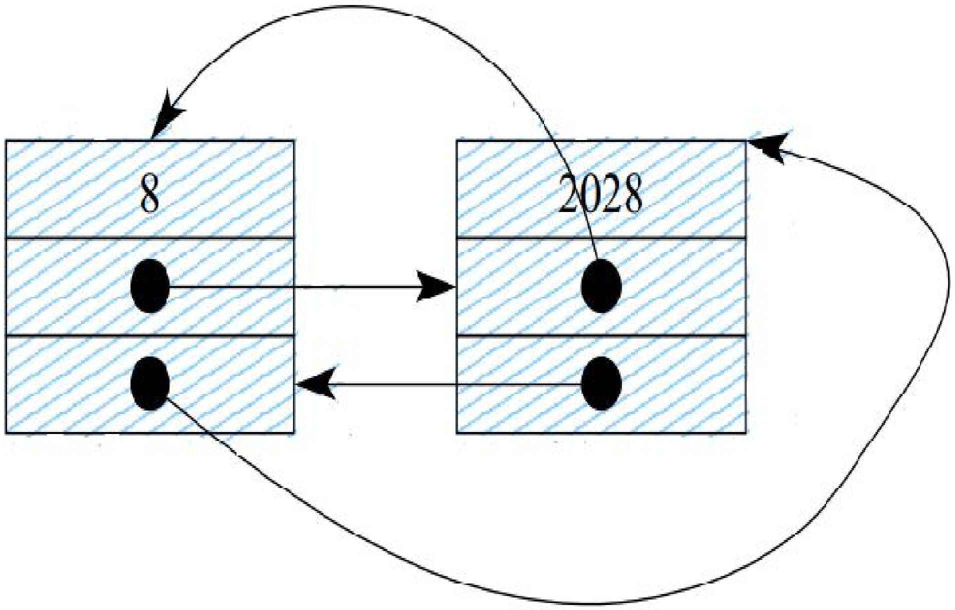
- Freelist and Allocation Tree in Heap Memory



6.1.4.1 Free block organization(7/8)

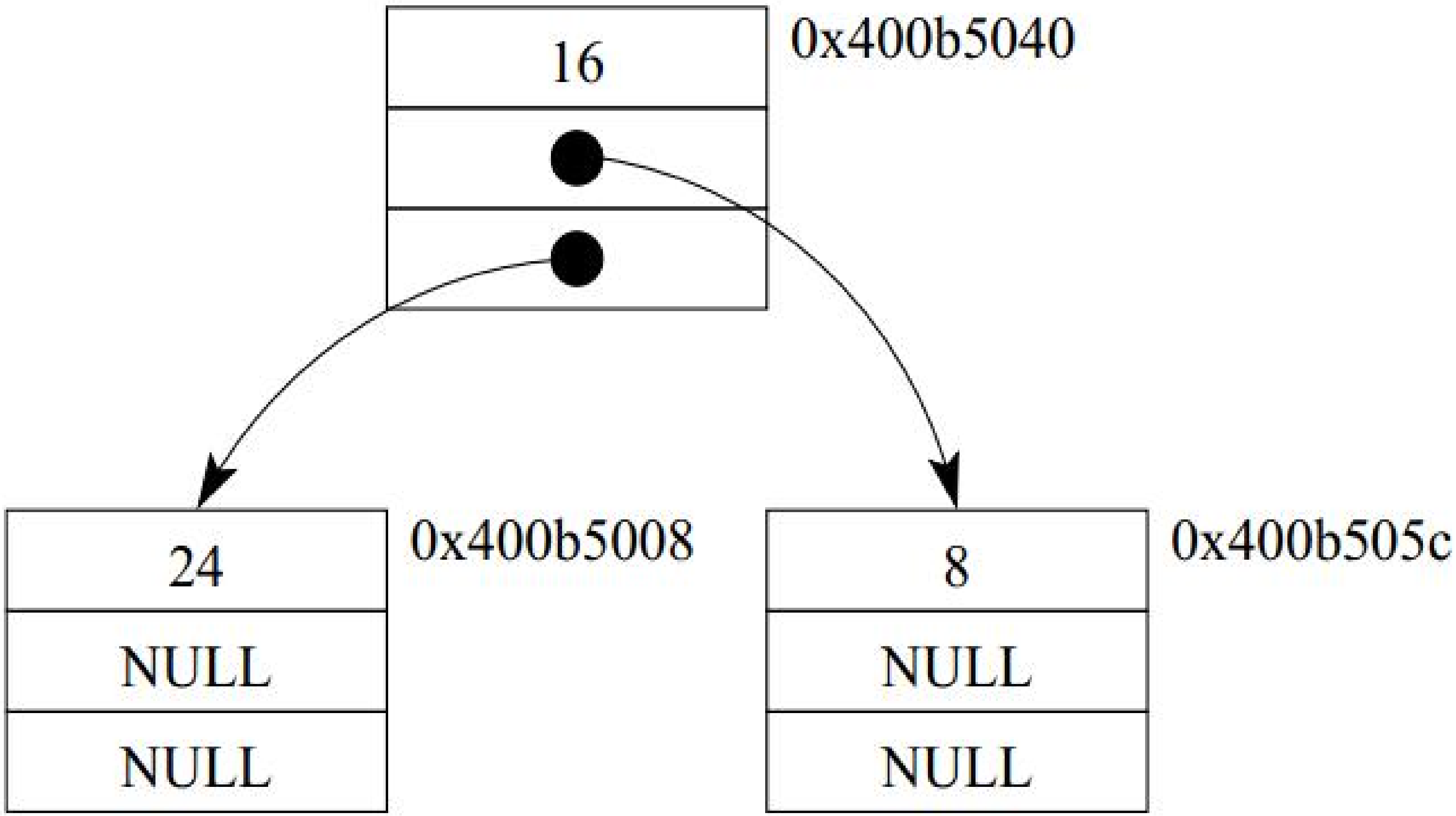
- Data structure of free list
  - Circular Doubly-Linked List





6.1.4.1 Free block organization(8/8)

- Data structure of allocation tree
  - Binary Search Tree



6.1.4 Memory Management Mechanism

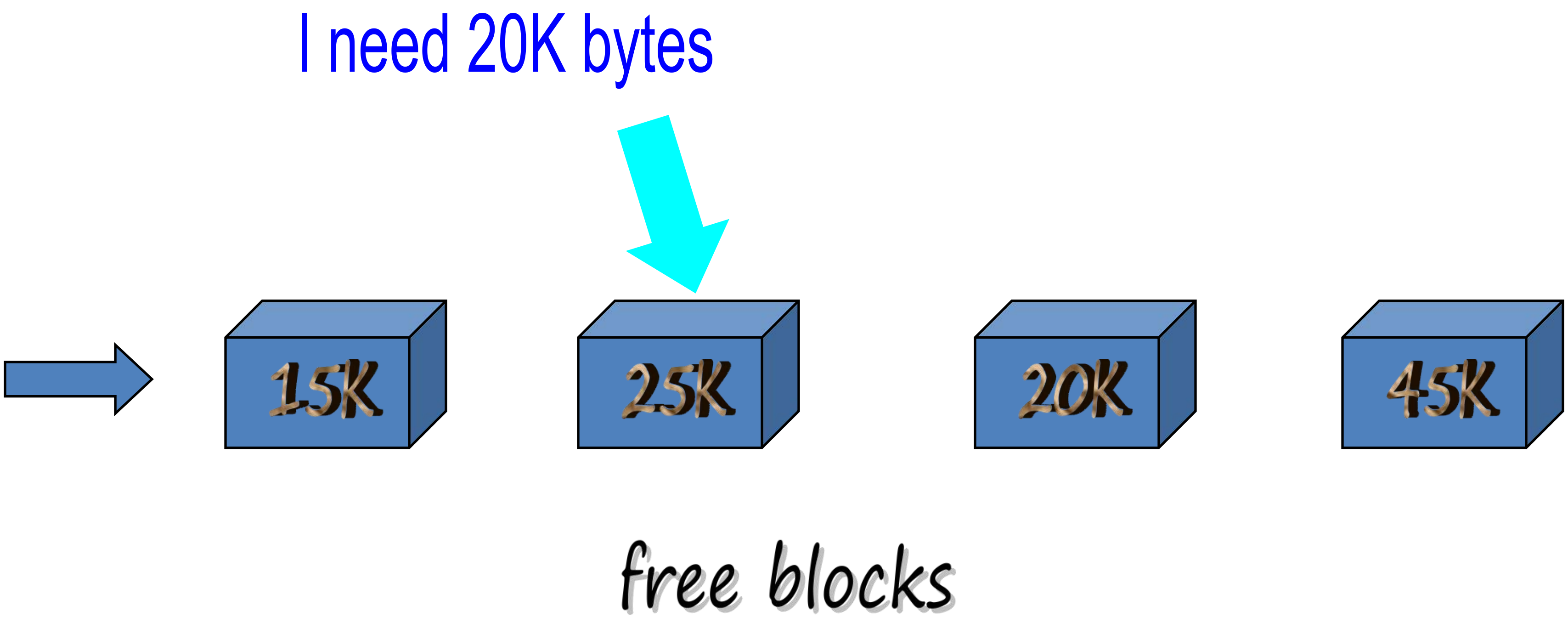
- 6.1.4.1 Free block organization
- <u>6.1.4.2 Placement Policy</u>
- 6.1.4.3 Coalescing

6.1.4.2 Placement Policy(1/6)

- First Fit
- Best Fit
- Worst Fit
- Segregated Fit

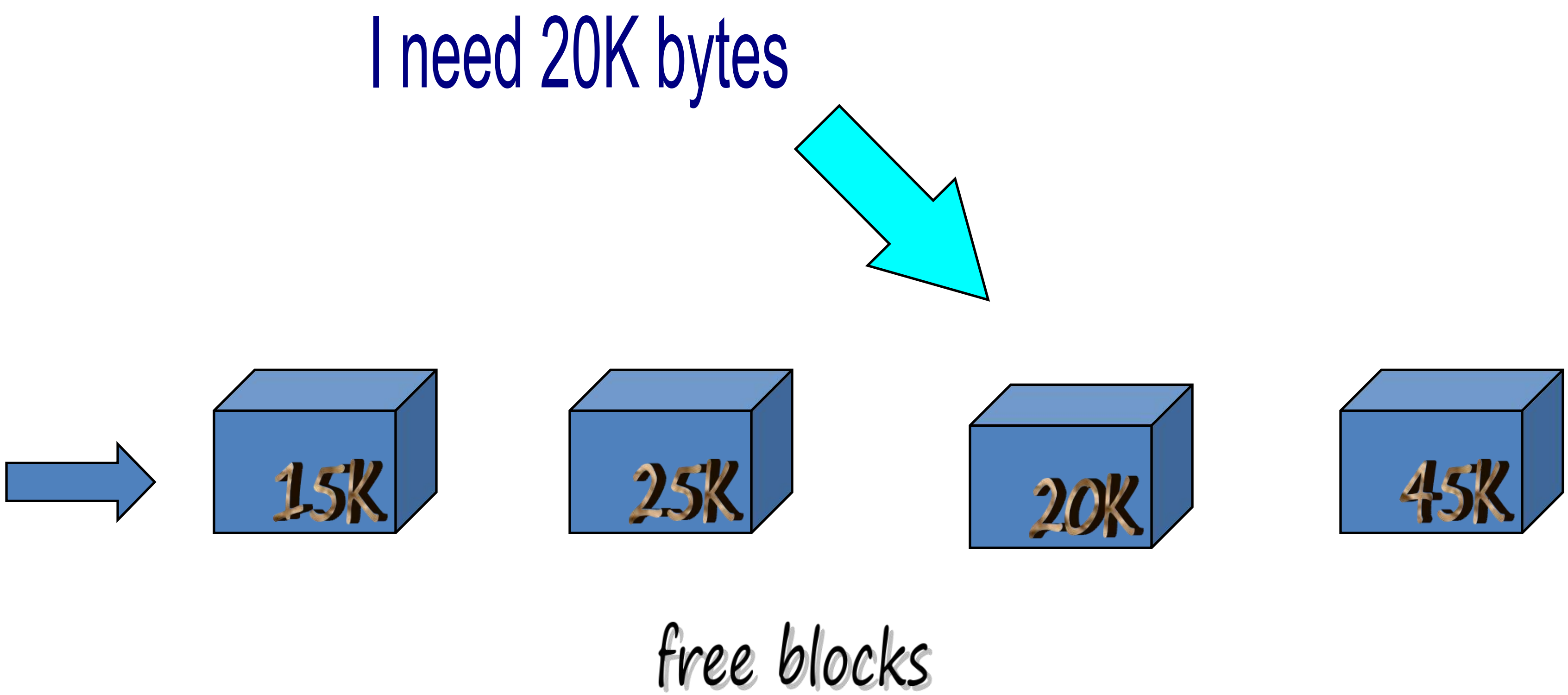
6.1.4.2 Placement Policy(2/6)

- First fit
  - Get the first block in the list that meets the requirement



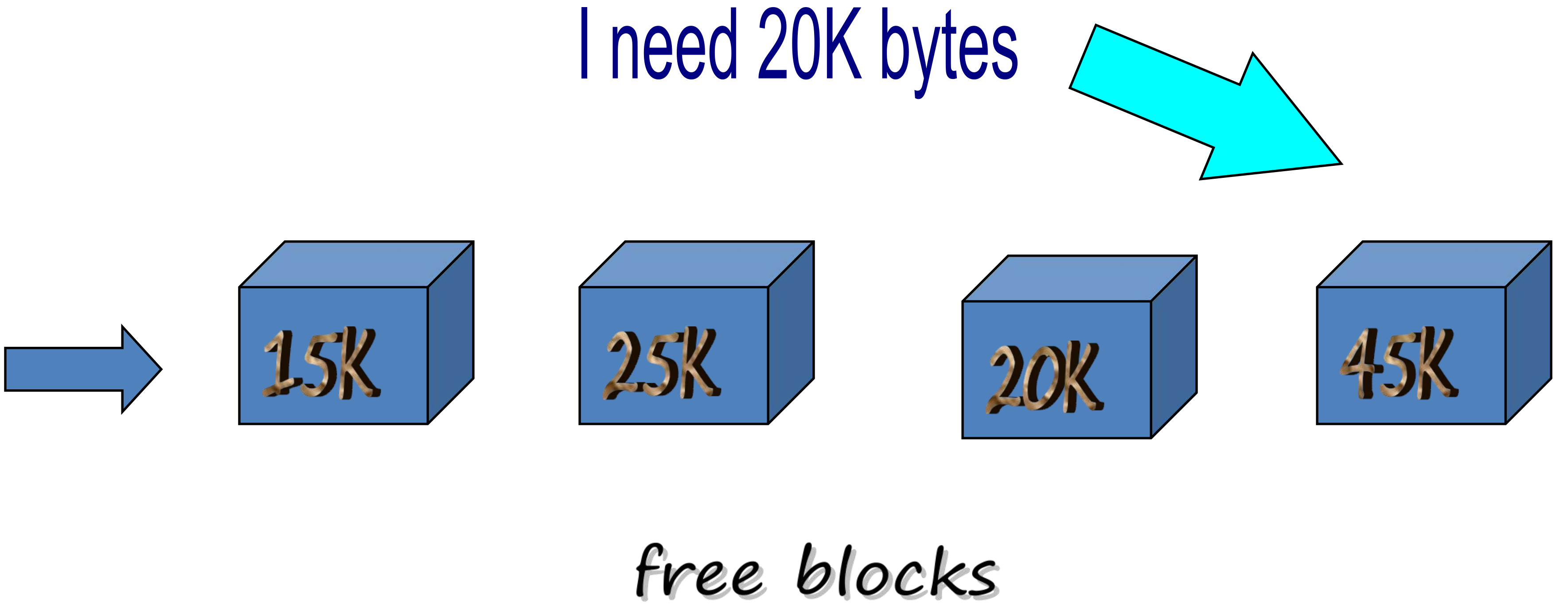
6.1.4.2 Placement Policy(3/6)

- Best fit
  - Get the block that perfectly meets the requirements.



6.1.4.2 Placement Policy(4/6)

- Worst fit
  – Get the largest one in the list.

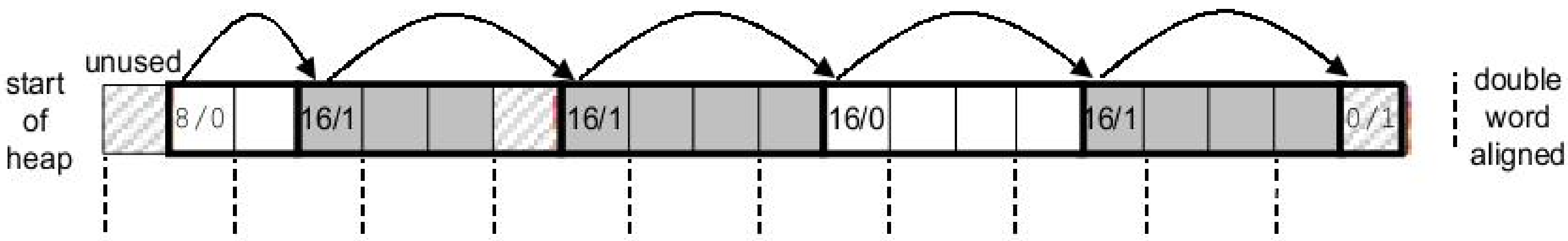


6.1.4.2 Placement Policy(5/6)

- Segregated Fit
  - do a first-fit search of the appropriate free list for a block that fits.
  - If we find one, then we split it and insert the fragment in the appropriate free list.
  - If we cannot find a block that fits, then we search the free list for the next larger size class.

6.1.4.2 Placement Policy(6/6)

- Splitting Free Blocks



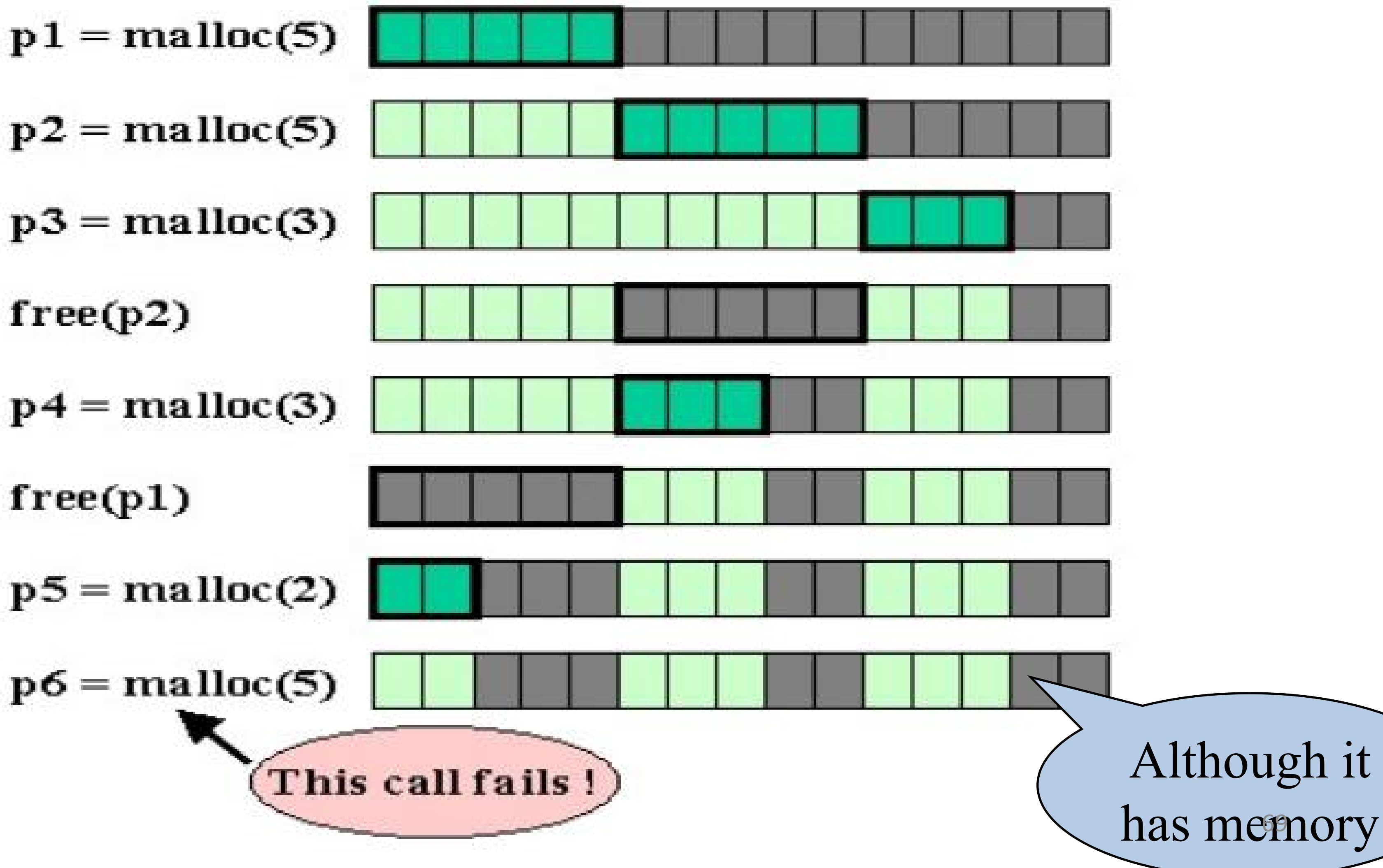
Splitting a free block to satisfy a three-word allocation request

6.1.4 Memory Management Mechanism

- 6.1.4.1 Free block organization
- 6.1.4.2 Placement Policy
- <u>6.1.4.3 Coalescing</u>

6.1.4.3 Coalescing(1/6)

- Fragmentation



6.1.4.3 Coalescing(2/6)

- Getting Additional Heap Memory
  - Coalescing free blocks
    - Immediate coalescing
    - Deferred coalescing (推迟合并)
      - fast allocators often opt for some form of deferred coalescing.
  - Asks the kernel for additional heap memory

6.1.4.3 Coalescing(3/6)

- Deallocation
  - Coalesce: To combine two or more nodes into one.

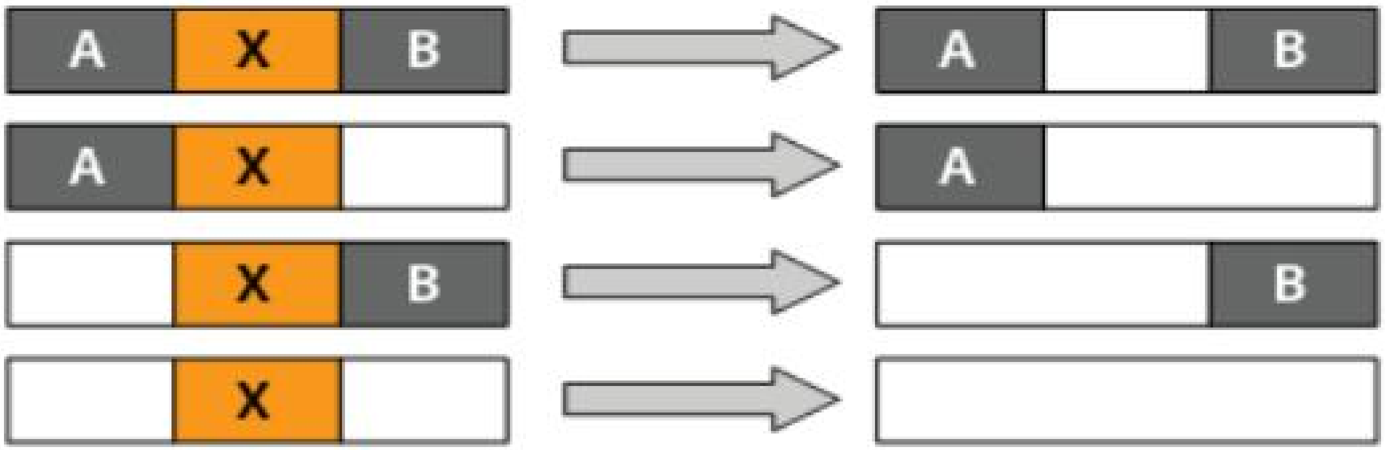
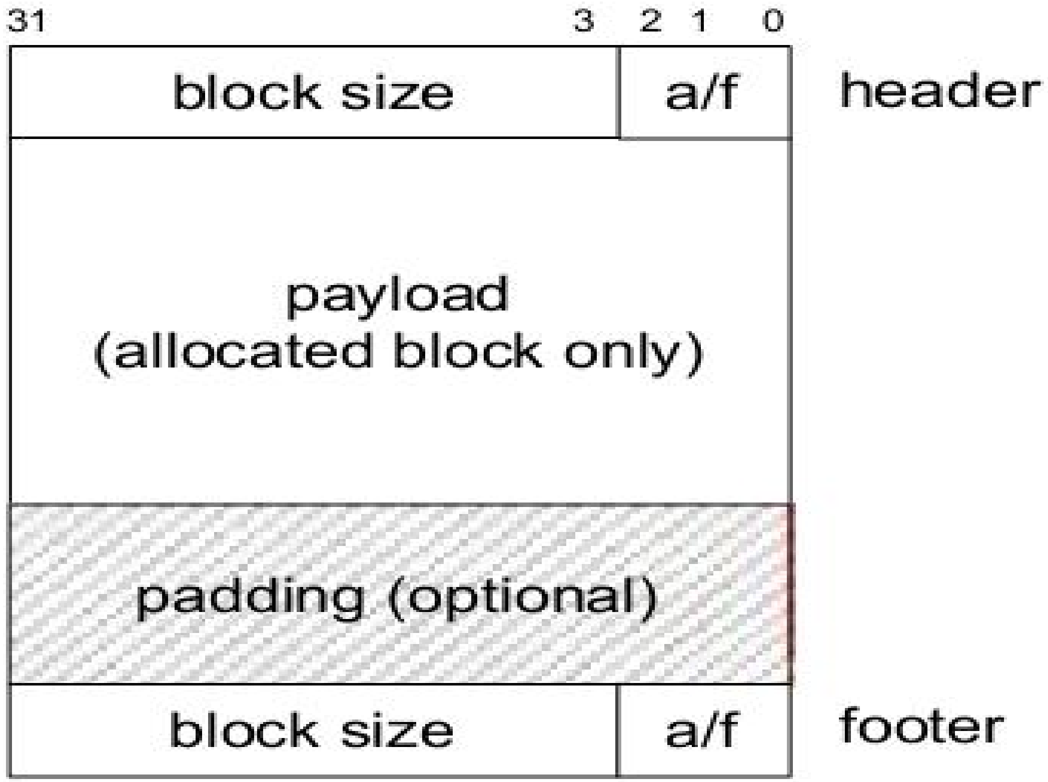


Figure 7.3: Coalescing free nodes on deallocation.

6.1.4.3 Coalescing(4/6)

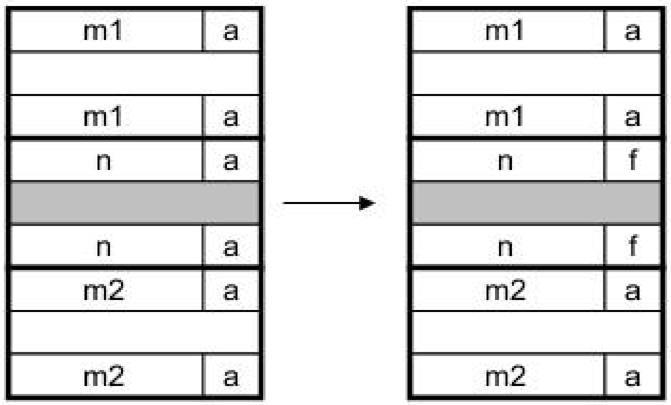
- Coalescing with Boundary Tags



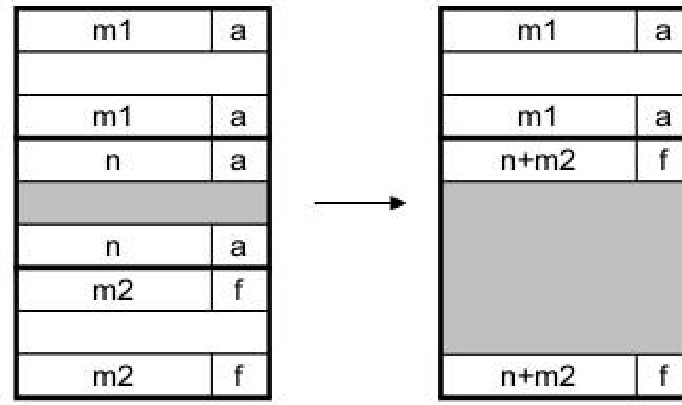
Format of heap block that uses a boundary tag.

6.1.4.3 Coalescing(5/6)

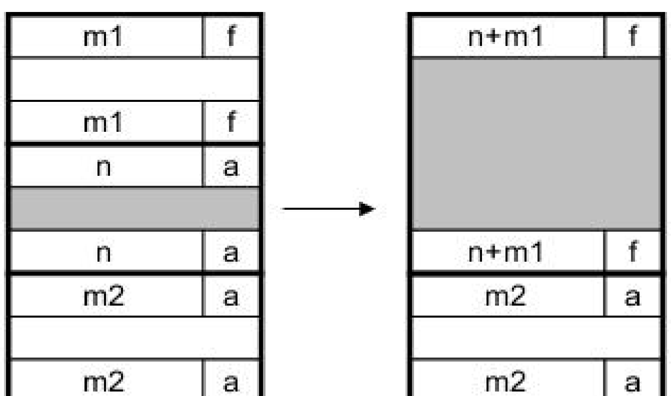
- Coalescing with Boundary Tags



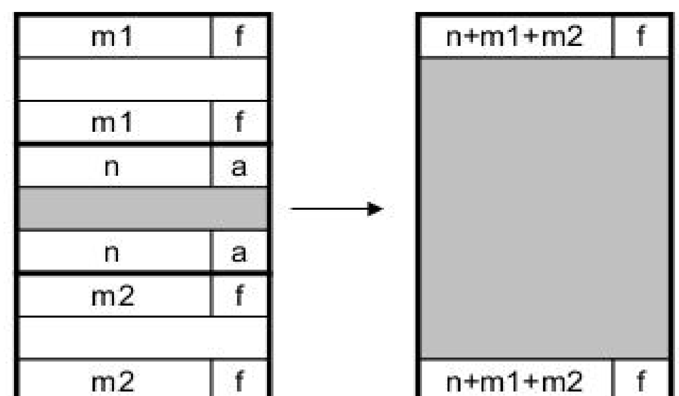
Case 1.



Case 2.



Case 3.



Case 4.

6.1.4.3 Coalescing(6/6)

- Getting Additional Heap Memory
  - Coalescing free blocks
  - Asks the kernel for additional heap memory
    - char *sbrk(int incr);