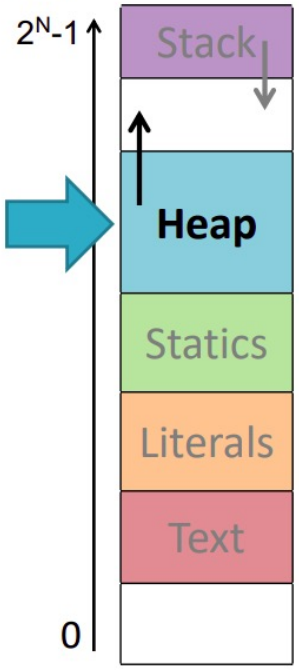
Dynamic Memory Allocation: Basic Concepts

Today

- Basic concepts
- Implicit free lists

Memory Layout

Heap Allocation



Perm	Contents	Managed by	Initialized
RW	Procedure context	Compiler	Run-time
RW	**Dynamic data structures**	Programmer, **malloc/free,** new/GC	**Run-time**
RW	Global variables/ static data structures	Compiler/ Assembler/Linker	Startup
R	String literals	Compiler/ Assembler/Linker	Startup
X	Instructions	Compiler/ Assembler/Linker	Startup

Dynamic Memory Allocation

- Allocator maintains heap as collection of variable sized *blocks*, which are either *allocated* or *free*
- Types of allocators
  - ***Explicit allocator*:** application allocates and frees space
    - E.g., `malloc` and `free` in C
  - ***Implicit allocator:*** application allocates, but does not free space
    - E.g. garbage collection in Java, and Lisp

- Will discuss simple explicit memory allocation today

The `malloc` Package

```
#include <stdlib.h>

void* *malloc(size_t size)
```

- Successful:
  - Returns a pointer to a memory block of at least **`size`** bytes aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
  - If **`size == 0`**, returns NULL
- Unsuccessful: returns NULL (0) and sets **`errno`**

```
void* free(void *p)
```

- Returns the block pointed at by **`p`** to pool of available memory
- **`p`** must come from a previous call to **`malloc`** or **`realloc`**

Other functions

- **`calloc`**: Version of **`malloc`** that initializes allocated block to zero.
- **`realloc`**: Changes the size of a previously allocated block.
- **`sbrk`**: Used internally by allocators to grow or shrink the heap

Example (Anti Pattern)

```
#include <stdio.h>
#include <stdlib.h>
#define MAXN 15213

int array[MAXN]

int main(){
      int i, n;
      scanf("%d, &n);
      If (n > MAXN){
           app_error("Input file too big");
      }
      for( i = 0; i <n; i++){
         scanf("%d". &array[i]);
       }


}
```

Example (Anti Pattern)

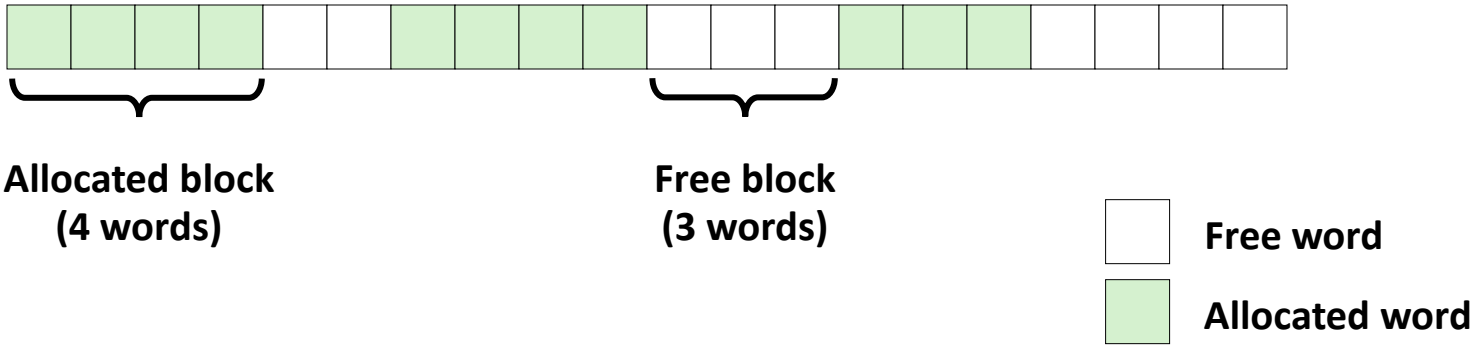
```
#include <stdio.h>
#include <stdlib.h>

int main(){
      Int *array, i, n;
      scanf("%d, &n);
      array =  (int *) Malloc(n*sizeof(int));
      for( i = 0; i <n; i++){
         scanf("%d". &array[i]);
       }


}
```

Assumptions Made in This Lecture

- Memory is word addressed.
- Words are int-sized.



`malloc` Example

```c
#include <stdio.h>
#include <stdlib.h>

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

    /* Allocate a block of n ints */
    p = (int *) malloc(n * sizeof(int));
    if (p == NULL) {
        perror("malloc");
        exit(0);
    }

    /* Initialize allocated block */
    for (i=0; i<n; i++)
        p[i] = i;


    /* Return allocated block to the heap */
    free(p);
}
```

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`** request must be to a **`malloc`**'d block

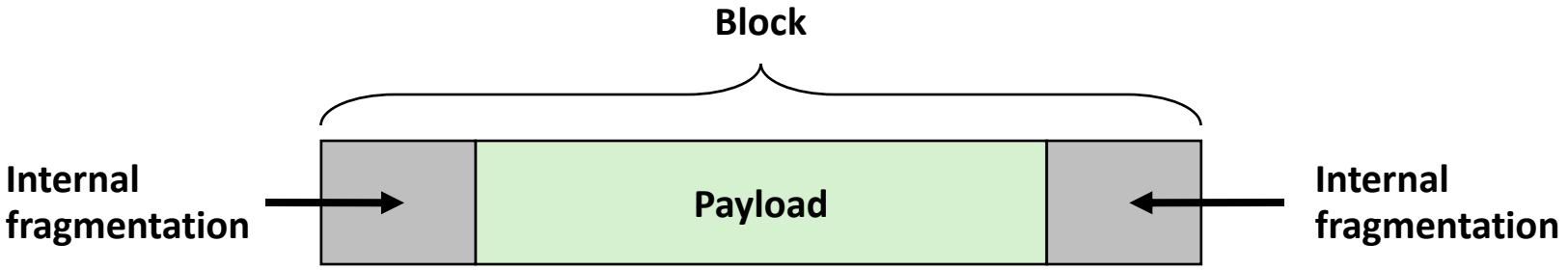
- Allocators
  - Can't control number or size of allocated blocks
  - Must respond immediately to **`malloc`** requests
    - *i.e.*, can't reorder or buffer requests
  - Must allocate blocks from free memory
    - *i.e.*, can only place allocated blocks in free memory
  - Must align blocks so they satisfy all alignment requirements
    - 8-byte (x86) or 16-byte (x86-64) alignment 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

Fragmentation

- Poor memory utilization caused by *fragmentation*
  - ***internal*** fragmentation
  - ***external*** fragmentation

Internal Fragmentation

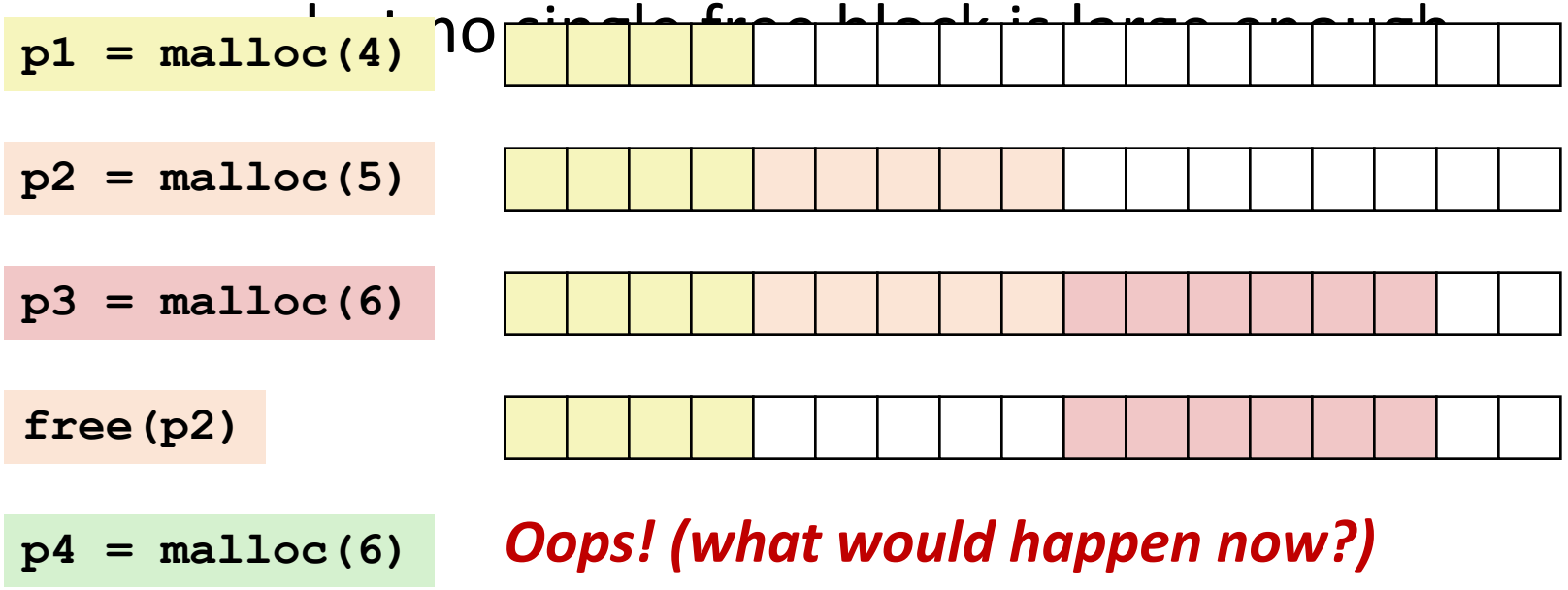
- For a given block, *internal fragmentation* occurs if payload is smaller than block size



- Caused by
  - Overhead of maintaining heap data structures
  - 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

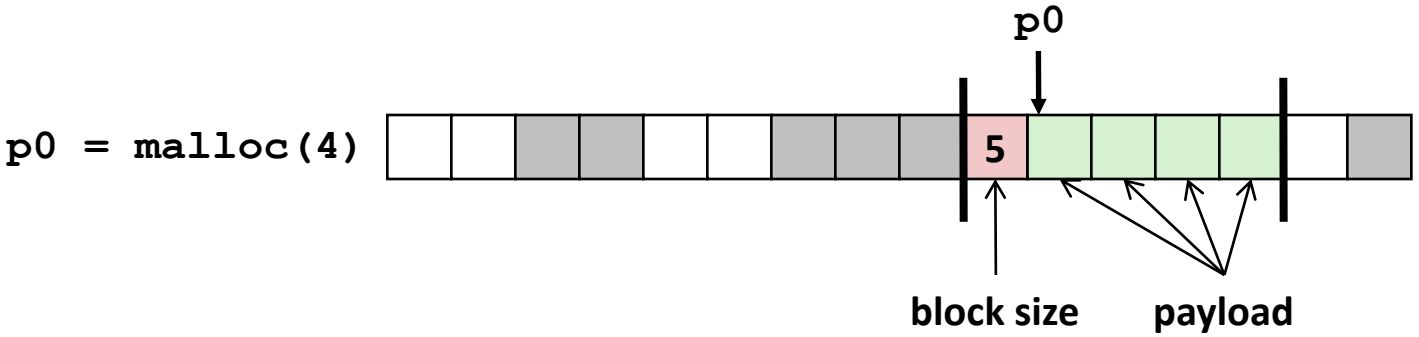


Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- What do we do with the extra space when allocating a structure that is smaller than the free block it is placed in?
- How do we pick a block to use for allocation -- many might fit?

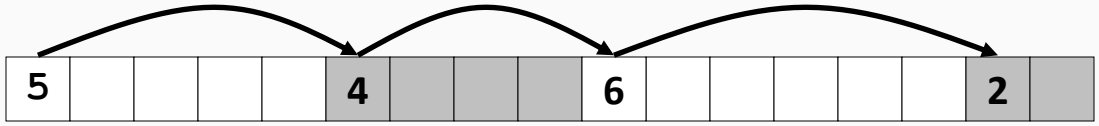
Knowing How Much to Free

- 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



Keeping Track of Free Blocks

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



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