Inner Workings of Malloc and Free

Prof. David August

COS 217

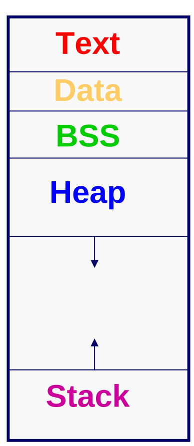
Goals of This Lecture

- Understanding how the heap is managed
  - Malloc: allocate memory
  - Free: deallocate memory
- K&R implementation (Section 8.7)
  - Free list
    - Free block with header (pointer and size) and user data
    - Aligning the header with the largest data type
    - Circular linked list of free blocks
  - Malloc
    - Allocating memory in multiples of header size
    - Finding the first element in the free list that is large enough
    - Allocating more memory from the OS, if needed
  - Free
    - Putting a block back in the free list
    - Coalescing with adjacent blocks, if any

Memory Layout: Heap

```
char* string = “hello”;
int iSize;

char* f(void)
{
    char* p;
    iSize = 8;
    p = malloc(iSize);
    return p;
}
```



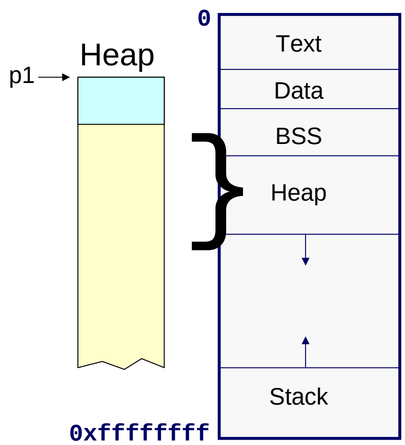
Using Malloc and Free

- Types
  - **`void*`**: generic pointer to any type (can be converted to other pointer types)
  - **`size_t`**: unsigned integer type returned by **`sizeof()`**
- **`void *malloc(size_t size)`**
  - Returns a pointer to space of size **`size`**
  - … or **`NULL`** if the request cannot be satisfied
  - E.g., **`int* x = (int *) malloc(sizeof(int));`**
- **`void free(void *p)`**
  - Deallocate the space pointed to by the pointer **`p`**
  - Pointer **`p`** must be pointer to space previously allocated
  - Do nothing if **`p`** is **`NULL`**

Heap: Dynamic Memory

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

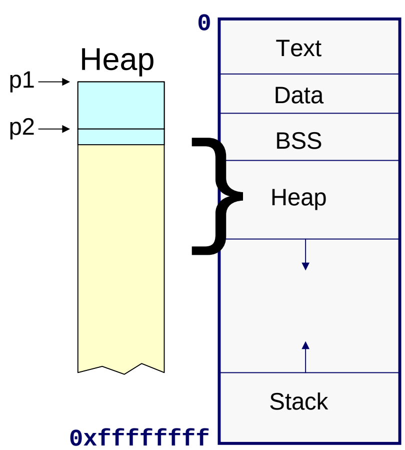
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

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

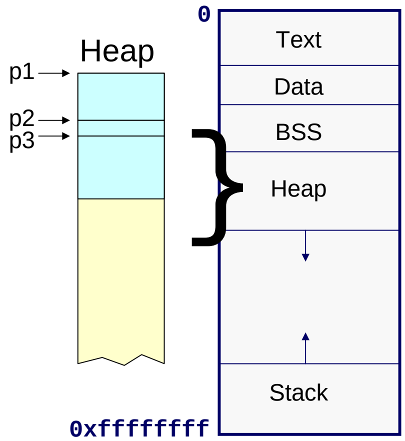
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);
```



Heap: Dynamic Memory

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

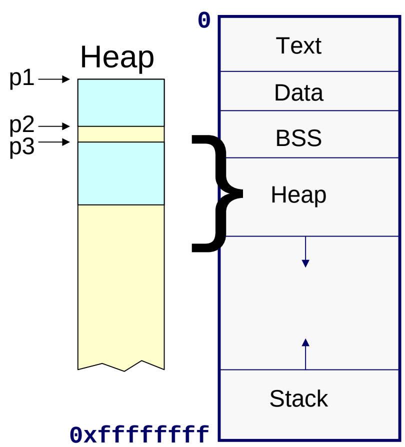
```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
➡  char *p3 = malloc(4);
   free(p2);
   char *p4 = malloc(6);
   free(p3);
   char *p5 = malloc(2);
   free(p1);
   free(p4);
   free(p5);
```



Heap: Dynamic Memory

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

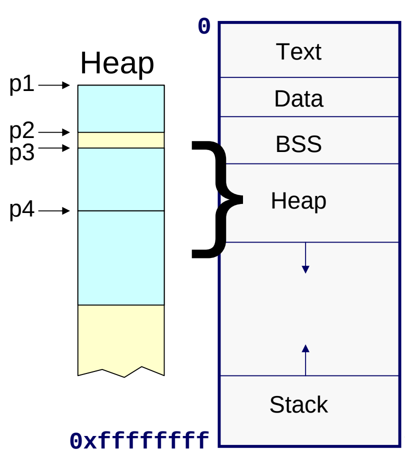
```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
   char *p3 = malloc(4);
➡  free(p2);
   char *p4 = malloc(6);
   free(p3);
   char *p5 = malloc(2);
   free(p1);
   free(p4);
   free(p5);
```



Heap: Dynamic Memory

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

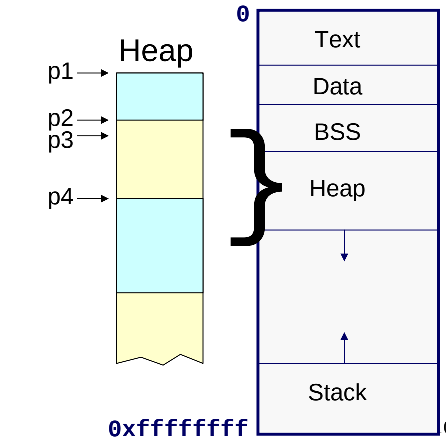
```
   char *p1 = malloc(3);
   char *p2 = malloc(1);
   char *p3 = malloc(4);
   free(p2);
➡  char *p4 = malloc(6);
   free(p3);
   char *p5 = malloc(2);
   free(p1);
   free(p4);
   free(p5);
```



Heap: Dynamic Memory

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

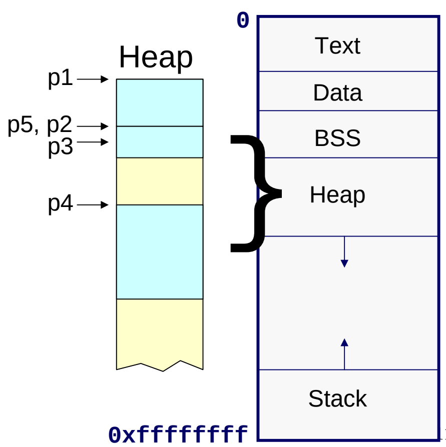
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
⇨ free(p3);
  char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Heap: Dynamic Memory

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

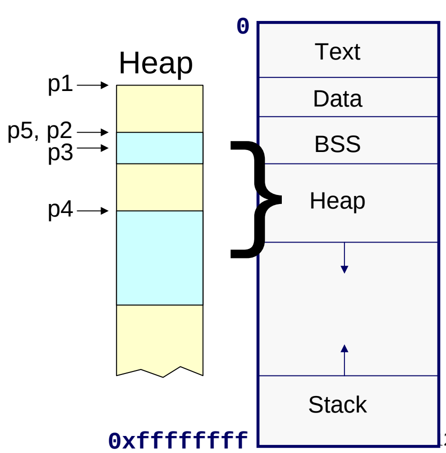
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
⇨ char *p5 = malloc(2);
  free(p1);
  free(p4);
  free(p5);
```



Heap: Dynamic Memory

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

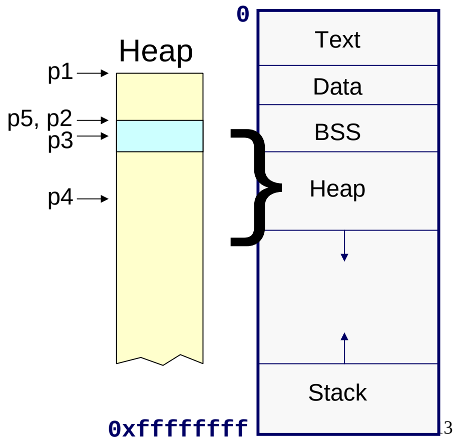
```
  char *p1 = malloc(3);
  char *p2 = malloc(1);
  char *p3 = malloc(4);
  free(p2);
  char *p4 = malloc(6);
  free(p3);
  char *p5 = malloc(2);
⇨ free(p1);
  free(p4);
  free(p5);
```



Heap: Dynamic Memory

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

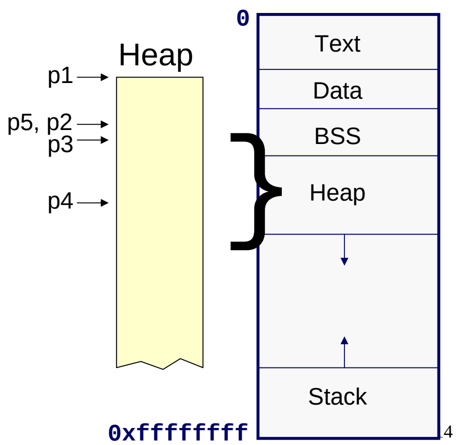
```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);   ⇨
free(p5);
```



Heap: Dynamic Memory

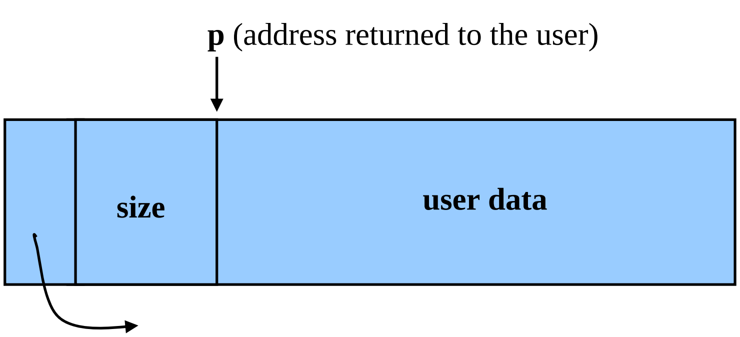
```
#include <stdlib.h>
void *malloc(size_t size);
void free(void *ptr);
```

```
char *p1 = malloc(3);
char *p2 = malloc(1);
char *p3 = malloc(4);
free(p2);
char *p4 = malloc(6);
free(p3);
char *p5 = malloc(2);
free(p1);
free(p4);
free(p5);   ⇨
```



Free Block: Pointer, Size, Data

- Free block in memory
  - Pointer to the next block
  - Size of the block
  - User data



Free Block: Memory Alignment

- Define a structure **s** for the header
  - Pointer to the next free block (**ptr**)
  - Size of the block (**size**)
- To simplify memory alignment
  - Make all memory blocks a multiple of the header size
  - Ensure header is aligned with largest data type (e.g., **long**)
- Union: C technique for forcing memory alignment
  - Variable that may hold objects of different types and sizes
  - Made large enough to hold the largest data type, e.g.,

```
union Tag {
    int ival;
    float fval;
    char *sval;
} u;
```


Free Block: Memory Alignment

```
/* align to long boundary */
typedef long Align;

union header { /* block header */
   struct {
      union header *ptr;
      unsigned size;
   } s;
   Align x;    /* Force alignment */
}
typedef union header Header;
```


Allocate Memory in Units

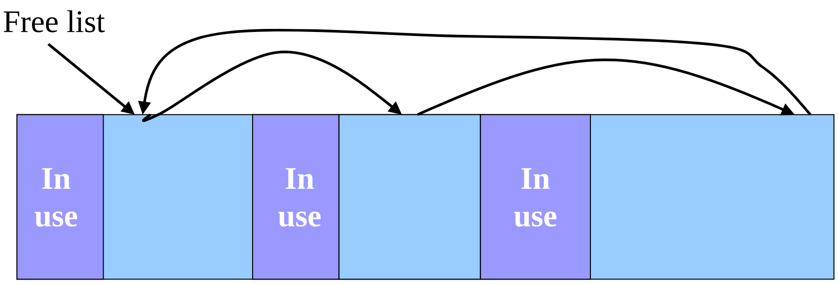
- Keep memory aligned
  - Requested size is rounded up to multiple of header size
- Rounding up when asked for nbytes
  - Header has size **sizeof(Header)**
  - Round: **(nbytes + sizeof(Header) – 1)/sizeof(Header)**
- Allocate space for user data, plus the header itself

```
void *malloc(unsigned int nbytes) {
   unsigned nunits;

   nunits = (nbytes + sizeof(Header)
              – 1)/sizeof(Header) + 1;
   …
}
```

Free List: Circular Linked List

- Free blocks, linked together
  - Example: circular linked list
- Keep list in order of increasing addresses
  - Makes it easier to coalesce adjacent free blocks

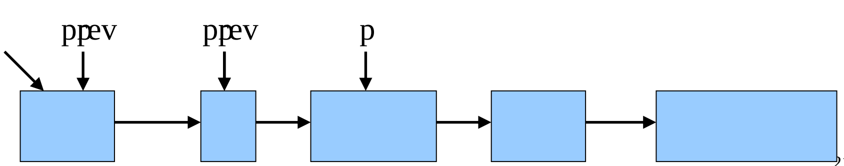


Allocation Algorithms

- Handling a request for memory (e.g., malloc)
  - Find a free block that satisfies the request
  - Must have a “size” that is big enough, or bigger
- Which block to return?
  - First-fit algorithm
    - Keep a linked list of free blocks
    - Search for the *first* one that is big enough
  - Best-fit algorithm
    - Keep a linked list of free blocks
    - Search for the *smallest* one that is big enough
    - Helps avoid fragmenting the free memory

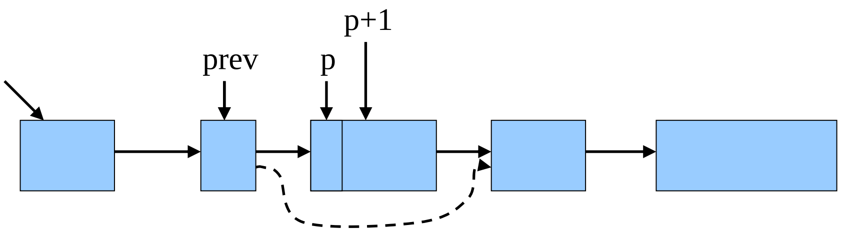
Malloc: First-Fit Algorithm

- Start at the beginning of the list
- Sequence through the list
  - Keep a pointer to the previous element
- Stop when reaching first block that is big enough
  - Patch up the list
  - Return a block to the user



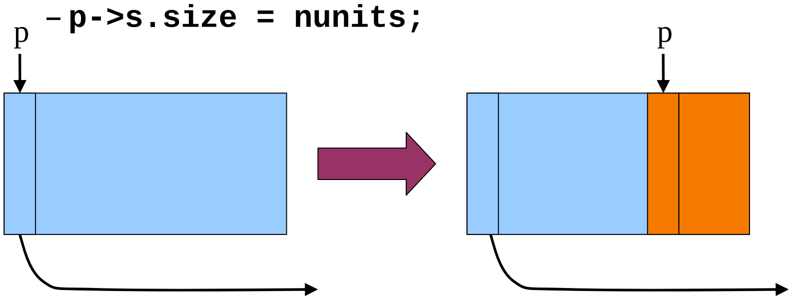
First Case: A Perfect Fit

- Suppose the first fit is a perfect fit
  - Remove the element from the list
  - Link the previous element with the next element
    - **`prev->s.ptr = p->s.ptr`**
  - Return the current element to the user (skipping header)
    - **`return (void *) (p+1)`**



Second Case: Block is Too Big

- Suppose the block is bigger than requested
  - Divide the free block into two blocks
  - Keep first (now smaller) block in the free list
    - **`p->s.size -= nunits;`**
  - Allocate the second block to the user
    - **`p += p->s.size;`**
    - **`p->s.size = nunits;`**



Combining the Two Cases

```
prevp = freep; /* start at beginning */
for (p=prevp->s.ptr; ; prevp=p,
                       p=p->s.ptr) {
  if (p->s.size >= nunits) {
    if (p->s.size == nunits) /* fit */
      prevp->s.ptr = p->s.ptr;
    else { /* too big, split in two */
      p->s.size -= nunits;  /* #1 */
      p += p->s.size;       /* #2 */
      p->s.size = nunits;   /* #2 */
    }
    return (void *)(p+1);
  }
}
```

Beginning of the Free List

- Benefit of making free list a circular list
  - Any element in the list can be the beginning
  - Don't have to handle the "end" of the list as special
  - Optimization: make head be where last block was found

```
prevp = freep; /* start at beginning */
for (p=prevp->s.ptr; ; prevp=p,
                       p=p->s.ptr) {
  if (p->s.size >= nunits) {
    /* Do stuff on previous slide */
    …
    freep = prevp; /* move the head */
    return (void *) (p+1);
  }
}
```


Oops, No Block is Big Enough!

- Cycling completely through the list
  - Check if the "for" loop returns back to the head of the list

```
prevp = freep; /* start at beginning */
for (p=prevp->s.ptr; ; prevp=p,
                       p=p->s.ptr) {
  if (p->s.size >= nunits) {
     /* Do stuff on previous slides */
     …
  }
  if (p == freep) /* wrapped around */
    Now, do something about it…
}
```

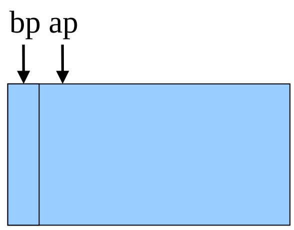

What to Do When You Run Out

- Ask the operating system for additional memory
  - Ask for a very large chunk of memory
  - … and insert the new chunk into the free list
  - ... and then try again, this time successfully
- Operating-system dependent
  - E.g., **sbrk** command in UNIX
  - See the **morecore()** function for details

```
if (p == freep) /* wrapped around */
   if ((p = morecore(nunits)) == NULL)
      return NULL;   /* none left */
```

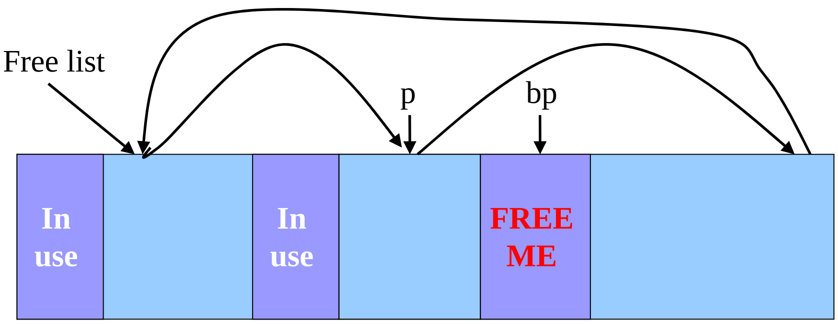
Free

- User passes a pointer to the memory block
  - `void free(void *ap);`
- Free function inserts block into the list
  - Identify the start of entry: `bp = (Header *) ap – 1;`
  - Find the location in the free list
  - Add to the list, coalescing entries, if needed



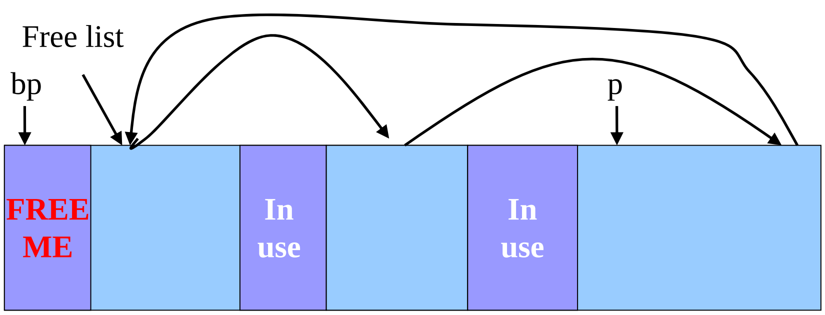
Scanning Free List for the Spot

- Start at the beginning: **p = freep;**
- Sequence through the list: **p = p->s.ptr;**
- Stop at last entry before the to-be-freed element
  - `(bp > p) && (bp < p->s.ptr);`



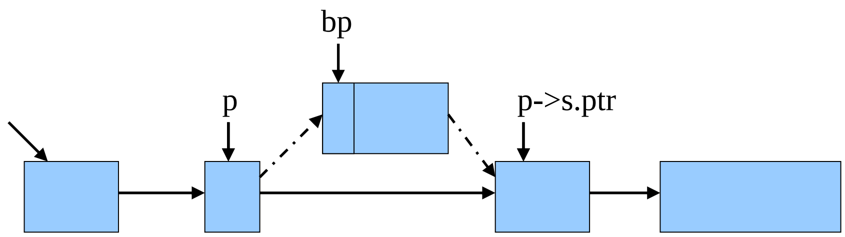
Corner Cases: Beginning or End

- Check for wrap-around in memory
  - `p >= p->s.ptr;`
- See if to-be-freed element is located there
  - `(bp > p) || (bp < p->s.ptr);`



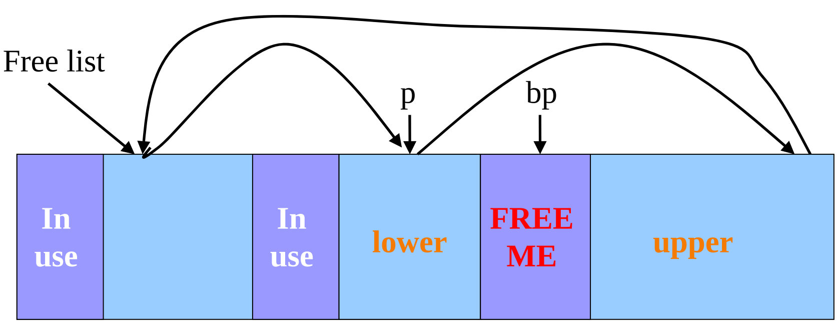
Inserting Into Free List

- New element to add to free list: **bp**
- Insert in between **p** and **p->s.ptr**
  - **bp->s.ptr = p->s.ptr;**
  - **p->s.ptr = bp;**
- But, there may be opportunities to coalesce



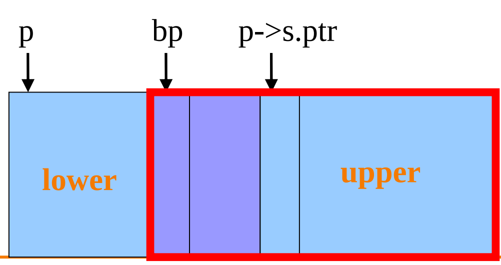
Coalescing With Neighbors

- Scanning the list finds the location for inserting
  - Pointer to to-be-freed element: **bp**
  - Pointer to previous element in free list: **p**
- Coalescing into larger free blocks
  - Check if contiguous to upper and lower neighbors



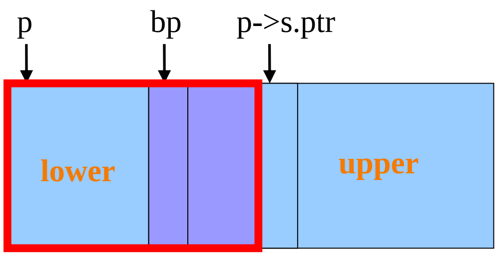
Coalesce With Upper Neighbor

- Check if next part of memory is in the free list
  - **if (bp + bp->s.size == p->s.ptr)**
- If so, make into one bigger block
  - Larger size: **bp->s.size += p->s.ptr->s.size;**
  - Copy next pointer: **bp->s.ptr = p->s.ptr->s.ptr;**
- Else, simply point to the next free element
  - **bp->s.ptr = p->s.ptr;**



Coalesce With Lower Neighbor

- Check if previous part of memory is in the free list
  - `if (p + p->s.size == bp)`
- If so, make into one bigger block
  - Larger size: `p->s.size += bp->s.size;`
  - Copy next pointer: `p->s.ptr = bp->s.ptr;`



Conclusions

- Elegant simplicity of K&R malloc and free
  - Simple header with pointer and size in each free block
  - Simple linked list of free blocks
  - Relatively small amount of code (~25 lines each)
- Limitations of K&R functions in terms of efficiency
  - Malloc requires scanning the free list
    - To find the first free block that is big enough
  - Free requires scanning the free list
    - To find the location to insert the to-be-freed block
- Next lecture, and programming assignment #4
  - Making malloc and free more efficient