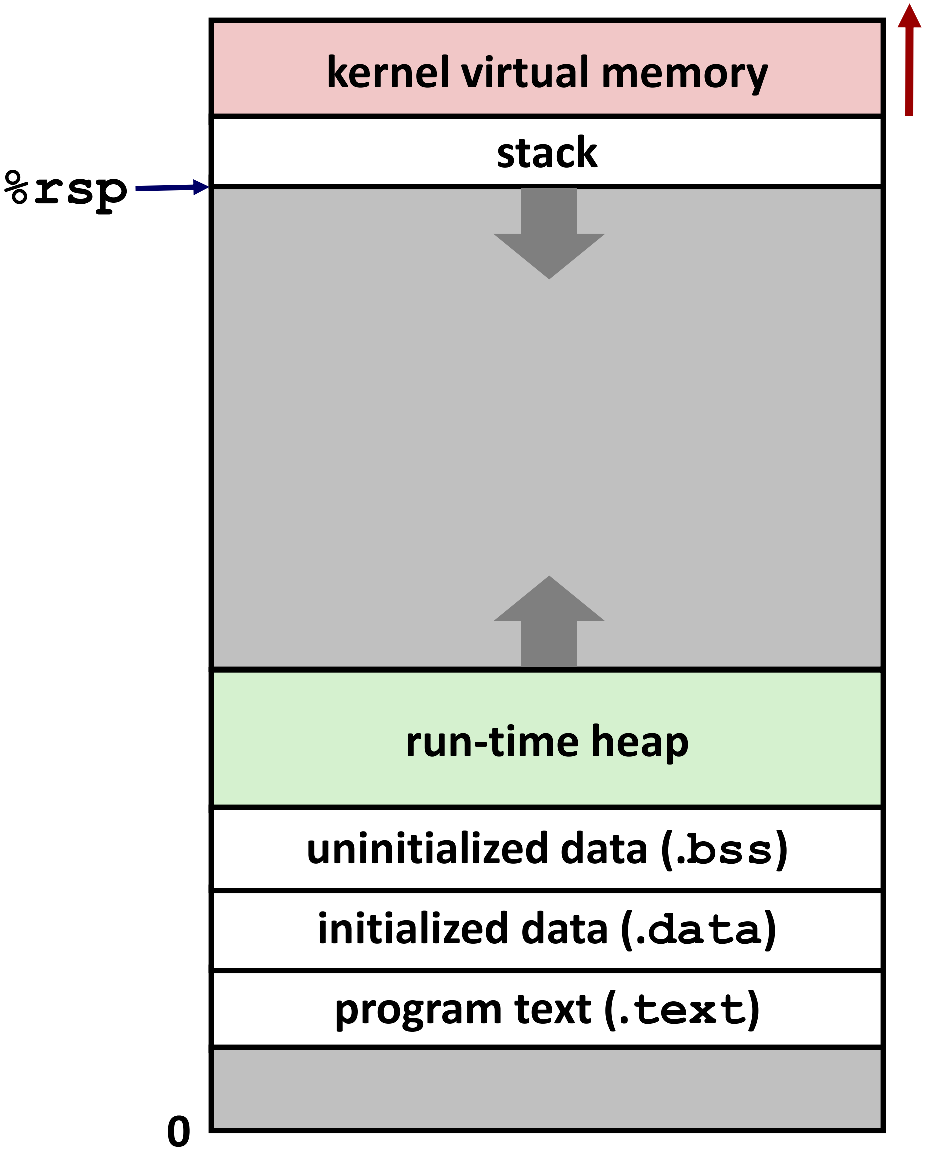
CSE 351
Section 8

2/22/12

Agenda

- Malloc/Free

Process Memory Image



memory protected
from user code

kernel virtual memory

stack

`%rsp`

run-time heap

uninitialized data (`.bss`)

initialized data (`.data`)

program text (`.text`)

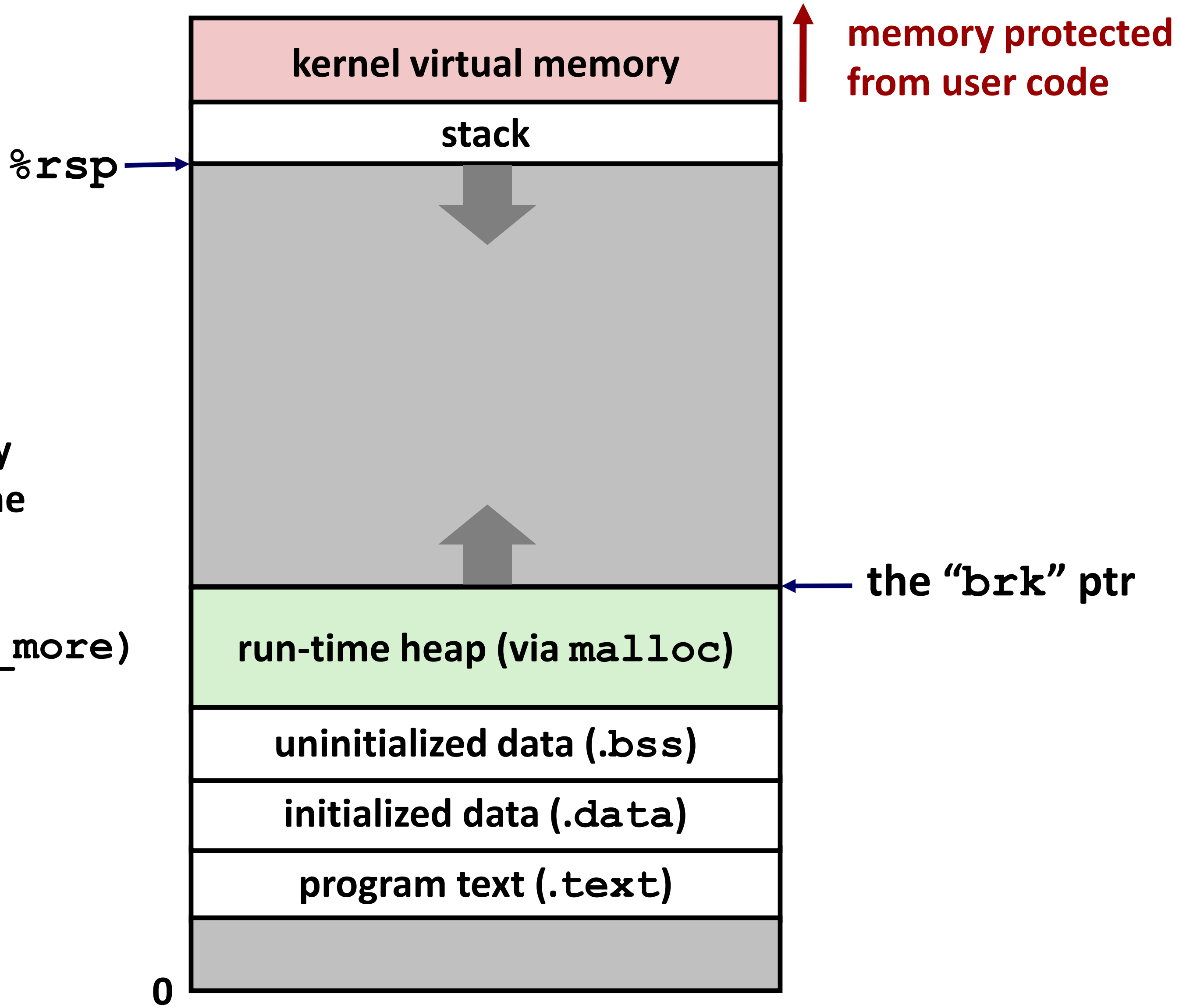
0

What is the heap for?
How do we use it?

Memory Allocation

- Dynamic memory allocation
  - Size of data structures may only be known at run time
  - Need to allocate space on the heap
  - Need to de-allocate (free) unused memory so it can be re-allocated
- Implementation --- “Memory allocator”

Process Memory Image

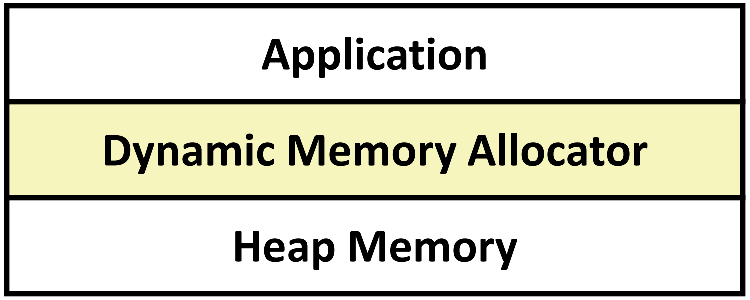


Allocators request additional **heap memory from the kernel using the `sbrk()` function:**

```
error = sbrk(amt_more)
```

Dynamic Memory Allocation

- Memory allocator?
  - VM hardware and kernel allocate pages
  - Application objects are typically smaller
  - Allocator manages objects within pages



Application

Dynamic Memory Allocator

Heap Memory

- Explicit vs. Implicit Memory Allocator
  - ***Explicit:*** application allocates and frees space
    - In C: **`malloc()`** and **`free()`**
  - ***Implicit:*** application allocates, but does not free space
    - In Java, ML, Lisp: garbage collection
- Allocation
  - A memory allocator doles out memory blocks to application
  - A "block" is a contiguous range of bytes of the appropriate size
    - What is appropriate size?

Malloc Package

- `#include <stdlib.h>`
- `void *malloc(size_t size)`
  - Successful:
    - Returns a pointer to a memory block of at least **`size`** bytes (typically) aligned to 8-byte boundary
    - If **`size == 0`**, returns NULL
  - Unsuccessful: returns NULL (0) and sets errno (a global variable)
- `void free(void *p)`
  - Returns the block pointed at by **`p`** to the pool of available memory
  - **`p`** must come from a previous call to **`malloc`** or **`realloc`**
- `void *realloc(void *p, size_t size)`
  - Changes size of block **`p`** and returns pointer to new block
  - Contents of new block unchanged up to min of old and new size
  - Old block has been **`free`**'d (logically, if new != old)

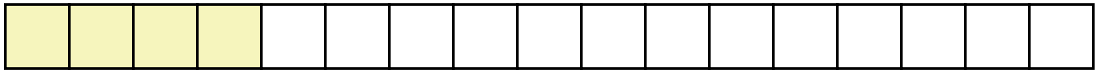
Malloc Example

```
void foo(int n, int m) {
  int i, *p;

  /* allocate a block of n ints */
  p = (int *)malloc(n * sizeof(int));
  if (p == NULL) {
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++) p[i] = i;

  /* add m bytes to end of p block */
  if ((p = (int *)realloc(p, (n+m) * sizeof(int))) == NULL) {
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++) p[i] = i;

  /* print new array */
  for (i=0; i<n+m; i++)
    printf("%d\n", p[i]);

  free(p); /* return p to available memory pool */
}
```

Why?

Allocation Example

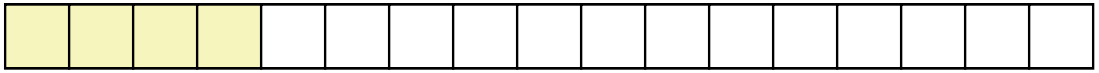
`p1 = malloc(4)`



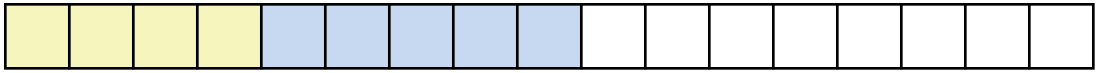
`p2 = malloc(5)`

Allocation Example

`p1 = malloc(4)`

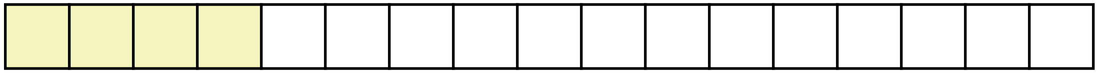


`p2 = malloc(5)`

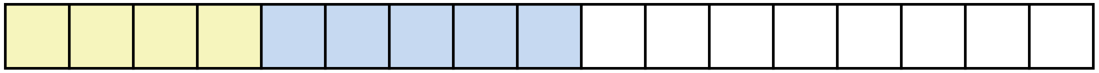


Allocation Example

`p1 = malloc(4)`



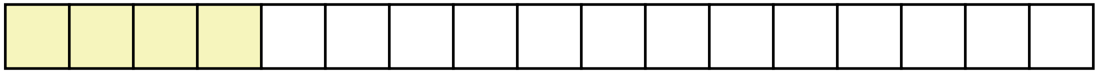
`p2 = malloc(5)`



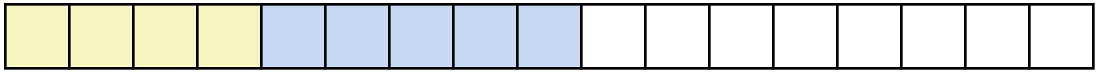
`p3 = malloc(6)`

Allocation Example

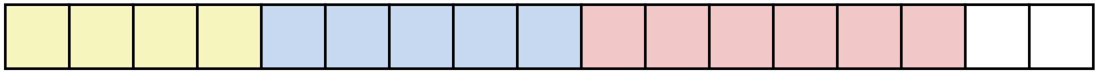
```
p1 = malloc(4)
```



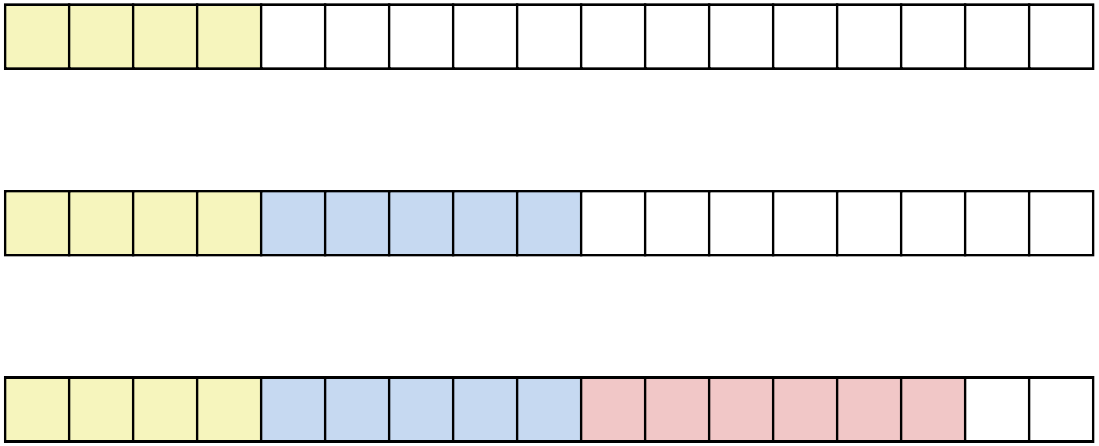
```
p2 = malloc(5)
```



```
p3 = malloc(6)
```



Allocation Example



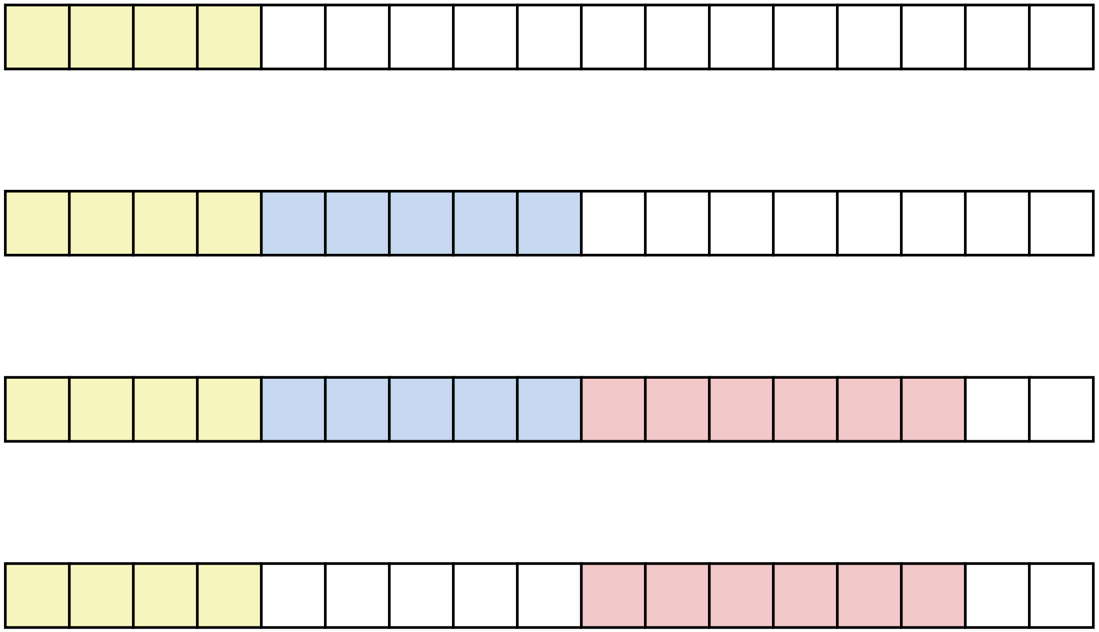
```
p1 = malloc(4)
```

```
p2 = malloc(5)
```

```
p3 = malloc(6)
```

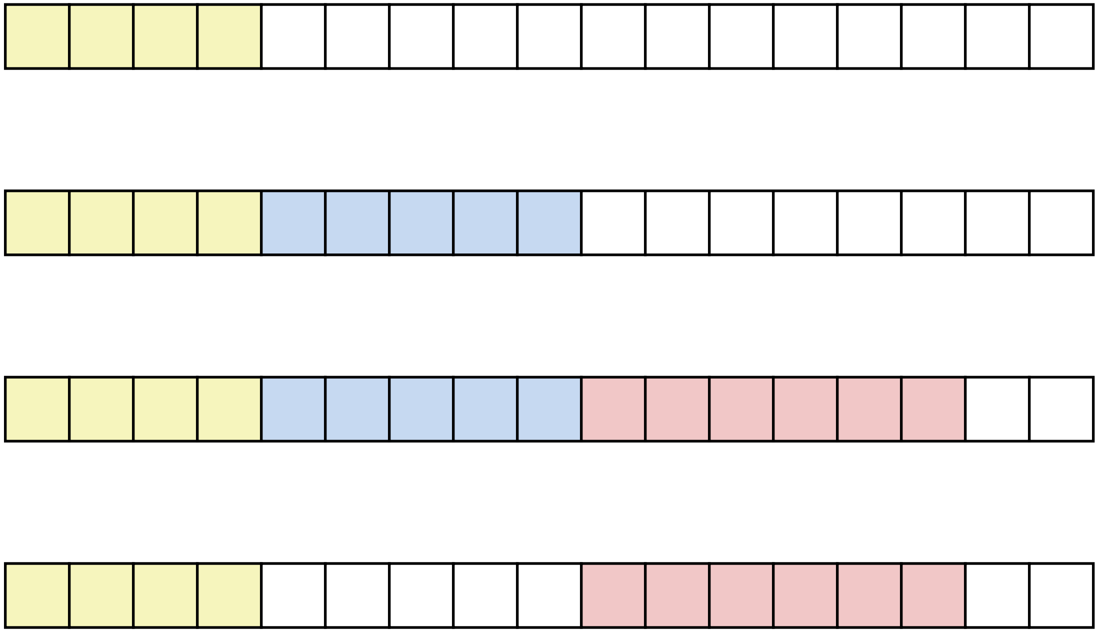
```
free(p2)
```

Allocation Example



```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)
```

Allocation Example



`p1 = malloc(4)`

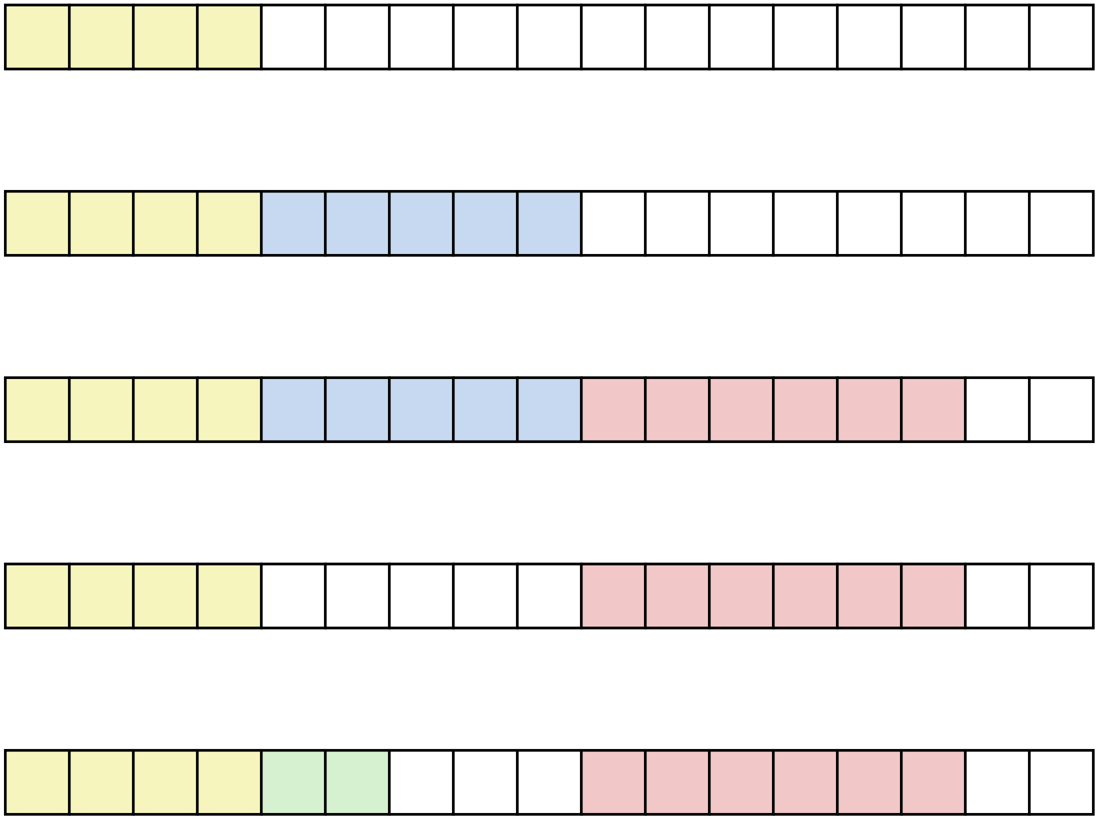
`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(2)`

Allocation Example



```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(2)
```

Constraints

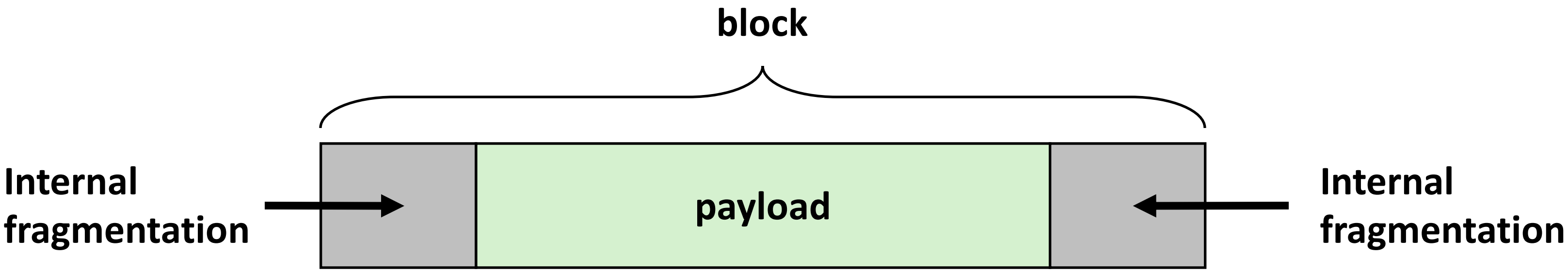
- Applications
  - Can issue arbitrary sequence of malloc() and free() requests
  - free() requests must be to a malloc()'d block
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to malloc() requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8 byte alignment for GNU malloc (**`libc`** malloc) on Linux boxes
  - Can manipulate and modify only free memory
  - Can't move the allocated blocks once they are malloc()'d
    - *i.e.*, compaction is not allowed. *Why not?*

Fragmentation

- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation
- Terminology
  - Block
    - The chunk of memory malloc reserves for a given malloc call
  - Payload
    - `malloc(p)` results in a block with a *payload* of p bytes

Internal Fragmentation

- For a given block, *internal fragmentation* occurs if payload is smaller than block size



- Caused by
  - overhead of maintaining heap data structures (inside block, outside payload)
  - padding for alignment purposes
  - explicit policy decisions
    (e.g., to return a big block to satisfy a small request)
- Depends only on the pattern of *previous* requests
  - Thus, easy to measure

External Fragmentation

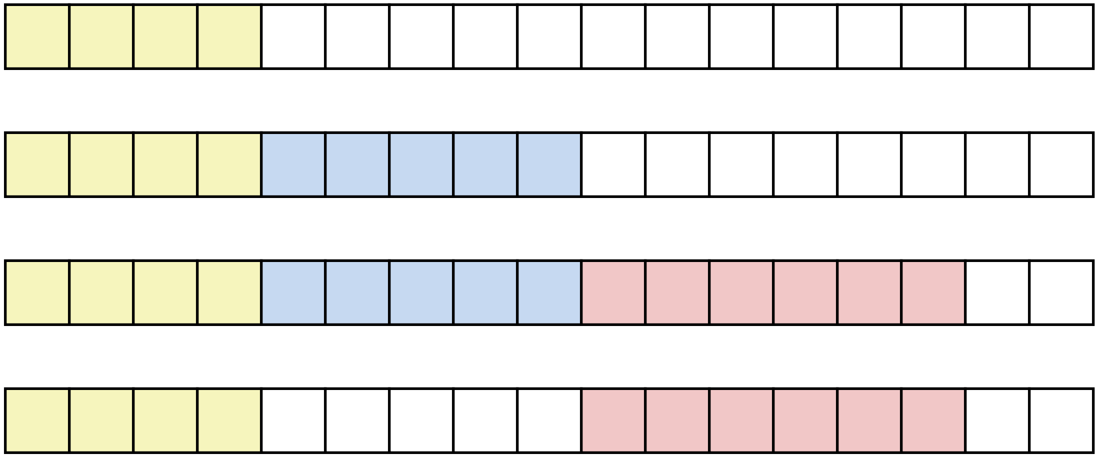
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

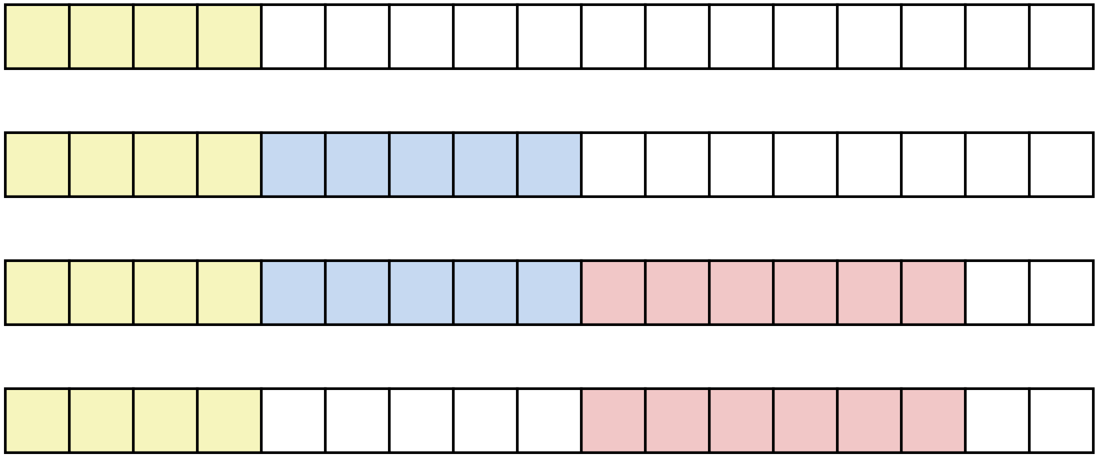
`free(p2)`



External Fragmentation

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

```
p1 = malloc(4)

p2 = malloc(5)

p3 = malloc(6)

free(p2)

p4 = malloc(6)
```



External Fragmentation

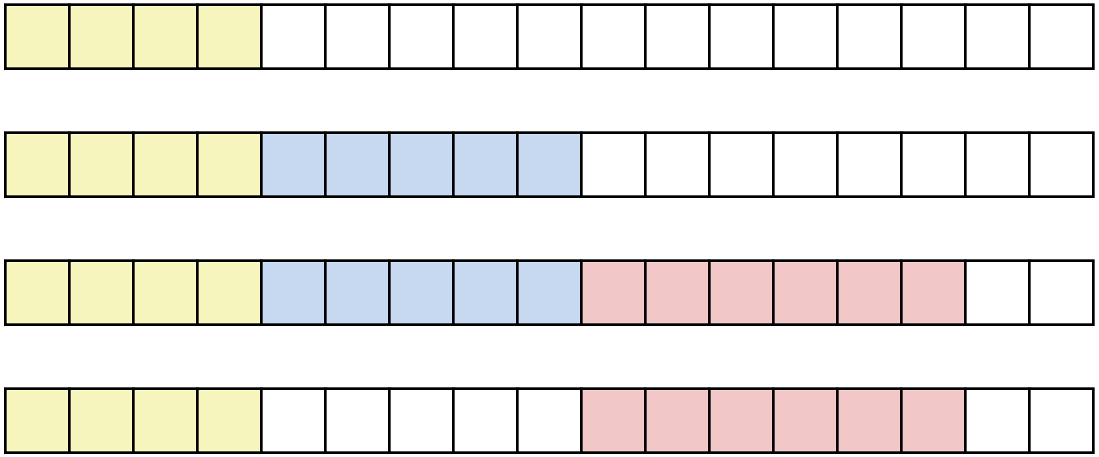
- Occurs when there is enough aggregate heap memory, but no single free block is large enough

`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

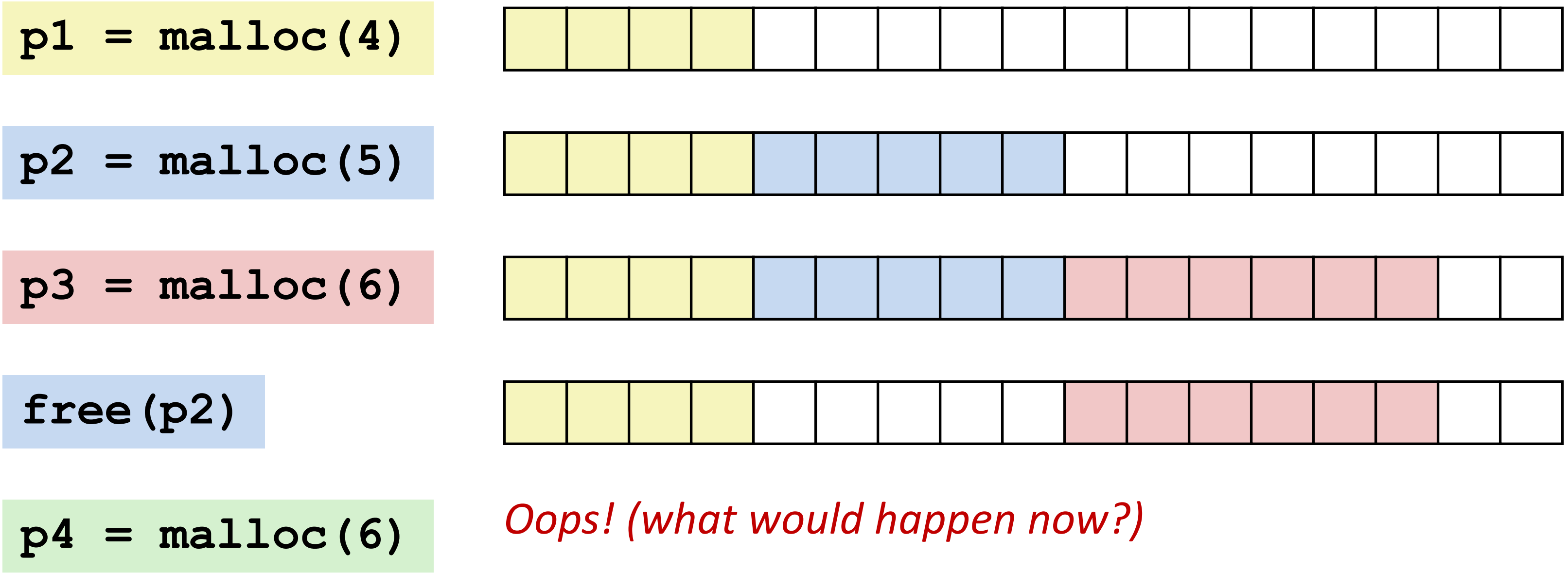


`p4 = malloc(6)`

Oops! (what would happen now?)

External Fragmentation

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



`p1 = malloc(4)`

`p2 = malloc(5)`

`p3 = malloc(6)`

`free(p2)`

`p4 = malloc(6)`

Oops! (what would happen now?)

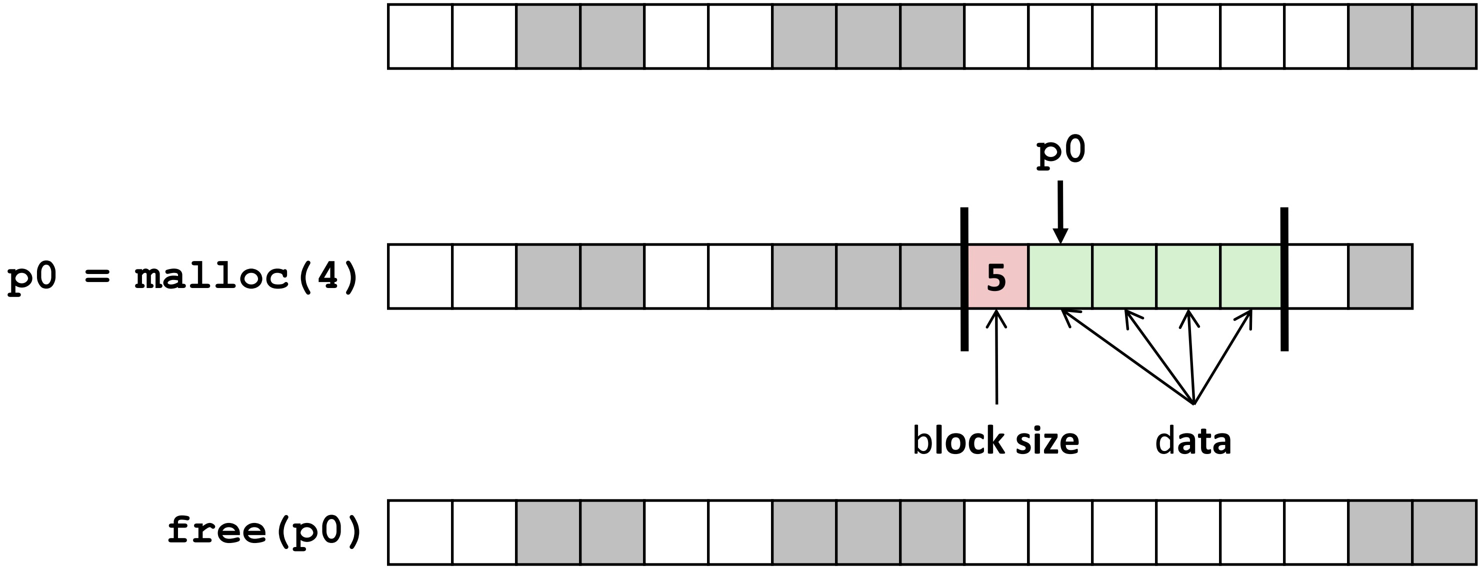
- Depends on the pattern of future requests
  - Thus, difficult to measure

Implementation Issues

- How to know how much memory is being `free()`’d when it is given only a pointer (and no length)?
- How to keep track of the free blocks?
- What to do with extra space when allocating a block that is smaller than the free block it is placed in?
- How to pick a block to use for allocation—many might fit?
- How to reinsert a freed block into the heap?

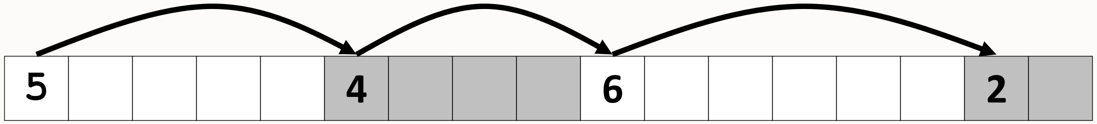
Knowing How Much to Free

- Standard method
  - Keep the length of a block in the word preceding the block.
    - This word is often called the ***header field*** or ***header***
  - Requires an extra word for every allocated block

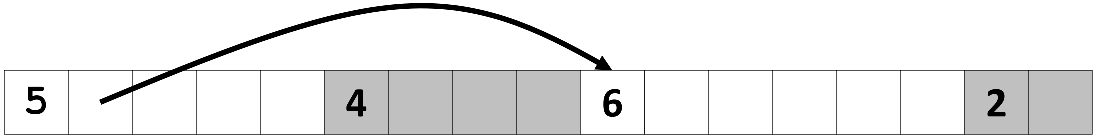


Keeping Track of Free Blocks

- Method 1: *Implicit list* using length—links all blocks



- Method 2: *Explicit list* among the free blocks using pointers



- Method 3: *Segregated free list*
  - Different free lists for different size classes

- Method 4: *Blocks sorted by size*
  - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key