# Operating Systems
# CMPSC 473

## Dynamic Memory Allocation

## CPU/memory virtualization

### September 19 2023 – Lecture 9

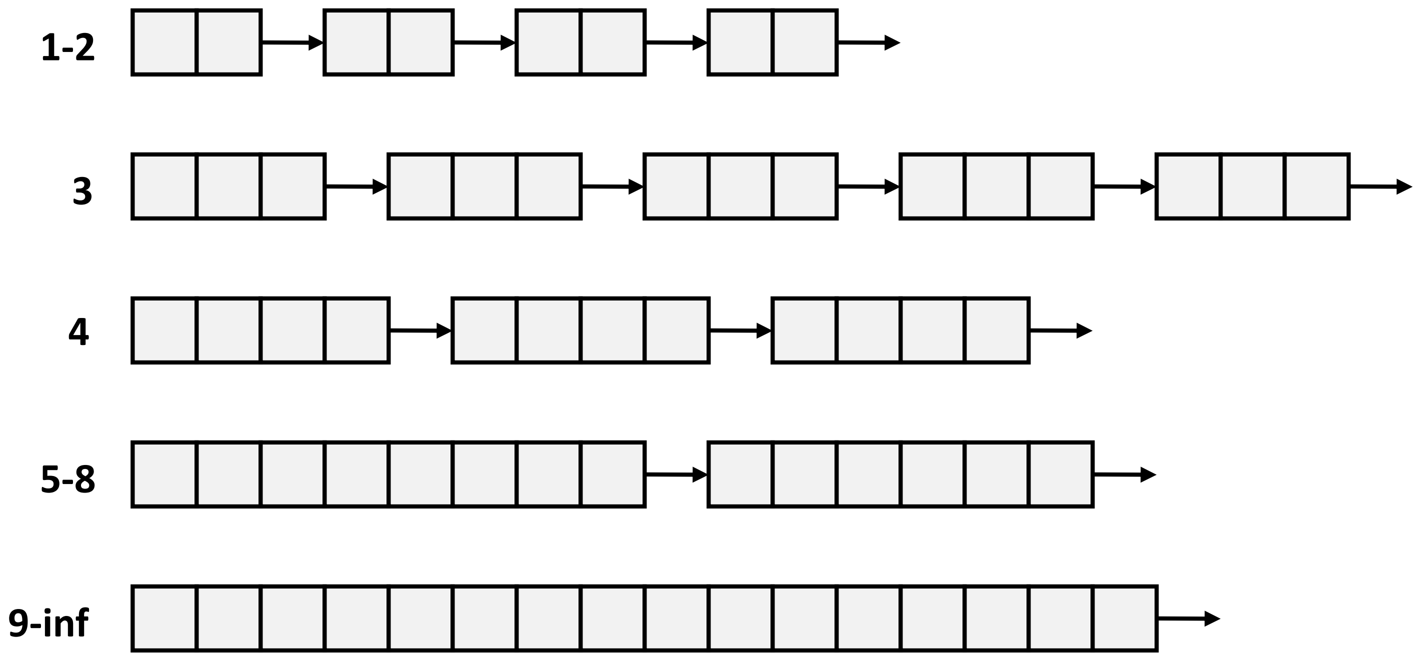

### Instructor: Ruslan Nikolaev



**ACK: some slides adapted from CSAPP text**

# Segregated List (Seglist) Allocators

- Each ***size class*** of blocks has its own free list




- Often have separate classes for each small size
- For larger sizes: One class for each two-power size

# Seglist Allocator

- **Given an array of free lists, each one for some size class**

- **To allocate a block of size *n*:**
  - Search appropriate free list for block of size $m >= n$
  - If an appropriate block is found:
    - Split block and place fragment on appropriate list (optional)
  - If no block is found, try next larger class
  - Repeat until block is found

- **If no block is found:**
  - Request additional heap memory from OS (using **`sbrk()`**)
  - Allocate block of *n* bytes from this new memory
  - Place remainder as a single free block in the corresponding size class

# Seglist Allocator (cont.)

- **To free a block:**
  - Coalesce and place on appropriate list

- **Advantages of seglist allocators**
  - Higher throughput
    - log time for power-of-two size classes
  - Better memory utilization
    - First-fit search of segregated free list approximates a best-fit search of entire heap.
    - Extreme case: Giving each block its own size class is equivalent to best-fit.

# Memory-Related Perils and Pitfalls

- Dereferencing bad pointers
- Reading uninitialized memory
- Overwriting memory
- Referencing nonexistent variables
- Freeing blocks multiple times
- Referencing freed blocks
- Failing to free blocks

# C operators

<table>
<tr><th>Operators</th><th>Associativity</th></tr>
<tr><td><code>() [] -> .</code></td><td>left to right</td></tr>
<tr><td><code>! ~ ++ -- + - * & (type) sizeof</code></td><td>right to left</td></tr>
<tr><td><code>* / %</code></td><td>left to right</td></tr>
<tr><td><code>+ -</code></td><td>left to right</td></tr>
<tr><td><code><< >></code></td><td>left to right</td></tr>
<tr><td><code>< <= > >=</code></td><td>left to right</td></tr>
<tr><td><code>== !=</code></td><td>left to right</td></tr>
<tr><td><code>&</code></td><td>left to right</td></tr>
<tr><td><code>^</code></td><td>left to right</td></tr>
<tr><td><code>|</code></td><td>left to right</td></tr>
<tr><td><code>&&</code></td><td>left to right</td></tr>
<tr><td><code>||</code></td><td>left to right</td></tr>
<tr><td><code>?:</code></td><td>right to left</td></tr>
<tr><td><code>= += -= *= /= %= &= ^= != <<= >>=</code></td><td>right to left</td></tr>
<tr><td><code>,</code></td><td>left to right</td></tr>
</table>

- **`->`, `()`, and `[]` have high precedence, with `*` and `&` just below**
- **Unary `+`, `-`, and `*` have higher precedence than binary forms**

**Source: K&R page 53**

# C Pointer Declarations: Test Yourself!

| | |
|---|---|
| `int *p` | p is a pointer to int |
| `int *p[13]` | p is an array[13] of pointer to int |
| `int *(p[13])` | p is an array[13] of pointer to int |
| `int **p` | p is a pointer to a pointer to an int |
| `int (*p)[13]` | p is a pointer to an array[13] of int |
| `int *f()` | f is a function returning a pointer to int |
| `int (*f)()` | f is a pointer to a function returning int |
| `int (*(*f())[13])()` | f is a function returning ptr to an array[13] of pointers to functions returning int |
| `int (*(*x[3])())[5]` | x is an array[3] of pointers to functions returning pointers to array[5] of ints |

Source: K&R Sec 5.12

# Dereferencing Bad Pointers

- **The classic `scanf` bug**

```
int val;

...

scanf("%d", val);
```

# Reading Uninitialized Memory

- **Assuming that heap data is initialized to zero**

```
/* return y = Ax */
int *matvec(int **A, int *x) {
   int *y = malloc(N*sizeof(int));
   int i, j;

   for (i=0; i<N; i++)
      for (j=0; j<N; j++)
         y[i] += A[i][j]*x[j];
   return y;
}
```

To make sure the code works correctly without relying on the mentioned assumption, you should either initialize the y vector to zeros manually after allocating it:

```
int *y = malloc(N * sizeof(int));
for (int k = 0; k < N; k++) {
    y[k] = 0;
}
```

# Overwriting Memory

- **Allocating the (possibly) wrong sized object**

```
int **p;

p = malloc(N*sizeof(int));

for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

```
int **p;
p = malloc(N * sizeof(int*));
for (i=0; i<N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

# Overwriting Memory

- **Off-by-one error**

```
int **p;

p = malloc(N*sizeof(int *));

for (i=0; i<=N; i++) {
   p[i] = malloc(M*sizeof(int));
}
```

```
for (i=0; i<N; i++) {
    p[i] = malloc(M*sizeof(int));
}
```

# Overwriting Memory

- **Not checking the max string size**

```
char s[8];
int i;

gets(s);  /* reads "123456789" from stdin */
```

- **Basis for classic buffer overflow attacks**

# Overwriting Memory

- **Misunderstanding pointer arithmetic**

```
int *search(int *p, int val) {

   while (*p && *p != val)
      p += sizeof(int);

   return p;
}
```

```
int *search(int *p, int val) {
   while (*p && *p != val)
      p++;
   return p;
}
```

# Overwriting Memory

- **Referencing a pointer instead of the object it points to**

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   *size--;
   Heapify(binheap, *size, 0);
   return(packet);
}
```

```
int *BinheapDelete(int **binheap, int *size) {
   int *packet;
   packet = binheap[0];
   binheap[0] = binheap[*size - 1];
   (*size)--;
   Heapify(binheap, *size, 0);
   return packet;
}
```

# Referencing Nonexistent Variables

- **Forgetting that local variables disappear when a function returns**

```
int *foo () {
   int val;

   return &val;
}
```

# Freeing Blocks Multiple Times

- **Nasty!**

```
x = malloc(N*sizeof(int));
        <manipulate x>
free(x);

y = malloc(M*sizeof(int));
        <manipulate y>
free(x);
```

# Referencing Freed Blocks

- **Evil!**

```
x = malloc(N*sizeof(int));
  <manipulate x>
free(x);
    ...
y = malloc(M*sizeof(int));
for (i=0; i<M; i++)
   y[i] = x[i]++;
```

# Failing to Free Blocks (Memory Leaks)

- **Slow, long-term killer!**

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   return;
}
```

The pointer x is allocated memory dynamically but is never freed before the function foo() returns. When the function returns, the local pointer x goes out of scope, and we lose the reference to the allocated memory, leading to a memory leak.

To fix the memory leak, you need to free the allocated memory before the function returns:

```
foo() {
   int *x = malloc(N*sizeof(int));
   ...
   free(x);
   return;
}
```

# Failing to Free Blocks (Memory Leaks)

- **Freeing only part of a data structure**

```
struct list {
   int val;
   struct list *next;
};

foo() {
   struct list *head = malloc(sizeof(struct list));
   head->val = 0;
   head->next = NULL;
   <create and manipulate the rest of the list>
     ...
   free(head);
   return;
}
```

This loop iteratively frees each node in the list by temporarily holding a reference to the current node, moving the head pointer to the next node, and then freeing the temporary node.

The corrected foo() function would be:

c
Copy code

```
foo() {
    struct list *head = malloc(sizeof(struct list));
    head->val = 0;
    head->next = NULL;
    // <create and manipulate the rest of the list>

    while (head != NULL) {
        struct list *temp = head;
        head = head->next;
        free(temp);
    }
    return;
}
```