SSD 6

icarnegie

curriculum powered by Carnegie Mellon

Lecture 5

Memory Layout and Allocation

武汉大学

国际软件学院

Lecture 5 Memory Layout and Allocation

The Contents in Unit 3 cover:

3.1 Several Uses of Memory

3.2 Dynamic allocation

3.3 Heap memory management

3.4 Memory Bugs

Lecture 5 Several Uses of Memory

In this Part:

3.1.1 Static Allocation

3.1.2 Dynamic Allocation

3.1.3 Explicit vs. Implicit

3.1.4 Program Memory Layout

Lecture 5

Several Uses of Memory

What is ***Static Allocation ?***

- The word ***static*** refers to things that happen at compile time and link time when the program is constructed—as opposed to load time or run time when the program is actually started.
- The variable will exist at a fixed address throughout the execution of the program.

Lecture 5 Several Uses of Memory

What is ***Static Allocation ?***

```
// a statically allocated
variableint int my_var[128];
my_fn(int x)
{
    for (int i = 0; i < 128; i++)
        my_var[i] = 0;
}
```

See more at 3.1.1a ……

In practice, the compiler does not actually know the exact address for the variable. Instead, it uses a symbolic name for the variable, and in a later step, the linker will alter the generated code so that all memory addresses are correct.

Lecture 5 Several Uses of Memory

Static Allocation vs. C++ static Declarations

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

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

Lecture 5

Several Uses of Memory

Static Allocation vs. C++ static Declarations

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

The variable *my_var* is initialized by *my_fn*. The programmer wants to make sure *my_var* is only initialized once, so the flag *my_var_initialized* is used to remember when initialization has occurred. A test in *my_fn* returns immediately if *my_var* has been initialized.

Lecture 5

Several Uses of Memory

Static Allocation vs. C++ static Declarations

```
//a statically allocated variable
int my_var[128];

static bool my_var_initialized =
false;
int my_fn(int x)
{
    if (my_var_initialized) return;
    my_var_initialized = true;
    for (int i = 0; i < 128; i++)
        my_var[i] = 0;
}
```

The way this code fragment is written ***my_var*** and ***my_fn*** will be visible globally: any function can initialize ***my_var*** by calling ***my_fn***, and any function can declare and access ***my_var***. However, functions declared in other **FILES** cannot access ***my_var_initialized*** because it is declared with static.

Lecture 5 Several Uses of Memory

Static Allocation vs. C++ static Declarations

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

- If a variable within a function is declared with static, the variable is statically allocated. It is only visible within the function, but the variable is not recreated each time the function is called.
- This is useful if you want to have a value persist from function call to function call, but you want to hide the variable from other functions.
- Here *my_var_initialized* is only visible to *my_fn*

Lecture 5

Several Uses of Memory

Conclusions to *C++ static Declarations*

- Files vs. Modules
- C++ vs. Java
- Global, Static vs. Public, Private
- C programmers use the static attribute to hide variable and function declarations inside modules, much as you would use public and private declarations in Java and C++. C source files play the role of modules.
- Any global variable or function declared with the static attribute is private to that module. Similarly, any global variable or function declared without the static attribute is public, and can be accessed by any other module.
- It is good programming practice to protect your variables and functions with the static attribute wherever possible.

Lecture 5

Several Uses of Memory

More……

```
1  int f()
2  {
3      static int x = 0;
4      return x;
5  }
6
7  int g()
8  {
9      static int x = 1;
10     return x;
11 }
```

- Interestingly, local procedure variables that are defined with the C static attribute are not managed on the stack

- To understand, see more on SYMBOLS AND SYMBOL TABLES on *LINKING* strategies

Lecture 5

Several Uses of Memory

Precautions For Static Allocation

- Statically allocated data uses memory for the lifetime of the program
- It must be of a fixed size
- Not to allocate memory statically but to wait until run-time allocated (If you know the size)
- limitation on static data depending on systems

Lecture 5 Several Uses of Memory

Precautions For Static Allocation

- **SOLUTION:**

Dynamic Allocation !!

Lecture 5

Dynamic Allocation

Good reasons to use D.A. of variables :

- Naming gets to be a problem
- Programs do not always know how much storage is required until run time
- Static allocation reserves memory for the duration of the program
- Recursive functions and reentrant functions are important and useful

Lecture 5

Dynamic Allocation

Stack Allocation :

- Stacks support recursion and dynamic allocation very efficiently
- When a function is called:
  - Stack holds parameter values, local variables, and the address of the calling function.
- When a function returns:
  - stack space is reclaimed for reuse

Lecture 5 Dynamic Allocation

Stack Allocation :

- foo() calls bar()
- bar() calls baz(0)
- baz(0) → baz(1)
- ……

```
int foo()
{
    int b;
    b = bar();
    return b;
}
```

```
int bar()
{
    int b = 0;
    b = baz(b);
    return b;
}
```

```
int baz(int b)
{
    if (b < 1) return baz(b + 1);
    else return b;
}
```

Lecture 5

Dynamic Allocation

Stack Allocation :

- Variables in different functions can have the same name but still represent different variables.
- Each instance of a recursive function can have its own set of private variables.
- Recursive functions can create arbitrarily many instances of functions.

Lecture 5

Dynamic Allocation

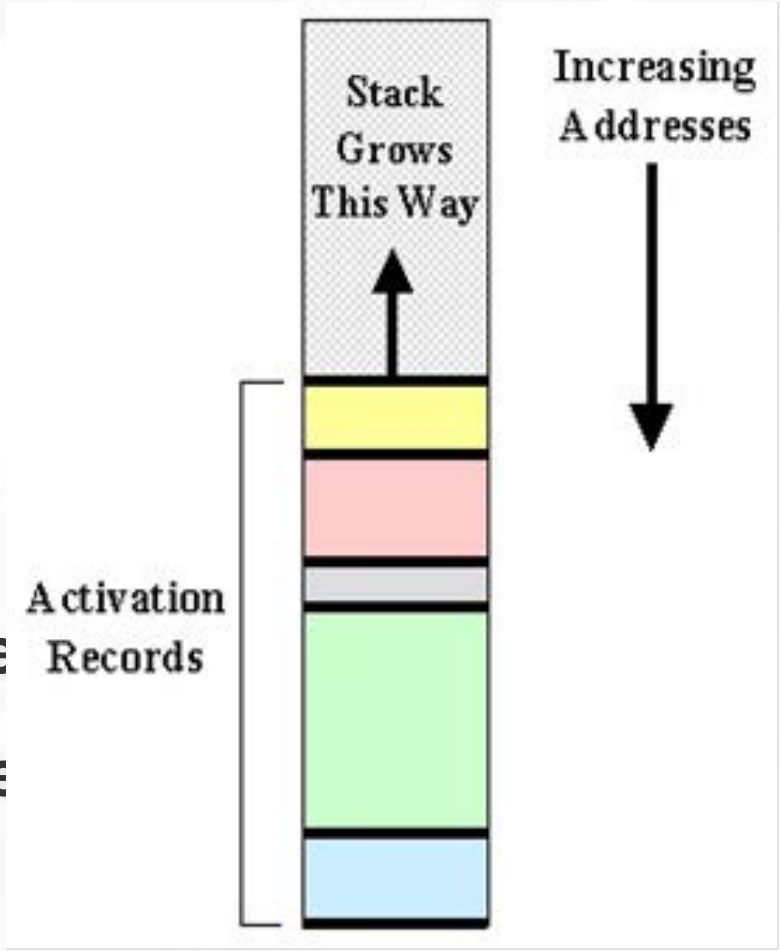
A Function Call Using the Stack :

- **Basic concepts:**
  - ***Activation Record* or *Stack Frame***
  - **TOS=Top of Stack**
  - **Frame=Activation Record Base**
  - **PC=Program Counter**

Lecture 5 Dynamic Allocation

A Function Call Using the Stack :

- Keep in mind:
  - On Pentium processors, the stack "grows" toward lower memory addresses
  - In VC++ debugger, "low" addresses will be at the top, and "high" addresses will be at the bottom of the display



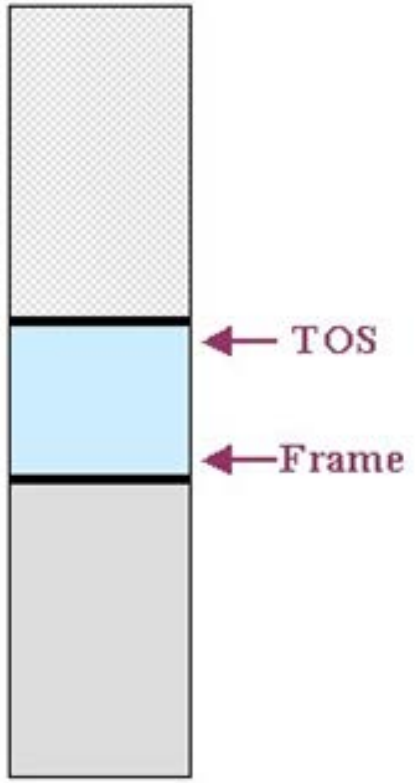
Lecture 5 Dynamic Allocation

A Function Call Using the Stack :

■ 1

TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

```
bar:
   ...
   b = baz(b)
   push b   <- PC
   call baz
   pop b
   ...
```



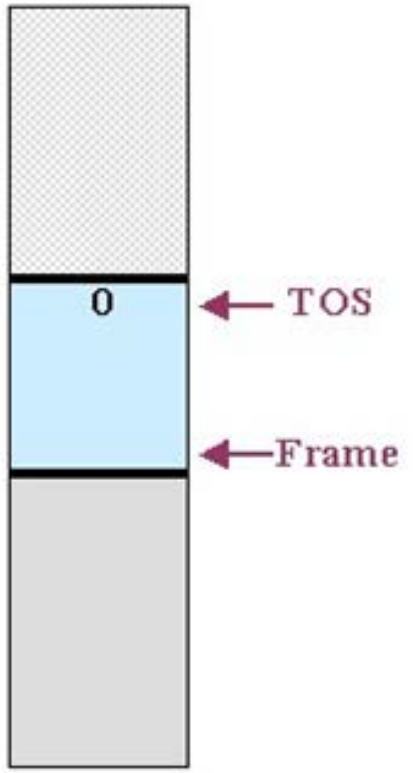
Lecture 5 Dynamic Allocation

A Function Call Using the Stack :

■ 2

TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

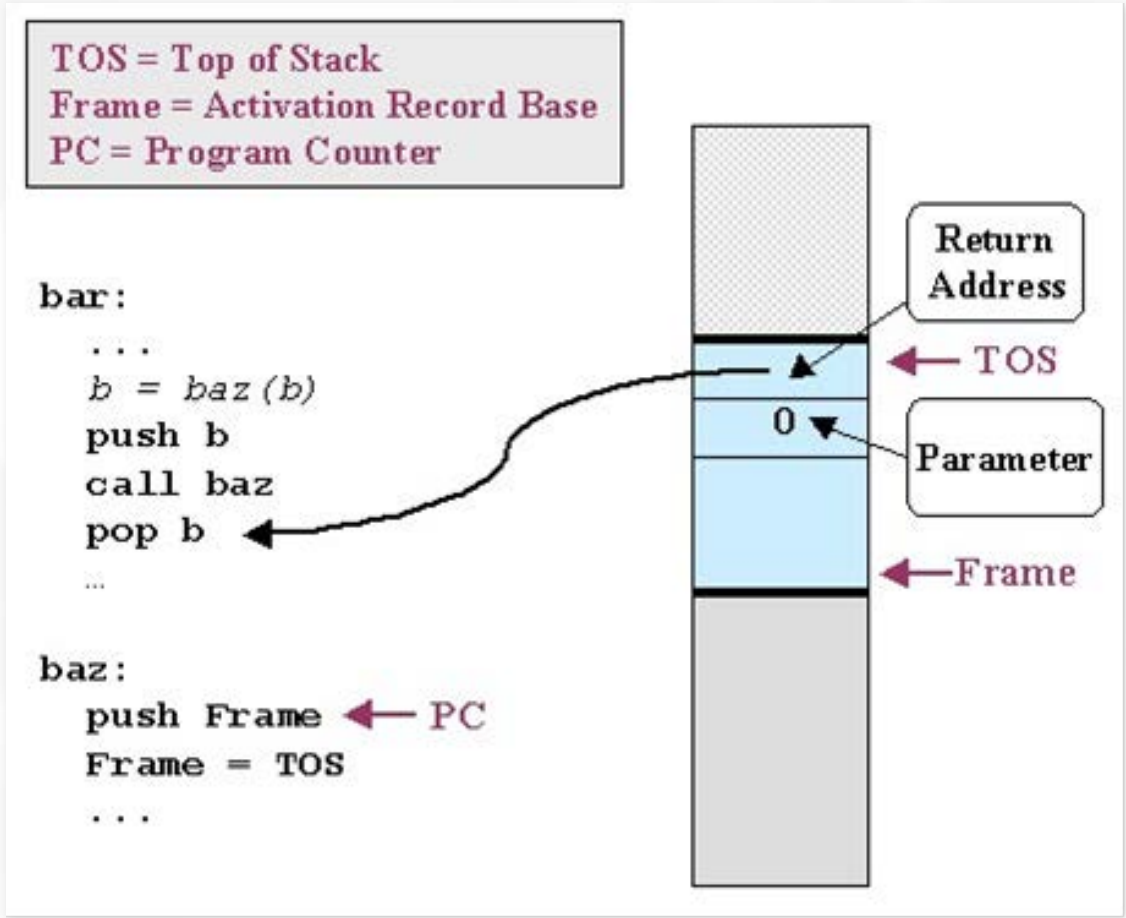
```
bar:
   ...
   b = baz(b)
   push b
   call baz   <- PC
   pop b
   ...
```



Lecture 5 Dynamic Allocation

A Function Call Using the Stack :

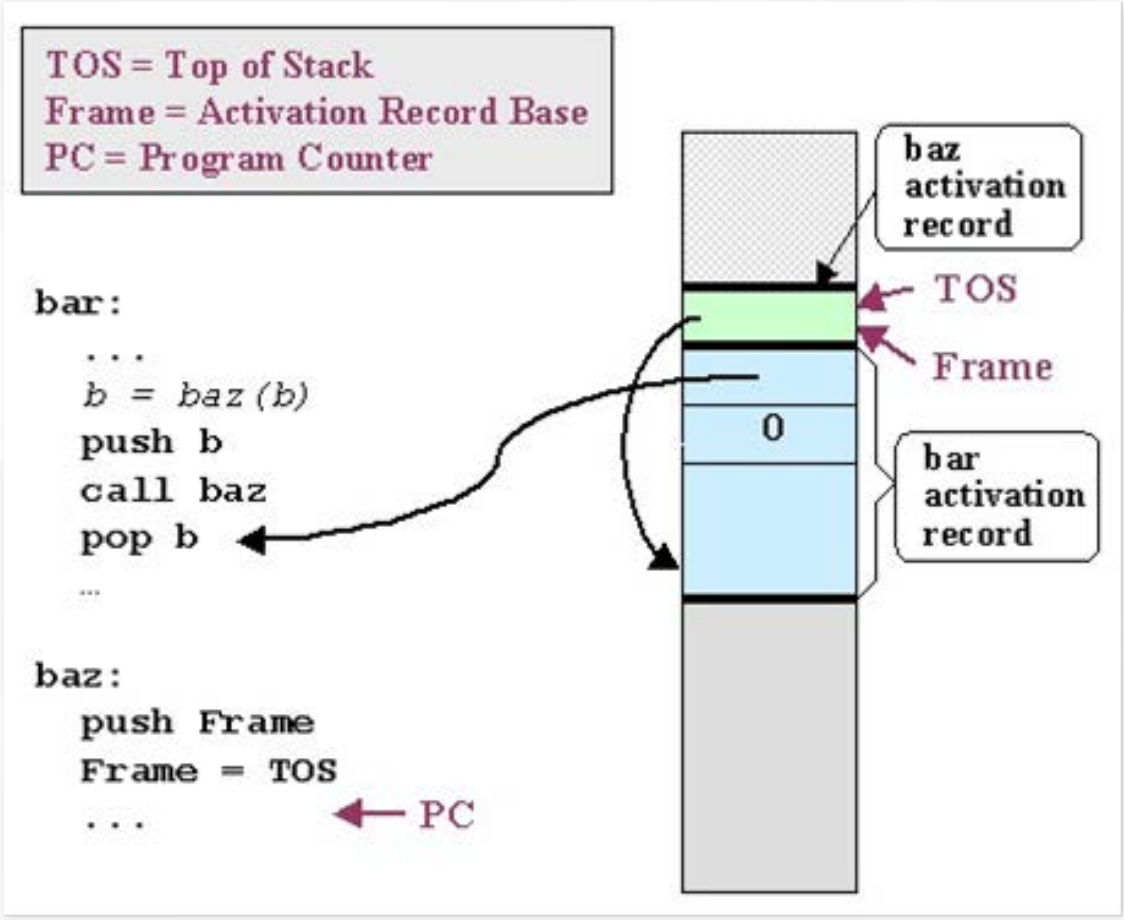
■ 3



Lecture 5 Dynamic Allocation

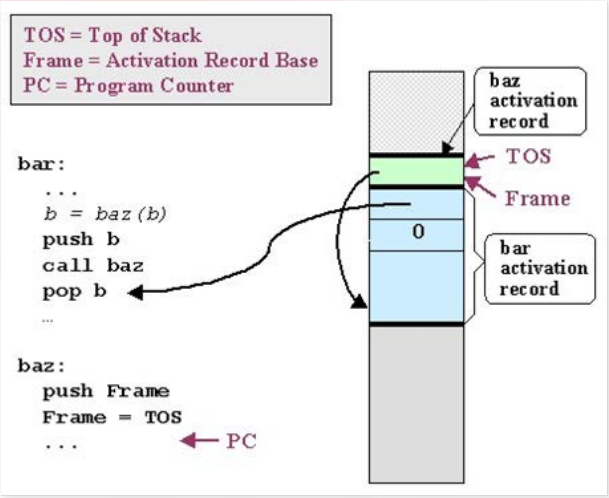
A Function Call Using the Stack :

■ 4



Lecture 5 Dynamic Allocation

A Function Call Using the Stack :



- **Conclusions:**
  - Parameters (In the stack)
  - The return address. (to restore the PC)
  - A pointer to the caller's activation record (to restore the Frame pointer)
  - Saved machine registers (to make fast)
  - Local variables

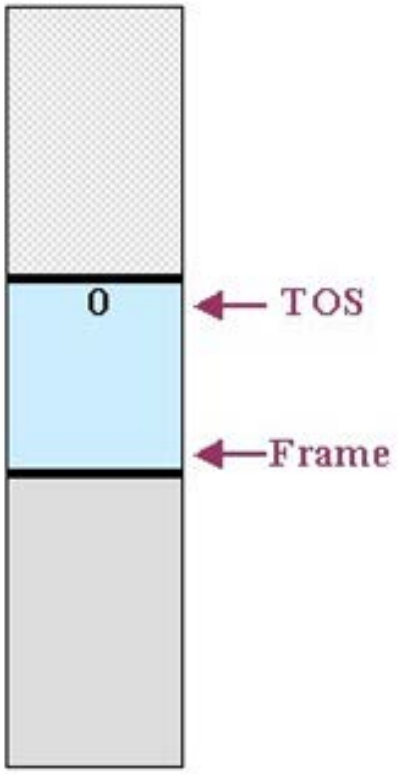
Lecture 5 Dynamic Allocation

A Function Call Using the Stack :

- Identical to function calls
- Everything runs in reverse
- Registers: bridging the caller & functions

TOS = Top of Stack
Frame = Activation Record Base
PC = Program Counter

```
bar:
   ...
   b = baz(b)
   push b
   call baz
   pop b  <- PC
   ...
```



Lecture 5 Dynamic Allocation

Stack Allocation For Local Variables :

- **A code example**

```
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;
        }
// Did loop complete? If so, we found a match.
        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"));
    getchar();
}
```

See 3_1_2b.dsw

Lecture 5 Dynamic Allocation

- A code example

Call Stack

```
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;
        }
        // Did loop complete? If so, we found
```

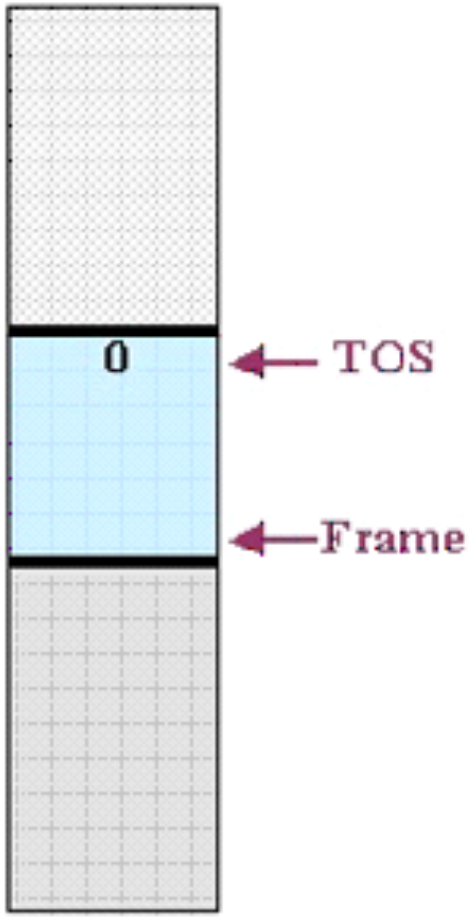
Call Stack

```
find(char * 0x00420fc8 `string', char * 0x00
main() line 25 + 15 bytes
mainCRTStartup() line 206 + 25 bytes
KERNEL32! 7c816d4f()
```

Lecture 5 Dynamic Allocation

- A code example

View/Debugger Windows/Registers



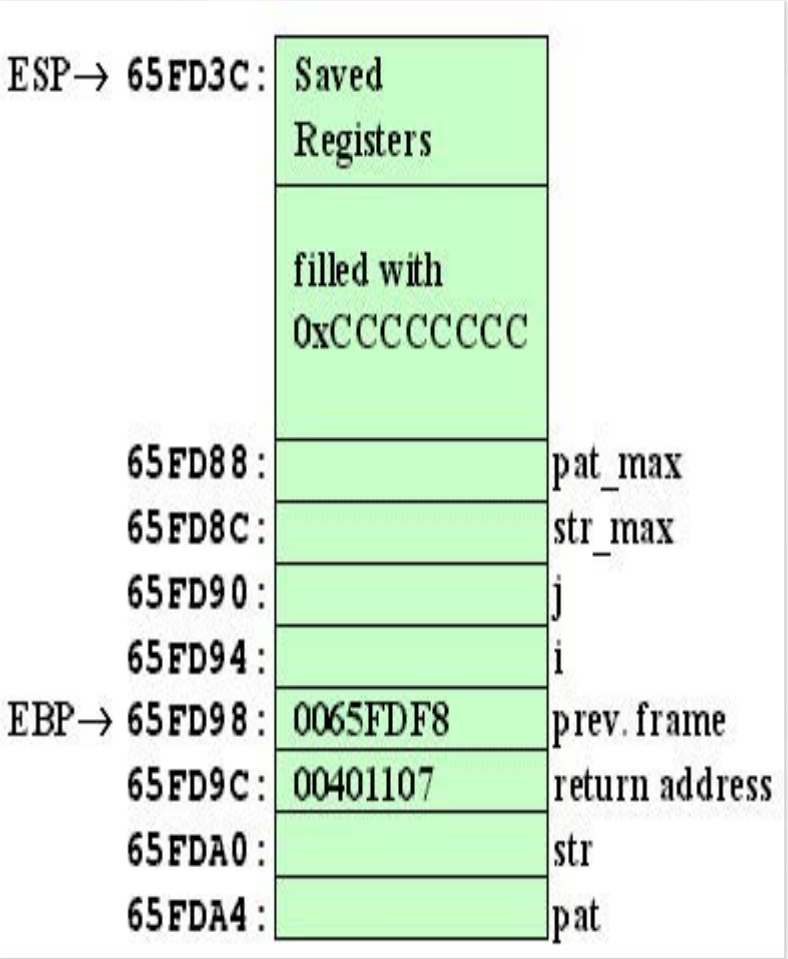
Registers

```
EAX = 0000000C EBX = 7FFDC000
ECX = 00420FC8 EDX = 7EFF7372
ESI = 00000000 EDI = 0012FF20
EIP = 00401059 ESP = 0012FEC4
EBP = 0012FF20 EFL = 00000206 CS = 001B
DS = 0023 ES = 0023 SS = 0023 FS = 003B
GS = 0000 OV=0 UP=0 EI=1 PL=0 ZR=0 AC=0
PE=1 CY=0
```

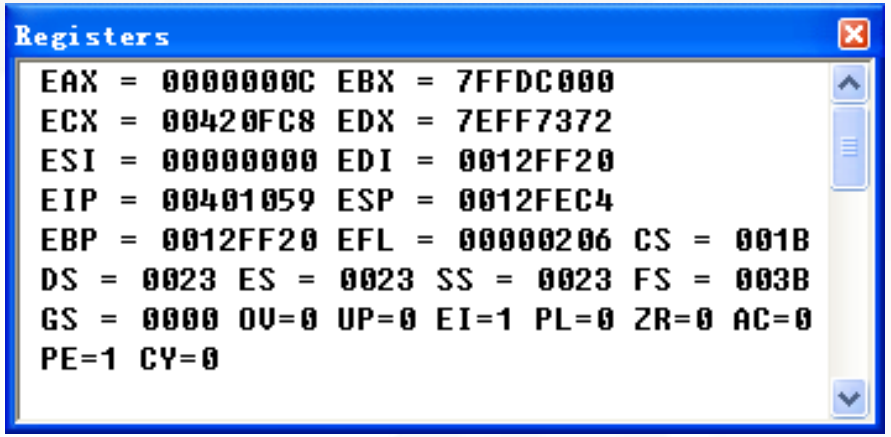
- TOS=ESP
- Frame=EBP

Lecture 5 Dynamic Allocation

- A code example



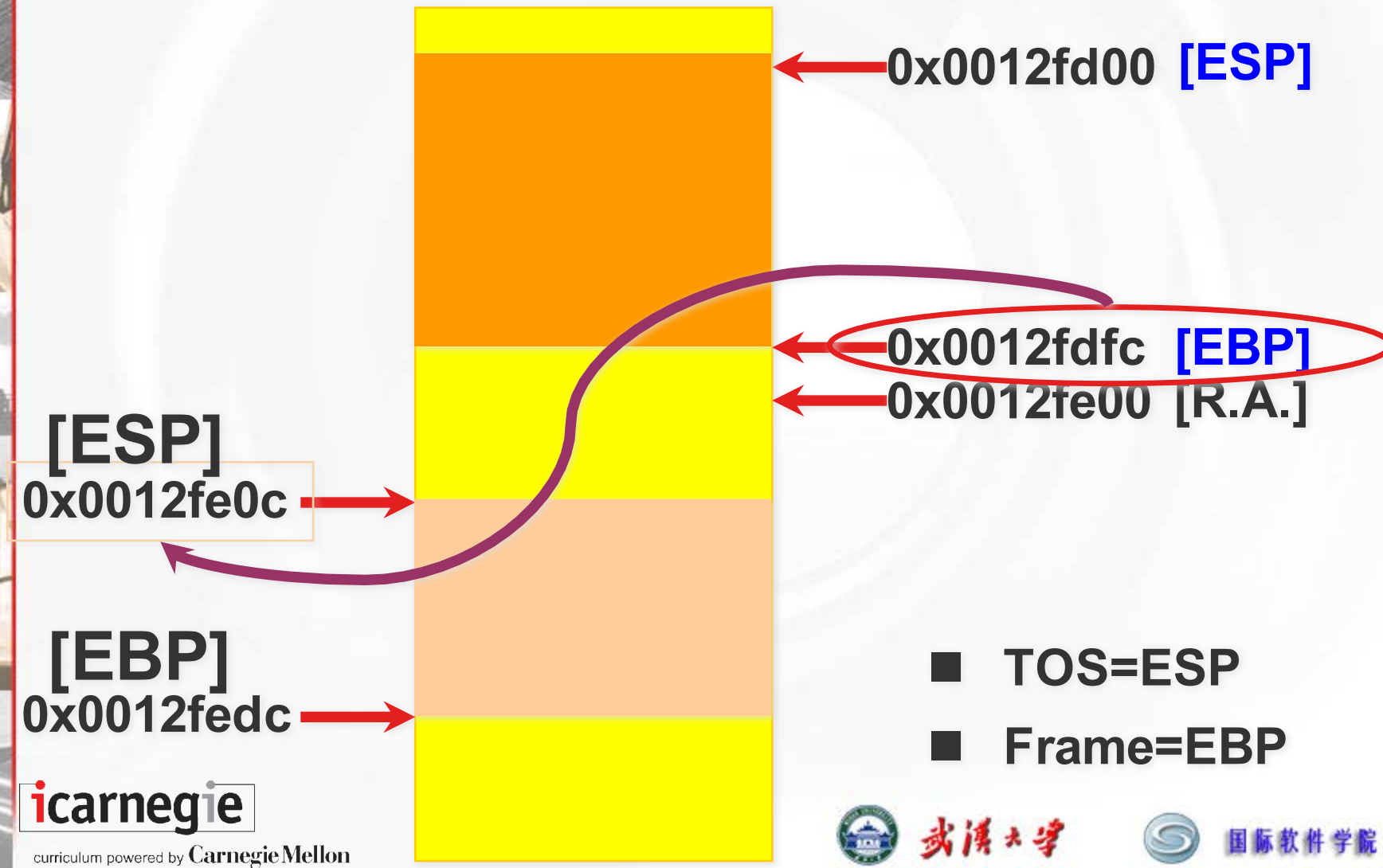
View/Debugger Windows/Registers



- TOS=ESP
- Frame=EBP

Lecture 5 Dynamic Allocation

- A code example



- TOS=ESP
- Frame=EBP

Lecture 5

Dynamic Allocation

- setjmp(jmp_buf j) //first load
- longjmp(jmp_buf j, int i) //destroy buf, and back
- Compare with goto
- catch/throw

Lecture 5

Dynamic Allocation

Heap (Explicit) Allocation :

- *To allocate memory that survives beyond the duration of a function call*
- *Never return the address of a local variable* at the stack
- *Heap* memory: an alternative to stack memory
  - Allocate memory: malloc or new
  - Return memory: free or delete.
- Memory on the heap is always represented and accessed by a pointer

Lecture 5 Dynamic Allocation

Common memory bugs :

- ***Forget to free memory***
- ***Memory leak***
- ***Dangling pointer problem***
- ***C++ never tries to debug the above***

Lecture 5

Dynamic Allocation

Explicit vs. Implicit :

- Free memory automatically or NOT?
  - *Java, Python, Perl, and Lisp*
- C programmer is free to coerce pointers from one type to another
- *Explicit memory management:*
  - *C and C++ leave everything up to the programmer*
- *Implicit memory management*:
  - *Run-time system helps to free memory*

Lecture 5 Dynamic Allocation

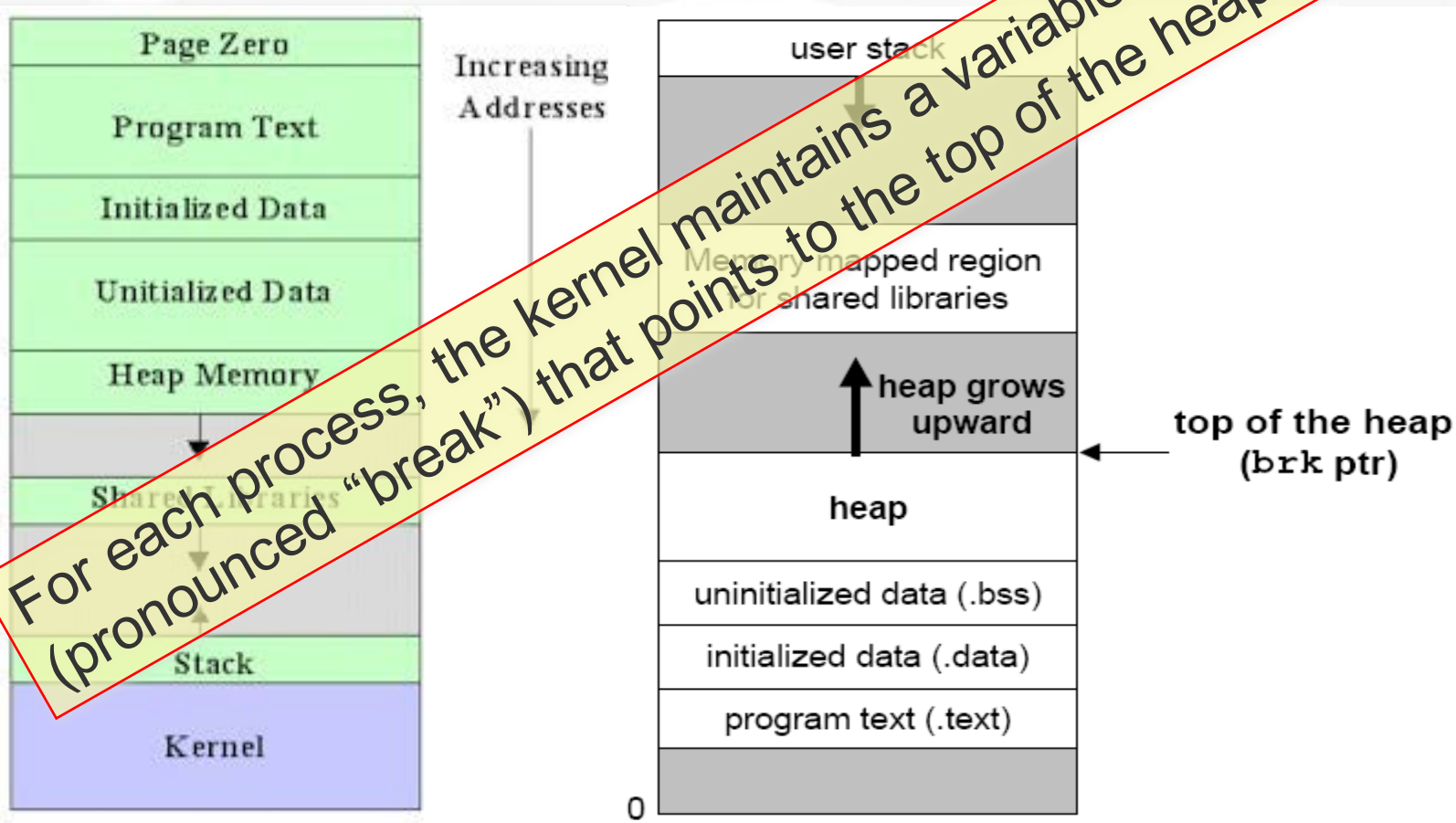
Program Memory Layout :

- **Memory is just a big array of bytes, but programs partition this pool of bytes into different areas.**
  - Program Text
  - Initialized Data
  - Uninitialized Data
  - Heap Memory
  - Shared Libraries
  - The Stack
  - The Kernel
  - Page Zero

} A program

Lecture 5 Dynamic Allocation

Program Memory Layout :



For each process, the kernel maintains a variable **brk** (pronounced "break") that points to the top of the heap.

Lecture 5

Memory Bugs

In this Part:

3.2.1 Review of Pointers in C

3.2.2 Making and Using Bad References

3.2.3 Overwriting Memory

3.2.4 Twice free

3.2.5 Memory Leaks

3.2.6 Exterminating Memory Bugs

3.2.7 Garbage Collection

Lecture 5 Next Time.....

- Read:
  - Continue Unit 3 (the second half)
- Do:
  - Multiple-Choice Quizzes in Unit 3

Lecture 5 In the Lab……

- Excise 2
- Excise 3

SSD
6

icarnegie
curriculum powered by Carnegie Mellon

Bonus for

Malloc

&

Free

武漢大學

国际软件学院

Bonus

Malloc

Explicit Allocators:

- Malloc()
- Calloc()
- Realloc()
- Sbrk()

Bonus

Malloc

Malloc Function:

```
#include <stdlib.h>

void *malloc(size_t size);
```

- The *malloc* function returns a pointer to a block of memory of at least size bytes that is suitably aligned for any kind of data object that might be contained in the block.
- If *malloc* encounters a problem (e.g., the program requests a block of memory that is larger than the available virtual memory), then it returns **NULL** and sets **errno**. *Malloc* does not initialize the memory it returns

Bonus

Malloc

Sbrk( ) Function:

```
#include <unistd.h>

void *sbrk(int incr);
```

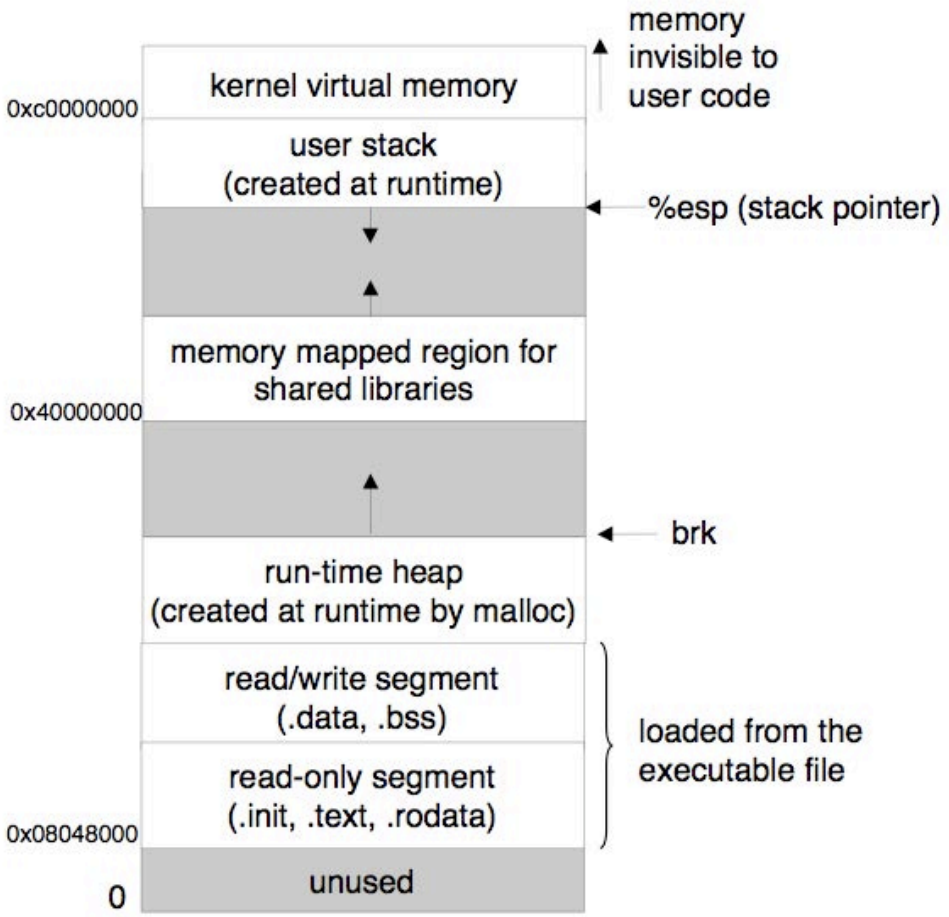
- The sbrk function grows or shrinks the heap by adding incr to the kernel's brk pointer.
- If successful, it returns the old value of brk, otherwise it returns -1 and sets errno to ENOMEM.
- If incr is zero, then sbrk returns the current value of brk.

Bonus

Malloc

Memory image of a *Linux* process

- Programs always start at virtual address `0x08048000`.
- The user stack always starts at virtual address `0xbfffffff`.
- Shared objects are always loaded in the region beginning at virtual address `0x40000000`



Bonus

Malloc

free( ) Function:

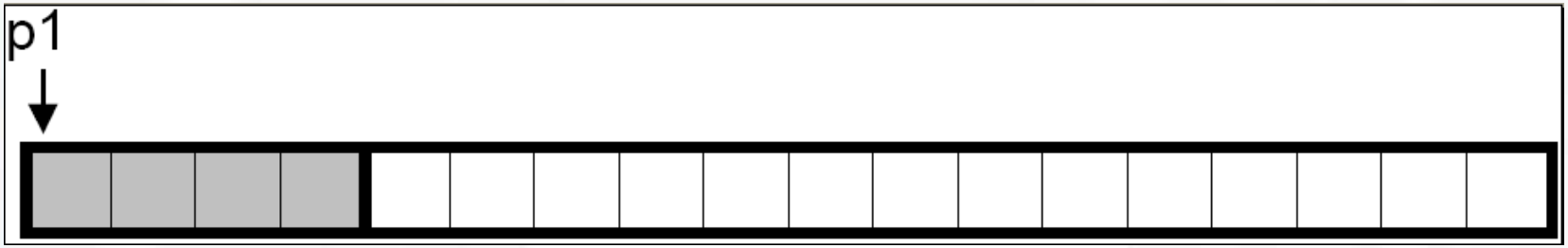
```
#include <stdlib.h>

void free(void *ptr);
```

- The ptr argument must point to the beginning of an allocated block that was obtained from malloc.
- If not, then the behavior of free is undefined.
- Even worse, since it returns nothing, free gives no indication to the application that something is wrong.
- This can produce some baffling run-time errors.

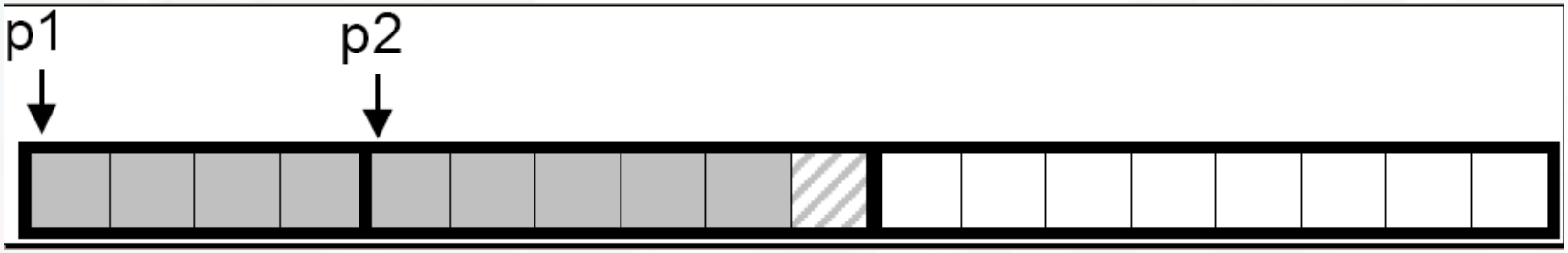
Bonus

Malloc

Allocating & Freeing blocks with malloc():



P1=malloc(4*sizeof(int))

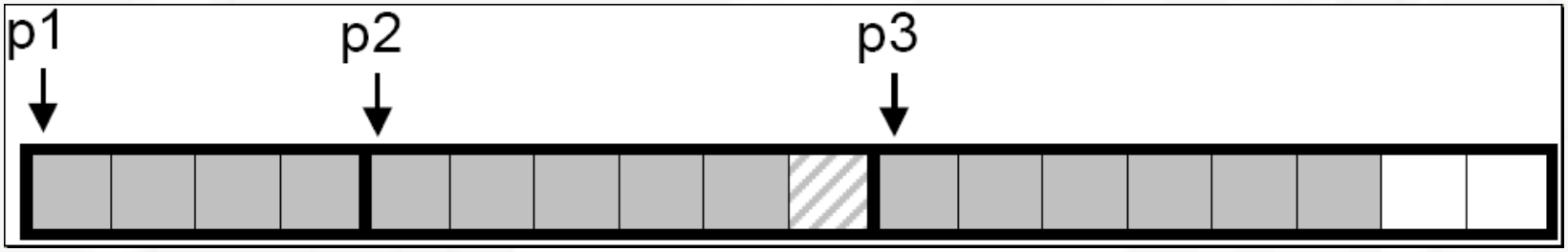


P2=malloc(5*sizeof(int))

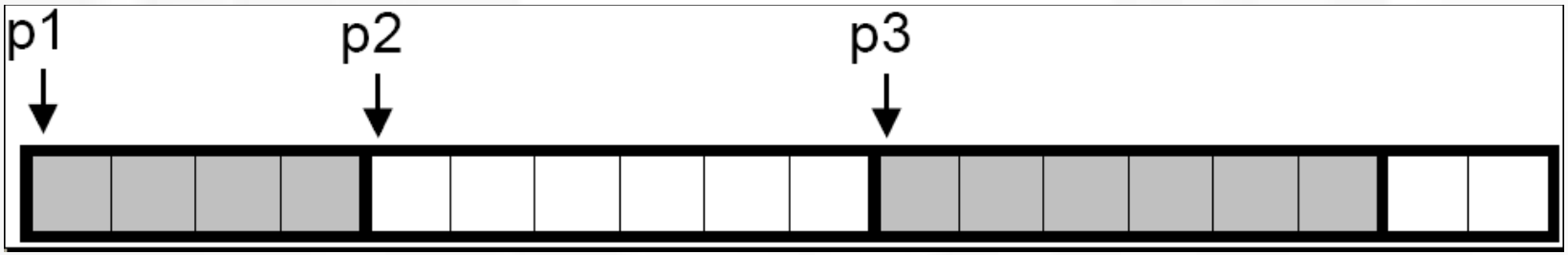
Bonus

Malloc

Allocating & Freeing blocks with malloc():



P3=malloc(6*sizeof(int))

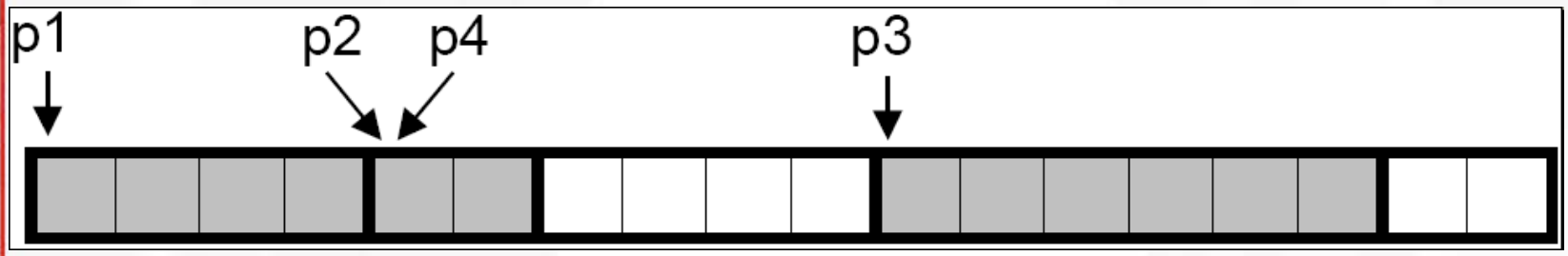


Free(p2)

Bonus

Malloc

Allocating & Freeing blocks with malloc():



P4=malloc(2*sizeof(int))

Bonus

Malloc

Why need Dynamic Memory Allocation?

- They do not know the sizes of certain data structures until the program actually runs
- Use hard-coded sizes for array is a bad idea
- Instead, we use dynamic allocation

Bonus

Malloc

Why need Dynamic Memory Allocation?

```
1  #include "csapp.h"
2  #define MAXN 15213
3
4  int array[MAXN];
5
6  int main()
7  {
8      int i, n;
9
10     scanf("%d", &n);
11     if (n > MAXN)
12         app_error("Input file too big");
13     for (i = 0; i < n; i++)
14         scanf("%d", &array[i]);
15     exit(0);
16 }
```

```
1  #include "csapp.h"
2
3  int main()
4  {
5      int *array, i, n;
6
7      scanf("%d", &n);
8      array = (int *)Malloc(n * sizeof(int));
9      for (i = 0; i < n; i++)
10         scanf("%d", &array[i]);
11     exit(0);
12 }
```

Bonus

Allocator

Allocator Requirements and Goals

- *Handling arbitrary request sequences.* *(No assumptions of ordering of allocate and free requests)*
- *Making immediate responses to requests.* *(No reorder or buffer allowed)*
- *Using only the heap.* *(Make allocator scalable)*
- *Aligning blocks* *(alignment requirement).*
- *Not modifying allocated blocks.* *(No compactions)*

Bonus

Allocator

Allocator Requirements and Goals

- *Goal 1: Maximizing throughput.*

Given some sequence of *n* allocate and free requests:

$$R_0, R_1, ..., R_k, ..., R_{n-1}$$

we would like to maximize an allocator's ***throughput***, which is defined as the number of requests that it completes per unit time. For example, if an allocator completes 500 allocate requests and 500 free requests in 1 second, then its throughput is 1,000 operations per second.

Bonus

Allocator

Allocator Requirements and Goals

- *Goal 2: Maximizing memory utilization.*

If an application requests a block of ***n*** bytes, then the resulting allocated block has a *payload* of ***p*** bytes. After request R_k has completed, let the *aggregate payload*, denoted P_k, be the sum of the payloads of the currently allocated blocks, and let H_k denote the current (monotonically nondecreasing) size of the heap.

Then, over the first ***k*** requests,

$$U_k = \frac{\max_{i \le k} P_i}{H_k}$$

Bonus

Allocator

Allocator Requirements and Goals

- *Goal 1: Maximizing throughput.*
- *Goal 2: Maximizing memory utilization.*

- there is a tension between maximizing throughput and utilization.
- In particular, it is easy to write an allocator that maximizes throughput at the expense of heap utilization.
- One of the interesting challenges in any allocator design is finding an appropriate balance between the two goals.

Bonus

Fragmentation

Fragmentation:

- Internal fragmentation (*when an allocated block is larger than the payload*)
- External fragmentation *(when there is enough aggregate free memory to satisfy an allocate request, but no single free block is large enough to handle the request*)
- External fragmentation is much more difficult to quantify than internal fragmentation, because it depends not only on the pattern of previous requests and the allocator implementation, but also on the pattern of *future* requests.

Bonus

Implementation Issues

How to implement a simple allocator?

First to know:

- The simplest allocator would organize the heap as a large array of bytes and a pointer ***p*** that initially points to the first byte of the array. To allocate size bytes, ***malloc*** would save the current value of ***p*** on the stack, increment ***p*** by size, and return the old value of ***p*** to the caller.
- Free would simply return to the caller without doing anything.
- Since each ***malloc*** and ***free*** execute only a handful of instructions, throughput would be extremely good. However, since the allocator never reuses any blocks, memory utilization would be extremely bad.

Bonus

Implementation Issues

Strategies

- A practical allocator that strikes a better balance between throughput and utilization must consider the following issues:
  - *Free block organization:* How do we keep track of free blocks?
  - *Placement*: How do we choose an appropriate free block in which to place a newly allocated block?
  - *Splitting*: After we place a newly allocated block in some free block, what do we do with the remainder of the free block?
  - *Coalescing*: What do we do with a block that has just been freed?

Bonus

Implementation Issues

Implicit Free Lists

- Any practical allocator needs some data structure that allows it to distinguish block boundaries and to distinguish between allocated and free blocks.
  - In this case, a block consists of a one-word header, the payload, and possibly some additional padding.
  - The header encodes the block size (including the header and any padding) as well as whether the block is allocated or free. If we impose a double-word alignment constraint, then the block size is always a multiple of eight and the three low-order bits of the block size are always zero.

Bonus

Implementation Issues

Implicit Free Lists (cont'd)

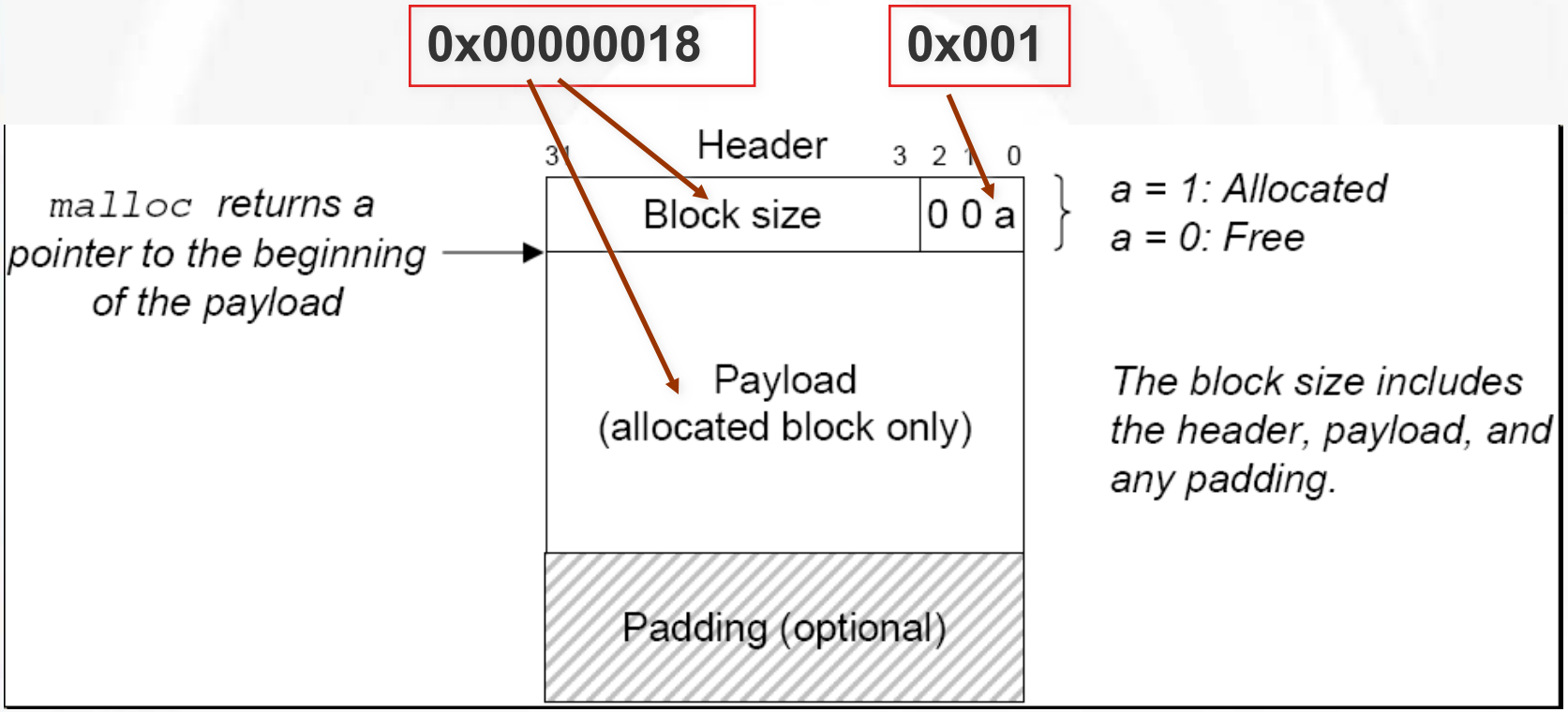
- Any practical allocator needs some data structure that allows it to distinguish block boundaries and to distinguish between allocated and free blocks.

  - Thus, we need to store only the 29 high-order bits of the block size, freeing the remaining three bits to encode other information.
  - In this case, we are using the least significant of these bits (the allocated bit) to indicate whether the block is allocated or free.

Bonus

Implementation Issues

Implicit Free Lists (cont'd)

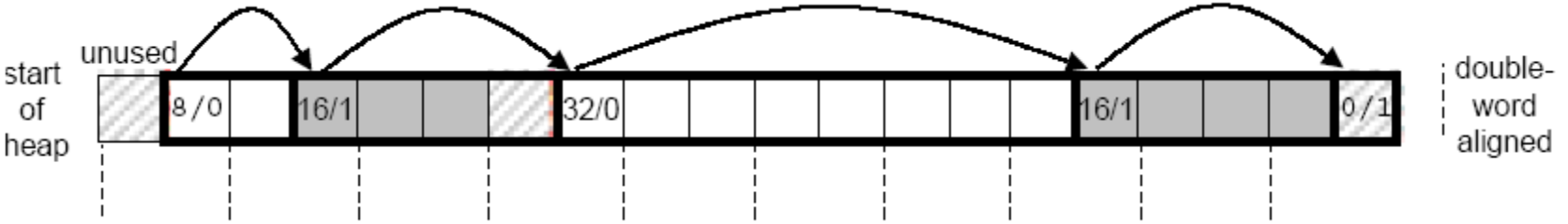


Bonus

Implementation Issues

Implicit Free Lists (cont'd)

Organize the heap as a sequence of contiguous allocated and free blocks

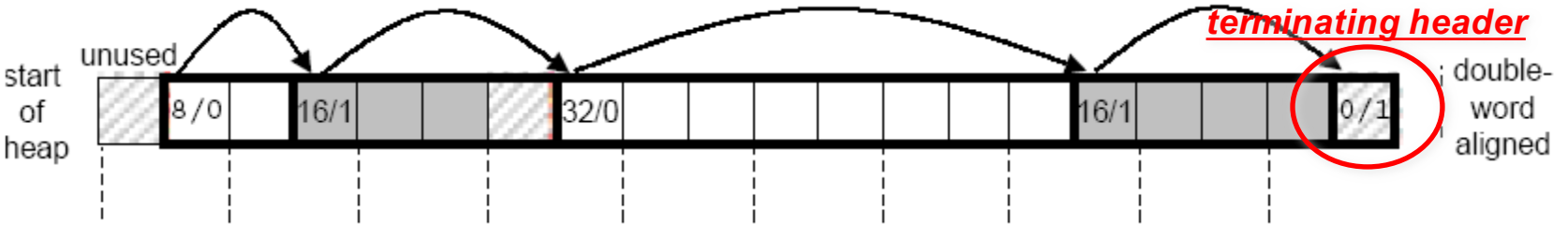


- *Allocated blocks are shaded.*
- *Free blocks are unshaded.*
- *Headers are labeled with (size (bytes)/allocated bit).*

Bonus

Implementation Issues

Implicit Free Lists (cont'd)

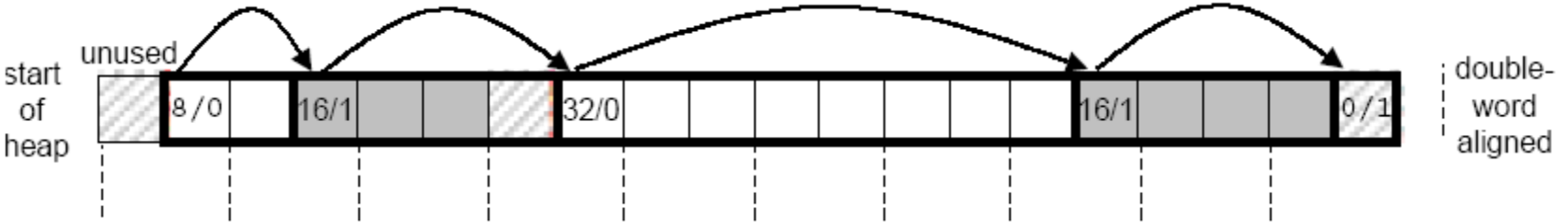


- We call this organization an implicit free list because the free blocks are linked ***implicitly*** by the size fields in the headers.
- The allocator can indirectly ***traverse*** the entire set of free blocks by traversing all of the blocks in the heap.
- We need some kind of specially marked end block, a ***terminating header*** with the allocated bit set and a size of zero, as above.

Bonus

Implementation Issues

Placing Allocated Blocks

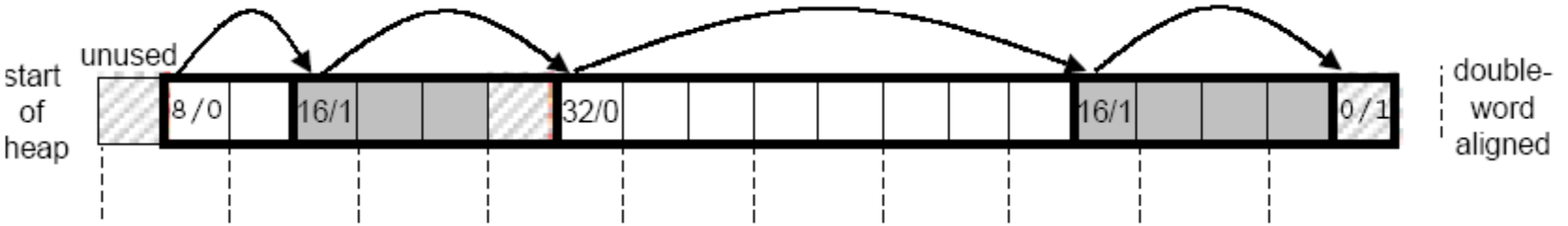


- When an application requests a block of k bytes, the allocator searches the free list for a free block that is large enough to hold the requested block.
- The manner in which the allocator performs this search is determined by the placement policy.
- Some common policies are:
  - first fit,
  - next fit,
  - and best fit.

Bonus

Implementation Issues

Placing Allocated Blocks

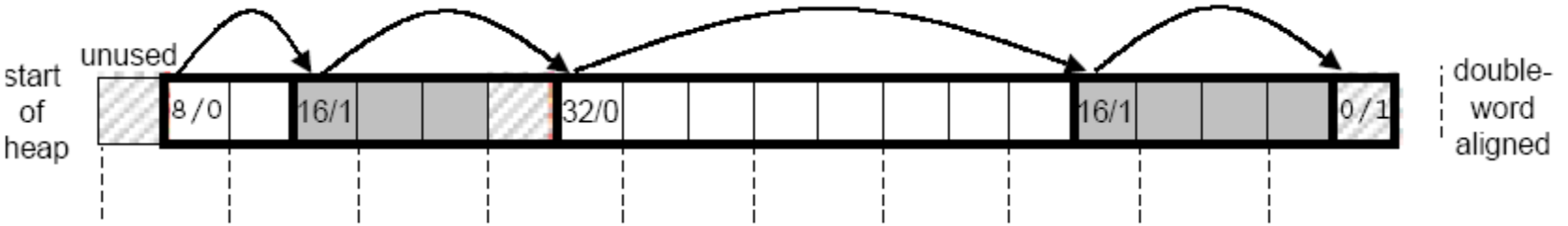


- ***First fit*** searches the free list from the beginning and chooses the first free block that fits.
- ***Next fit*** is similar to first fit, but instead of starting each search at the beginning of the list, it starts each search where the previous search left off.
- ***Best fit*** examines every free block and chooses the free block with the smallest size that fits.

Bonus

Implementation Issues

Splitting Free Blocks

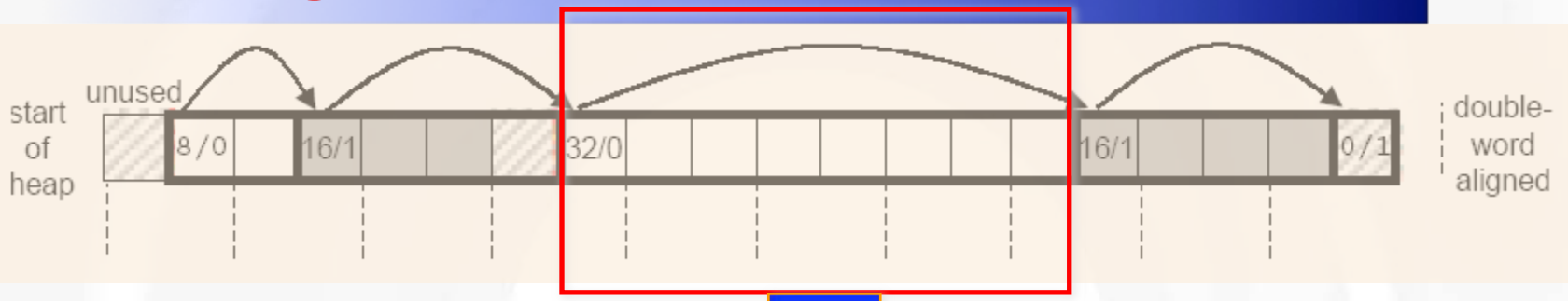


- Once the allocator has located a free block that fits, it must make another policy decision about how much of the free block to allocate.
- One option is to use the entire free block. Although simple and fast, the main disadvantage is that it introduces internal fragmentation.
- If the placement policy tends to produce good fits, then some additional internal fragmentation might be acceptable.

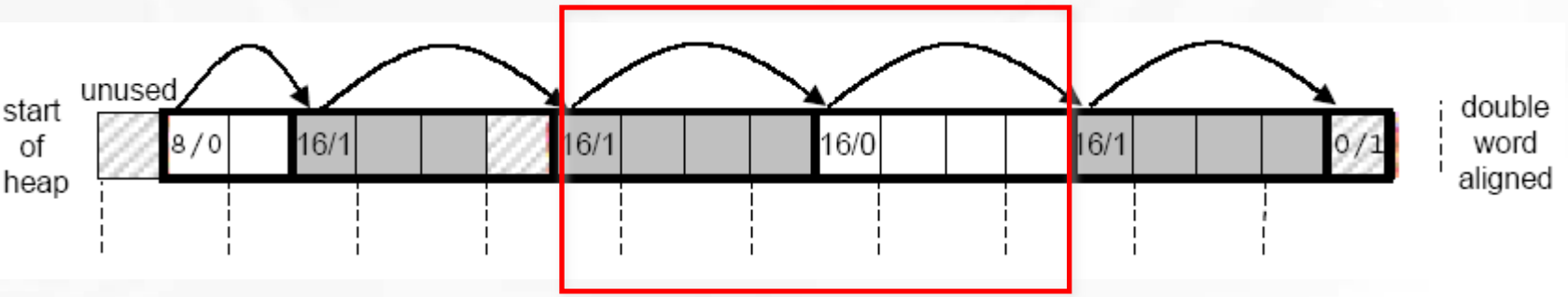
Bonus

Implementation Issues

Splitting Free Blocks



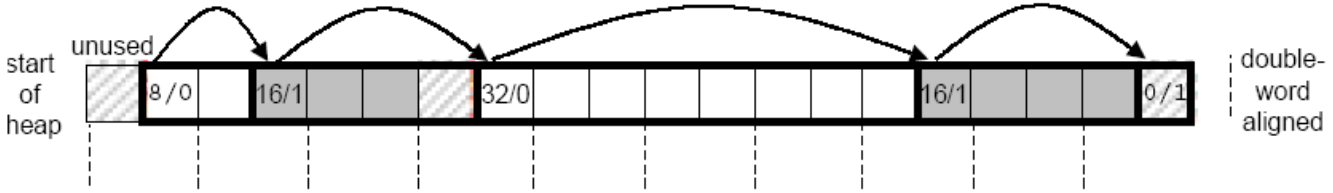
- However, if the fit is not good, then the allocator will usually opt to split the free block into two parts.
- The first part becomes the allocated block, and the remainder becomes a new free block.



Bonus

Implementation Issues

Getting Additional Heap Memory

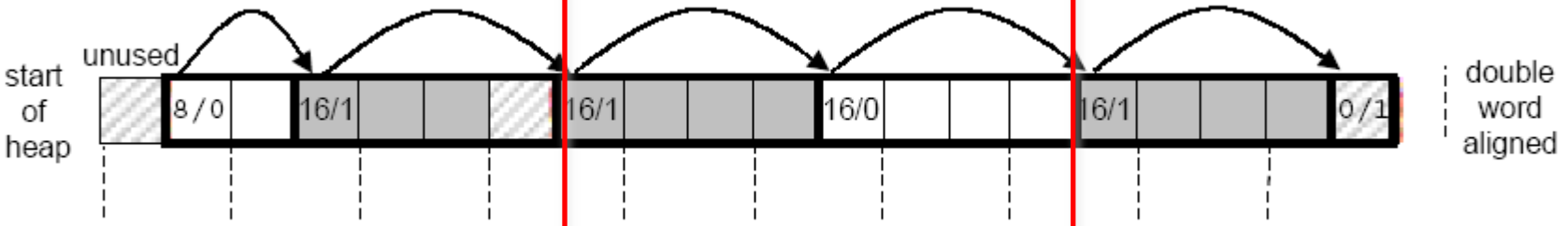


- What happens if the allocator is unable to find a fit for the requested block?
- Try to create some larger free blocks by merging (coalescing) free blocks that are physically adjacent in memory.
- However, if this does not work, or if the free blocks are already maximally coalesced, then the allocator asks the kernel for additional heap memory, either by calling the mmap or sbrk functions.

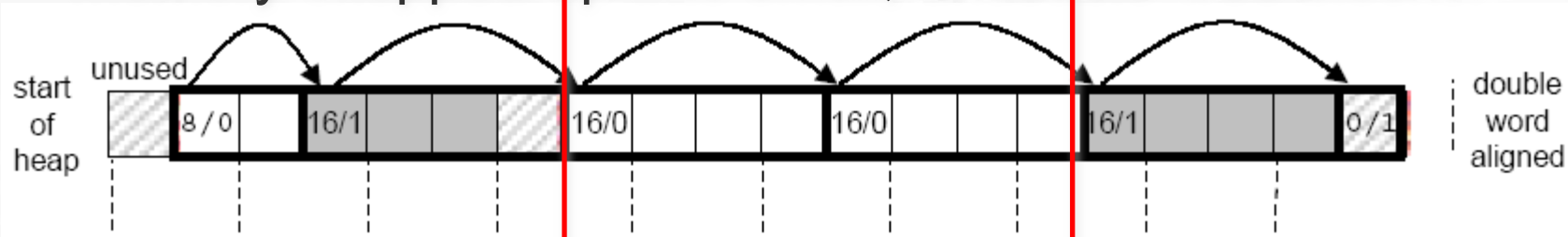
Bonus

Implementation Issues

Coalescing Free Blocks



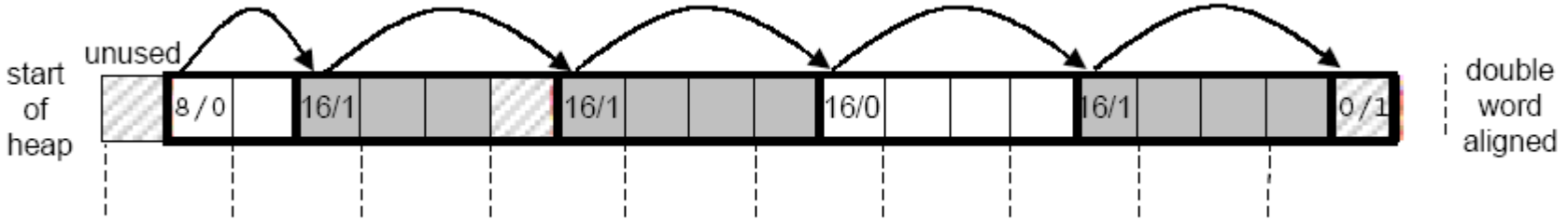
- When the allocator frees an allocated block, there might be other free blocks that are adjacent to the newly freed block.
- Such adjacent free blocks can cause ***false fragmentation***, where there is a lot of available free memory chopped up into small, unusable free blocks.



Bonus

Implementation Issues

Coalescing with Boundary Tags



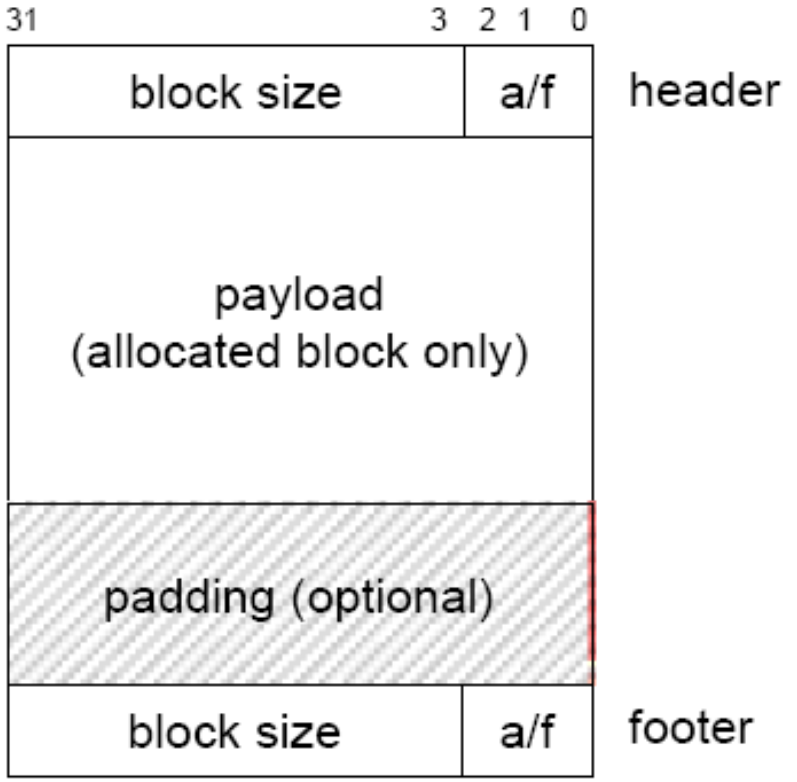
- Easy to coalesce the next free blocks
- Difficult to coalesce the previous free blocks
- ***Boundary Tags*** technique solves this problem

Bonus

Implementation Issues

Coalescing with Boundary Tags

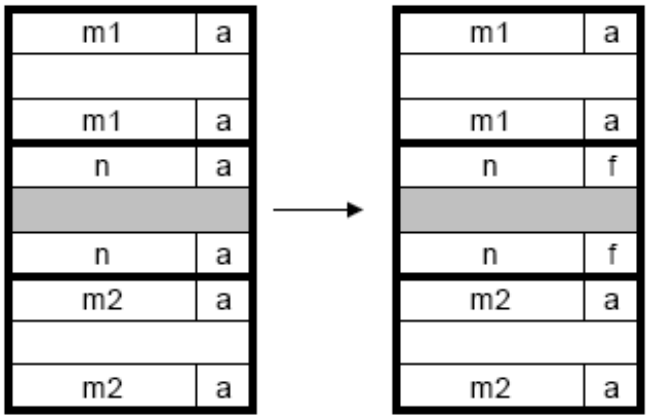
- Consider all cases that can exist when the allocator frees the current block:
  - The previous and next blocks are both allocated.
  - The previous block is allocated and the next block is free.
  - The previous block is free and the next block is allocated.
  - The previous and next blocks are both free.



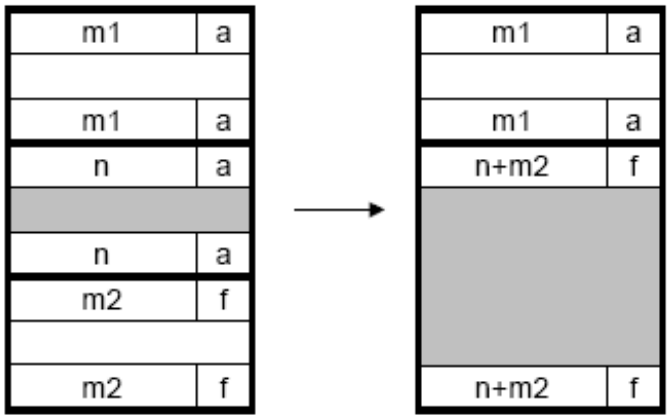
Bonus

Implementation Issues

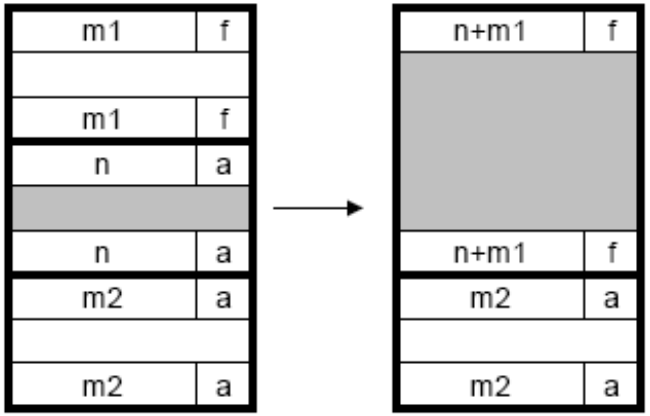
Coalescing with Boundary Tags



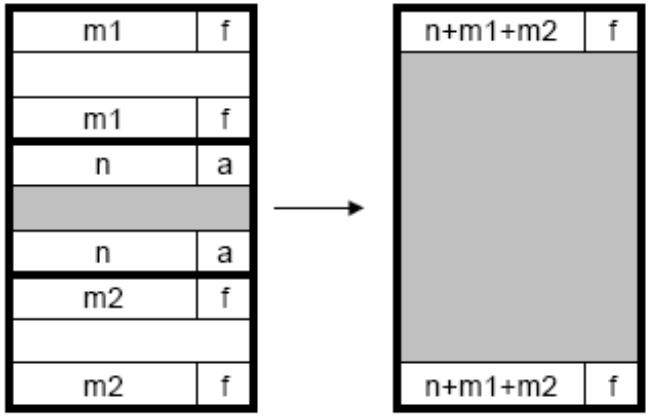
Case 1.



Case 2.



Case 3.



Case 4.

Bonus

Implementation Issues

Implementing a Simple Allocator

- Run our allocator without interfering with the existing system-level malloc package.
- The mem_init function models the virtual memory available to the heap as a double-word aligned array.
- The bytes between mem_start_brk and mem_brk represent allocated virtual memory.
- The bytes following mem_brk represent unallocated virtual memory.
- The allocator requests additional heap memory by calling the mem_sbrk function, which has the same interface as the system's sbrk function, and the same semantics, except that it rejects requests to shrink the heap.

Bonus

Implementation Issues

Implementing a Simple Allocator

- The allocator exports three functions to application programs:

```
1 int mm_init(void);
2 void *mm_malloc(size_t size);
3 void mm_free(void *bp);
```

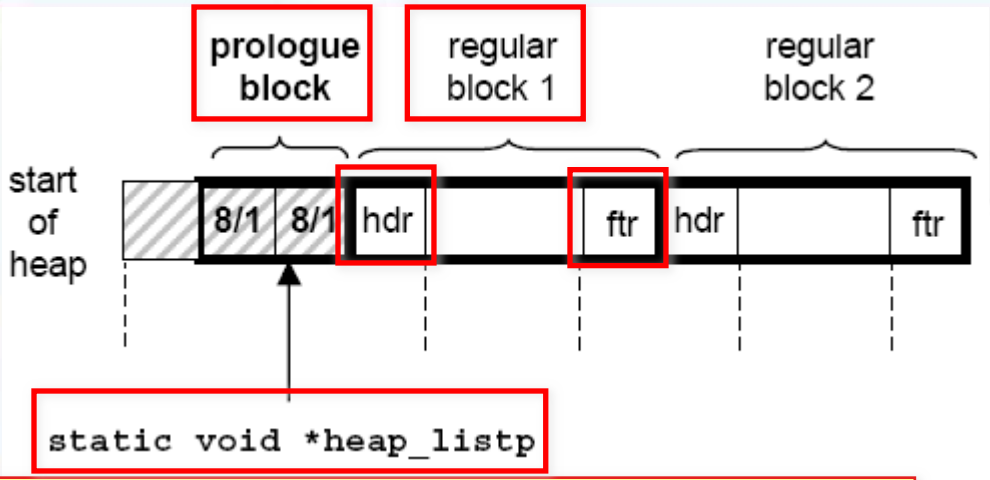
Bonus

Memory system model

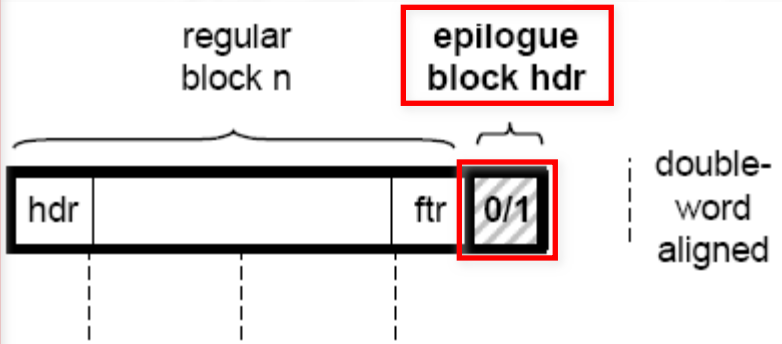
```
1  #include "csapp.h"
2
3  /* private global variables */
4  static void *mem_start_brk;  /* points to first byte of the heap */
5  static void *mem_brk;        /* points to last byte of the heap */
6  static void *mem_max_addr;   /* max virtual address for the heap */
7
8  /*
9   * mem_init - initializes the memory system model
10  */
11 void mem_init(int size)
12 {
13     mem_start_brk = (void *)Malloc(size); /* models available VM */
14     mem_brk = mem_start_brk;              /* heap is initially empty */
15     mem_max_addr = mem_start_brk + size;  /* max VM address for heap */
16 }
17
18 /*
19  * mem_sbrk - simple model of the the sbrk function. Extends the heap
20  *    by incr bytes and returns the start address of the new area. In
21  *    this model, the heap cannot be shrunk.
22  */
23 void *mem_sbrk(int incr)
24 {
25     void *old_brk = mem_brk;
26
27     if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
28         errno = ENOMEM;
29         return (void *)-1;
30     }
31     mem_brk += incr;
32     return old_brk;
33 }
```

Bonus

Memory system model



- The ***mm_init*** function initializes the allocator, returning 0 if successful and -1 otherwise.
- The ***mm_malloc*** and ***mm_free*** functions have the same interfaces and semantics as their system counterparts.



Bonus

Basic Constants and Macros

```
1  /* Basic constants and macros */
2  #define WSIZE       4       /* word size (bytes) */
3  #define DSIZE       8       /* doubleword size (bytes) */
4  #define CHUNKSIZE  (1<<12)  /* initial heap size (bytes) */
5  #define OVERHEAD    8       /* overhead of header and footer (bytes) */
6
7  #define MAX(x, y) ((x) > (y)? (x) : (y))
8
9  /* Pack a size and allocated bit into a word */
10 #define PACK(size, alloc)  ((size) | (alloc))
11
12 /* Read and write a word at address p */
13 #define GET(p)       (*(size_t *)(p))
14 #define PUT(p, val)  (*(size_t *)(p) = (val))
15
16 /* Read the size and allocated fields from address p */
17 #define GET_SIZE(p)  (GET(p) & ~0x7)
18 #define GET_ALLOC(p) (GET(p) & 0x1)
19
20 /* Given block ptr bp, compute address of its header and footer */
21 #define HDRP(bp)       ((void *)(bp) - WSIZE)
22 #define FTRP(bp)       ((void *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)
23
24 /* Given block ptr bp, compute address of next and previous blocks */
25 #define NEXT_BLKP(bp)  ((void *)(bp) + GET_SIZE(((void *)(bp) - WSIZE)))
26 #define PREV_BLKP(bp)  ((void *)(bp) - GET_SIZE(((void *)(bp) - DSIZE)))
```

Bonus

Creating the Initial Free List

```
1  int mm_init(void)
2  {
3      /* create the initial empty heap */
4      if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
5          return -1;
6      PUT(heap_listp, 0);                          /* alignment padding */
7      PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
8      PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
9      PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));   /* epilogue header */
10     heap_listp += DSIZE;
11
12     /* Extend the empty heap with a free block of CHUNKSIZE bytes */
13     if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
14         return -1;
15     return 0;
16 }
```

Bonus

Creating the Initial Free List

```
1  static void *extend_heap(size_t words)
2  {
3      char *bp;
4      size_t size;
5
6      /* Allocate an even number of words to maintain alignment */
7      size = (words % 2) ? (words+1) * WSIZE : words * WSIZE;
8      if ((int)(bp = mem_sbrk(size)) < 0)
9          return NULL;
10
11     /* Initialize free block header/footer and the epilogue header */
12     PUT(HDRP(bp), PACK(size, 0));          /* free block header */
13     PUT(FTRP(bp), PACK(size, 0));          /* free block footer */
14     PUT(HDRP(NEXT_BLKP(bp)), PACK(0, 1)); /* new epilogue header */
15
16     /* Coalesce if the previous block was free */
17     return coalesce(bp);
18 }
```

Bonus

Freeing and Coalescing Blocks

```
void mm_free(void *bp)
{
    size_t size = GET_SIZE(HDRP(bp));

    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}
```

Bonus

Freeing and Coalescing Blocks

```
static void *coalesce(void *bp)
{
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) {              /* Case 1 */
        return bp;
    }

    else if (prev_alloc && !next_alloc) {        /* Case 2 */
        size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
        PUT(HDRP(bp), PACK(size, 0));
        PUT(FTRP(bp), PACK(size,0));
        return(bp);
    }

    else if (!prev_alloc && next_alloc) {        /* Case 3 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp)));
        PUT(FTRP(bp), PACK(size, 0));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        return(PREV_BLKP(bp));
    }

    else {                                       /* Case 4 */
        size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
            GET_SIZE(FTRP(NEXT_BLKP(bp)));
        PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
        PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
        return(PREV_BLKP(bp));
    }
}
```

Bonus

Allocating Blocks

```
void *mm_malloc(size_t size)
{
    size_t asize;      /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char *bp;

    /* Ignore spurious requests */
    if (size <= 0)
        return NULL;

    /* Adjust block size to include overhead and alignment reqs. */
    if (size <= DSIZE)
        asize = DSIZE + OVERHEAD;
    else
        asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
        place(bp, asize);
        return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
        return NULL;
    place(bp, asize);
    return bp;
}
```

Bonus

alloc @ Objective-C

Extension

alloc @ Objective-C

Bonus

Alloc @ Objective-C

Objective-C的对象生成于堆之上，生成之后，需要一个指针来指向它。

```
ClassA *obj1 = [[ClassA alloc] init];
```

Objective-C的对象在使用完成之后不会自动销毁，需要执行dealloc来释放空间（销毁），否则内存泄露。

```
[obj1 dealloc];
```

下面代码中obj2是否需要调用dealloc？

```
ClassA *obj1 = [[ClassA alloc] init];
ClassA *obj2 = obj1;
[obj1 hello];      //输出hello
[obj1 dealloc];
[obj2 hello];      //能够执行这一行和下一行吗？
[obj2 dealloc];
```

Bonus

Alloc @ Objective-C

Objective-C采用了引用计数(ref count或者retain count)。
对象的内部保存一个数字，表示被引用的次数。例如，某个对象被两个指针所指向（引用）那么它的retain count为2。需要销毁对象的时候，不直接调用dealloc，而是调用release。
release会让retain count减1，
只有retain count等于0，系统才会调用dealloc真正销毁这个对象。

```
ClassA *obj1 = [[ClassA alloc] init]; //对象生成时，retain count = 1
[obj1 release]; //release使retain count减1，retain count = 0，dealloc自动被调用,对象被销毁
```

我们回头看看刚刚那个无效指针的问题，把dealloc改成release解决了吗？

```
ClassA *obj1 = [[ClassA alloc] init]; //retain count = 1
ClassA *obj2 = obj1; //retain count = 1
[obj1 hello];              //输出hello
[obj1 release];            //retain count = 0，对象被销毁
[obj2 hello];
[obj2 release];
```

Bonus

Alloc @ Objective-C

Objective-C指针赋值时，retain count不会自动增加，需要手动retain。

```
ClassA *obj1 = [[ClassA alloc] init];      //retain count = 1
ClassA *obj2 = obj1;                       //retain count = 1

[obj2 retain];                             //retain count = 2
[obj1 hello];                              //输出hello
[obj1 release];                            //retain count = 2 – 1 = 1
[obj2 hello];                              //输出hello
[obj2 release];                            //retain count = 0，对象被销毁
```

Bonus

Alloc @ Objective-C

Objective-C中引入了autorelease pool（自动释放对象池）
在遵守一些规则的情况下，可以自动释放对象。

新生成的对象，只要调用autorelease就行了，无需再调用release！

ClassA *obj1 = [[[ClassA alloc] init] autorelease]; //retain count = 1 但无需调用release

对于存在指针赋值的情况，与前面类似

```
ClassA *obj1 = [[[ClassA alloc] init] autorelease]; //retain count = 1
ClassA *obj2 = obj1;                //retain count = 1
[obj2 retain];                      //retain count = 2
[obj1 hello];                       //输出hello
//对于obj1，无需调用（实际上不能调用）release
[obj2 hello];                       //输出hello
[obj2 release];                     //retain count = 2-1 = 1
```

Bonus

Alloc @ Objective-C

autorelease pool原理剖析

1. autorelease pool需要手动创立。

   NSAutoreleasePool *pool = [[NSAutoreleasePool alloc] init];

2. NSAutoreleasePool内部包含一个数组（NSMutableArray），用来保存声明为autorelease的所有对象。

   如果一个对象声明为autorelease，系统所做的工作就是把这个对象加入到这个数组中去。

   ClassA *obj1 = [[[ClassA alloc] init] autorelease];

   //retain count = 1，把此对象加入autorelease pool中

3. NSAutoreleasePool自身在销毁的时候，会遍历一遍这个数组，release数组中的每个成员。如果此时数组中成员的retain count为1那么release之后，retain count为0，对象正式被销毁。

   如果此时数组中成员的retain count大于1，那么release之后，retain count大于0，此对象依然没有被销毁，内存泄露。

Bonus

Alloc @ Objective-C

默认只有一个autorelease pool

```
int main (int argc, const char *argv[])
{
NSAutoreleasePool *pool;
pool = [[NSAutoreleasePool alloc] init];

// do something

[pool release];
return (0);
}
```

大量的对象标记为autorelease，不能很好的利用内存

```
int main (int argc, const char *argv[])
{
NSAutoreleasePool *pool =
                [[NSAutoreleasePool alloc] init];
int i, j;

for (i = 0; i < 100; i++ )
{
 for (j = 0; j < 100000; j++ )
   [NSString stringWithFormat:@“1234567890”];
                //产生的对象是autorelease的
}

[pool release];
return (0);
}
```

SSD
6

icarnegie
curriculum powered by Carnegie Mellon

Lecture 5 (2/2)

Memory Layout and Allocation

武漢大學

国际软件学院

Lecture 3 Memory Layout and Allocation

The Contents in SSD6 cover:

3.1 Several Uses of Memory

3.2 Memory Bugs

3

Lecture 3

Memory Bugs

In this Part:

- 3.2.1 Review of Pointers in C
- 3.2.2 Making and Using Bad References
- 3.2.3 Overwriting Memory
- 3.2.4 Twice free
- 3.2.5 Memory Leaks
- 3.2.6 Exterminating Memory Bugs
- 3.2.7 Garbage Collection

Lecture 3 Memory Bugs

Malloc Function:

According to MSDN:

Run-Time Library Reference

malloc

Allocates memory blocks.

```
void *malloc(
   size_t size
);
```

Parameter

size
Bytes to allocate.

See crt_malloc.htm

Lecture 3 Memory Bugs

About *size_t*:

- Originally defined in C Programming Language
- Usually defined as Unsigned INT @ i386x32
- Or, defined as Unsigned LONG @ x64
- For example, in Redhat 9:
  - Defined at /usr/include/linux/types.h
    - ✓ typedef __kernel_size_t size_t;
  - While __kernel_size_t was defined at
    - ✓ /usr/include/asm/posix_types.h
    - ✓ typedef unsigned int __kernel_size_t;

Lecture 3

Review of Pointers in C

```
// take the address of a variable
int var;          // declare the variable
int *var_ptr; // declare the pointer
var_ptr = &var; // take the address of var
*var_ptr = 3; // stores 3 into var
// access variable pointed to by var_ptr
if (var == *var_ptr) printf("ok\n");
```

```
// allocate an integer with malloc

// the result must be coerced into an (int *)
var_ptr = (int *) malloc(sizeof(int));
*var_ptr = 4;

// free the memory
free(var_ptr);
```

```
// declare a structure
typedef struct {
        int int_field;
        double dbl_field;
} my_struct_type;


// allocate a structure on the heap
my_struct_type *s;
s = (my_struct_type *) malloc(sizeof(my_struct_type));
// initialize fields of s
s->int_field = 0;
s->dbl_field = 0.0;



// access s in another way:
(*s).int_field = 0; // equivalent to s->int_field = 0;
```

Lecture 3

Review of Pointers in C

```
void mult_array_by_scalar(double x[], int n, double y)
{

        double *ptr = x; // point to first element in x
        for (int i = 0; i < n; i++) {
                *ptr++ *= y;
        }
}
```

- Take the value of ptr and increment it by one.
- ptr++, evaluates to the value of ptr *before* the increment operation, so now take that original value of ptr and dereference it (indicated by the * operator) so that the expression now refers to a variable of type double;
- Use the *= operator, which multiplies the variable on the left by the value on the right.

Lecture 3 Memory Bugs

```
int sum(int a[], int n)
{
        int *p;
        int sum = 0;
        for (int i = 0; i < n; i++)
                sum += *p++;
}
```

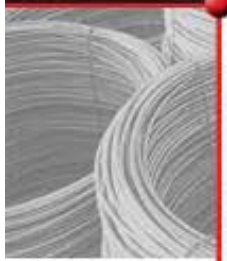
Uninitialized Data

```
int i;
double d;
scanf("%d %g", i, d); // wrong!!!
// here is the correct call:
scanf("%d %g", &i, &d);
```

scanf() is one of the few functions that do not specify types for all their arguments …

Lecture 3

Overwriting Memory





```
#define array_size 100
int *a = (int *) malloc(sizeof(int *) * array_size);
for (int i = 0; i <= array_size; i++)
        a[i] = NULL;
```

Off-by-One Bug

Lecture 3

Overwriting Memory

```
#define array_size 100
int *a = (int *) malloc(array_size);
a[99] = 0; // this overwrites memory beyond the block
```

Allocate too less memory

Lecture 3

Overwriting Memory

```
char *heapify_string(char *s)
{
        int len = strlen(s);

        char *new_s = (char *) malloc(len);
        strcpy(new_s, s);
        return new_s;
}
```

Data will be overwritten due to **Too Less Memory** allocated

Lecture 3

Overwriting Memory

```
// decrement a if a is greater than zero:
void dec_positive(int *a)
{
        *a--; // decrement the integer
        if (*a < 0) *a = 0; // make sure a is positive
}
```

Referencing a **Pointer** instead of the **Object** it points to

Lecture 3

Overwriting Memory

```
int *ptr_to_zero()

{
        int i = 0;
        return &i;
}
```

Never return a Local Pointer!

Lecture 3

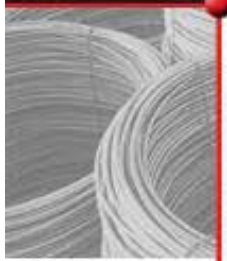
Twice Free

```
void my_write(x)
{
        ... use x ...
        free(x);
}


int *x = (int *) malloc(sizeof(int*) * N);
...
my_read(x);
...
my_write(x);
...
my_read(x); // oops, x was freed by my write!
```

Referencing Freed Blocks

Lecture 3

Memory Leaks





- The failure to deallocate a block of memory when it is no longer needed is often called a memory leak;
- Lucky:
  - Run-and-Exit
- Unlucky:
  - Exhausted-and-kick-out

Lecture 3

Memory Leaks

Memory leak examples [I]

```
void my_function(char *msg)
{
        // allocate space for a string
        char *full_msg = (char *) malloc(strlen(msg) + 100);
        strcpy(full_msg, "The following error was encountered: ");
        strcat(full_msg, msg);
        if (!display(full_msg)) return;
        ...
        free(full_msg);
}
```

Lecture 3

Memory Leaks

Memory leak examples [II]

```
typedef struct    {
        char *name;
        int age;
        char *address;
        int phone;
} Person;



void my_function()
{
        Person *p = (Person *) malloc(sizeof(Person));
        p->name = (char *) malloc(M);
        ...
        p->address = (char *) malloc(N);
        ...
        free(p); // what about name and address?
```

Lecture 3

Exterminating Memory Bugs

Memory Allocators as Debugging Tools

- **The file name and line number where the allocation occurred.** At the end of execution, any unallocated blocks that might indicate memory leaks can be listed along with the point in the code where the call to malloc() was made.
- **The status: whether the block is allocated or free.** If a block is freed twice, this can usually be detected. The program is stopped immediately, and the debugger is invoked to show from where the extra free() was called.
- **Padding before and after the block.** This padding is filled with a known value such as "0xdeadbeef" so that if memory is overwritten near the boundaries of the block, the changes to the known values can easily be detected.
- **Links to other blocks.** These enable allocated blocks to be scanned and checked.
- **An allocation sequence number.** This helps to identify blocks. Sometimes, you can set a breakpoint to stop when malloc() is about to allocate the block with a given sequence number. This is very useful when you want to figure out where a particular block came from.

Lecture 3

Exterminating Memory Bugs

Memory Allocators as Debugging Tools

- **Define your own *Malloc() & Free()***
- #define malloc(size) my_malloc(size, __FILE__, __LINE__)
- #undefine malloc

  ptr = malloc(size); // calls the "real" malloc

Lecture 3

Exterminating Memory Bugs

Debugging in VC++:

- A pointer to the previously allocated block;
- A pointer to the next allocated block;
- The source file name and line number where the allocation originated;
- An allocation sequence number;
- Memory filled with the constant 0xFD to catch overwrites.

_CrtCheckMemory(): check the integrity of the debug heap

Lecture 3

Exterminating Memory Bugs

```
void main()
{

    int *a=(int *)malloc(sizeof(int));
    int *b=(int *)malloc(sizeof(int));

    free (a);
    free(b);
}
```

```
    int *a=(int *)ma
    int *b=(int *)ma
```

```
    int
    int

    free
    free
```

```
004316E8   E0 06 00 00   ....
004316EC   31 00 00 00   1...
004316F0   20 17 43 00    .C.
004316F4   00 00 00 00   ....
004316F8   00 00 00 00   ....
004316FC   00 00 00 00   ....
00431700   04 00 00 00   ....
00431704   01 00 00 00   ....
00431708   33 00 00 00   3...
0043170C   FD FD FD FD
00431710   CD CD CD CD   屯屯
00431714   FD FD FD FD
00431718   31 00 00 00   1...
0043171C   31 00 00 00   1...
00431720   88 49 37 00   圍7.
00431724   F0 16 43 00   ..C.
```

```
0043171C   30 00 00 00   0...
00431720   FC 08 37 00   ..7.
00431724   FC 08 37 00   ..7.
00431728   DD DD DD DD   葺葺
0043172C   DD DD DD DD   葺葺
00431730   DD DD DD DD   葺葺
00431734   DD DD DD DD   葺葺
00431738   DD DD DD DD   葺葺
0043173C   DD DD DD DD   葺葺
00431740   DD DD DD DD   葺葺
00431744   DD DD DD DD   葺葺
00431748   30 00 00 00   0...
```

See 3.2.6

Lecture 3

Exterminating Memory Bugs

i=3

```
00370788   00 00 00 00   ....
0037078C   00 00 00 00   ....
00370790   09 00 29 00   ■.).
00370794   23 07 18 00   #...
00370798   B0 27 37 00   .'7.
0037079C   00 00 00 00   ....
003707A0   00 00 00 00   ....
003707A4   00 00 00 00   ....
003707A8   0C 00 00 00   ....
003707AC   01 00 00 00   ....
003707B0   2A 00 00 00   *...
003707B4   FD FD FD FD
003707B8   CD CD CD CD   屯屯
003707BC   CD CD CD CD   屯屯
003707C0   CD CD CD CD   屯屯
003707C4   FD FD FD FD
003707C8   AB AB AB AB
003707CC   AB AB AB AB
003707D0   00 00 00 00   ....
003707D4   00 00 00 00   ....
003707D8   4E 01 09 00   N.■.
003707DC   EE 04 EE 00   ....
003707E0   78 01 37 00   x.7.
003707E4   78 01 37 00   x.7.
003707E8   EE FE EE FE   锇锇
003707EC   EE FE EE FE   锇锇
```

i=8

```
00370788   00 00 00 00   ....
0037078C   00 00 00 00   ....
00370790   0D 00 29 00   ..).
00370794   65 07 18 00   e...
00370798   B0 27 37 00   .'7.
0037079C   00 00 00 00   ....
003707A0   00 00 00 00   ....
003707A4   00 00 00 00   ....
003707A8   20 00 00 00    ...
003707AC   01 00 00 00   ....
003707B0   2A 00 00 00   *...
003707B4   FD FD FD FD
003707B8   CD CD CD CD   屯屯
003707BC   CD CD CD CD   屯屯
003707C0   CD CD CD CD   屯屯
003707C4   CD CD CD CD   屯屯
003707C8   CD CD CD CD   屯屯
003707CC   CD CD CD CD   屯屯
003707D0   CD CD CD CD   屯屯
003707D4   CD CD CD CD   屯屯
003707D8   FD FD FD FD
003707DC   0D F0 AD BA   .凯.
003707E0   0D F0 AD BA   .凯.
003707E4   0D F0 AD BA   .凯.
003707E8   AB AB AB AB
003707EC   AB AB AB AB
```

Lecture 3 Next Time……

- Read:
  - Unit 4 (Performance Measurement and Improvement)
- Do:
  - Multiple-Choice Quizzes in Unit 4

Lecture 3

In the Lab……

- Excise 3 (Debugging Malloc Lab)
  - Get debugging_malloc.zip from ftp@iss
  - Define your own Malloc() & Free()
  - Define your own Structure for Payload