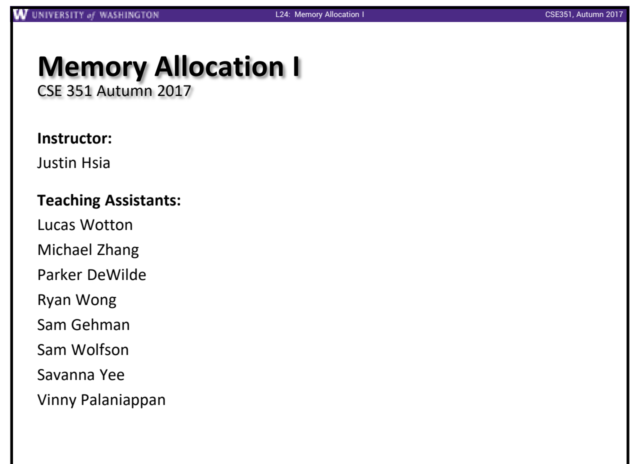
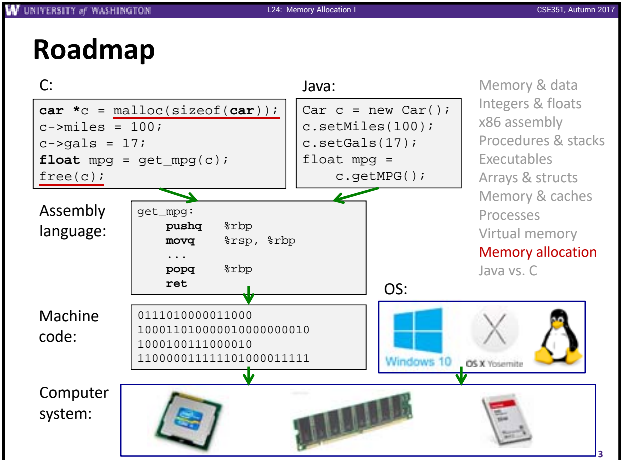


Administrivia

- Lab 4 due tonight
- Homework 5 due Friday
- Lab 5 (on Mem Alloc) released tomorrow, due 12/8
- **Final Exam:** Wed, 12/13, 12:30-2:20pm in KNE 120
  - Same seating arrangements as last time
  - Review Session: Mon, 12/11, 5-8pm in EEB 105
  - Cumulative (midterm clobber policy applies)
  - You get TWO double-sided handwritten 8.5×11" cheat sheets
    - Recommended that you reuse or remake your midterm cheat sheet

2



Multiple Ways to Store Program Data

```
int array[1024];

void foo(int n) {
  int tmp;
  int local_array[n];

  int* dyn =
    (int*)malloc(n*sizeof(int));
}
```

- Static global data
  - *Fixed size* at compile-time
  - Entire *lifetime of the program* (loaded from executable)
  - Portion is read-only (e.g. string literals)
- Stack-allocated data
  - Local/temporary variables
    - *Can* be dynamically sized (in some versions of C)
  - *Known lifetime* (deallocated on `return`)
- **Dynamic (heap) data**
  - Size known only at runtime (i.e. based on user-input)
  - Lifetime known only at runtime (long-lived data structures)

4

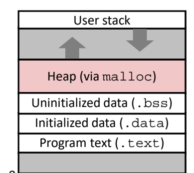
Memory Allocation

- **Dynamic memory allocation**
  - **Introduction and goals**
  - **Allocation and deallocation (free)**
  - **Fragmentation**
- Explicit allocation implementation
  - Implicit free lists
  - Explicit free lists (Lab 5)
  - Segregated free lists
- Implicit deallocation: garbage collection
- Common memory-related bugs in C

5

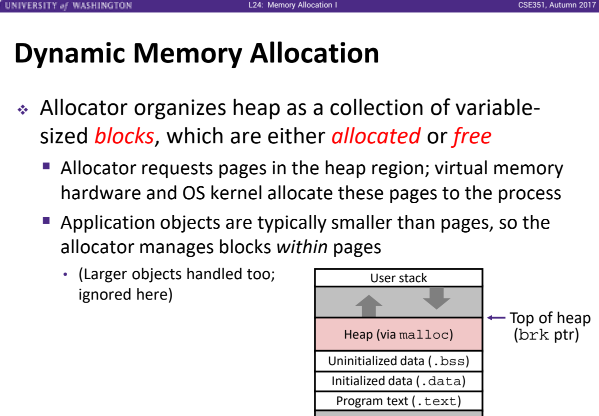
Dynamic Memory Allocation

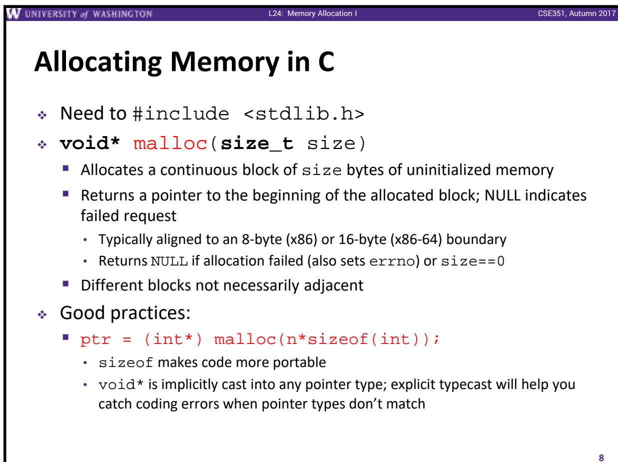
- Programmers use dynamic memory allocators to acquire virtual memory at run time
  - For data structures whose size (or lifetime) is known only at runtime
  - Manage the heap of a process' virtual memory:



- Types of allocators
  - ***Explicit* allocator:** programmer allocates and frees space
    - Example: `malloc` and `free` in C
  - ***Implicit* allocator:** programmer only allocates space (no free)
    - Example: garbage collection in Java, Caml, and Lisp

6





Allocating Memory in C

- Need to `#include <stdlib.h>`
- `void* malloc(size_t size)`
  - Allocates a continuous block of `size` bytes of uninitialized memory
  - Returns a pointer to the beginning of the allocated block; NULL indicates failed request
    - Typically aligned to an 8-byte (x86) or 16-byte (x86-64) boundary
    - Returns `NULL` if allocation failed (also sets `errno`) or `size==0`
  - Different blocks not necessarily adjacent
- Related functions:
  - `void* calloc(size_t nitems, size_t size)`
    - "Zeros out" allocated block
  - `void* realloc(void* ptr, size_t size)`
    - Changes the size of a previously allocated block (if possible)
  - `void* sbrk(intptr_t increment)`
    - Used internally by allocators to grow or shrink the heap

Freeing Memory in C

- Need to `#include <stdlib.h>`
- `void free(void* p)`
  - Releases whole block pointed to by `p` to the pool of available memory
  - Pointer `p` must be the address *originally* returned by `m/c/realloc` (i.e. beginning of the block), otherwise throws system exception
  - Don't call `free` on a block that has already been released or on `NULL`

Memory Allocation Example in C

```
void foo(int n, int m) {
  int i, *p;
  p = (int*) malloc(n*sizeof(int)); /* allocate block of n ints */
  if (p == NULL) {                  /* check for allocation error */
    perror("malloc");
    exit(0);
  }
  for (i=0; i<n; i++)               /* initialize int array */
    p[i] = i;
                                    /* add space for m ints to end of p block */
  p = (int*) realloc(p,(n+m)*sizeof(int));
  if (p == NULL) {                  /* check for allocation error */
    perror("realloc");
    exit(0);
  }
  for (i=n; i < n+m; i++)           /* initialize new spaces */
    p[i] = i;
  for (i=0; i<n+m; i++)             /* print new array */
    printf("%d\n", p[i]);
  free(p);                          /* free p */
}
```


Notation

= one box, 4 bytes

- We will draw memory divided into *boxes*
  - Each *box* can hold an `int` (32 bits/4 bytes)
  - Allocations will be in sizes that are a multiple of boxes, i.e. multiples of 4 bytes
  - Book and old videos use *word* instead of *box*
    - Holdover from 32-bit version of textbook ☹

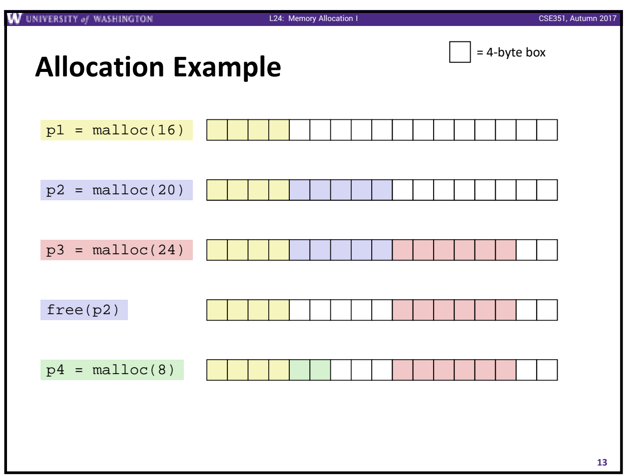
Allocated block (4 boxes)

Free block (3 boxes)

Free box

Allocated box

Allocation Example



Implementation Interface

- **Applications**
  - Can issue arbitrary sequence of `malloc` and `free` requests
  - Must never access memory not currently allocated
  - Must never free memory not currently allocated
    - Also must only use `free` with previously `malloc`'ed blocks
- **Allocators**
  - Can't control number or size of allocated blocks
  - Must respond immediately to `malloc`
  - Must allocate blocks from free memory
  - Must align blocks so they satisfy all alignment requirements
  - Can't move the allocated blocks

Performance Goals

- **Goals:** Given some sequence of `malloc` and `free` requests $R_0, R_1, \dots, R_k, \dots, R_{n-1}$, maximize throughput and peak memory utilization
  - These goals are often conflicting

1) Throughput
- Number of completed requests per unit time
- Example:
  - If 5,000 `malloc` calls and 5,000 `free` calls completed in 10 seconds, then throughput is 1,000 operations/second

Performance Goals

- Definition: *Aggregate payload* P_k
  - `malloc(p)` results in a block with a *payload* of `p` bytes
  - After request R_k has completed, the *aggregate payload* P_k is the sum of currently allocated payloads
- Definition: *Current heap size* H_k
  - Assume H_k is monotonically non-decreasing
    - Allocator can increase size of heap using `sbrk`

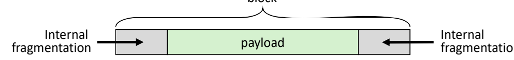
2) Peak Memory Utilization
- Defined as $U_k = (\max_{i \le k} P_i)/H_k$ after k+1 requests
- Goal: maximize utilization for a sequence of requests
- Why is this hard? And what happens to throughput?

Fragmentation

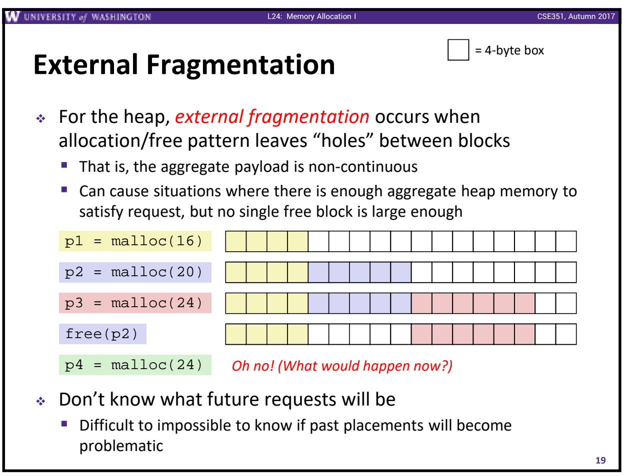
- Poor memory utilization is caused by *fragmentation*
  - Sections of memory are not used to store anything useful, but cannot satisfy allocation requests
  - Two types: *internal* and *external*
- **Recall:** Fragmentation in structs
  - Internal fragmentation was wasted space *inside* of the struct (between fields) due to alignment
  - External fragmentation was wasted space *between* struct instances (e.g. in an array) due to alignment
- Now referring to wasted space in the heap *inside* or *between* allocated blocks

Internal Fragmentation

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



- **Causes:**
  - Padding for alignment purposes
  - Overhead of maintaining heap data structures (inside block, outside payload)
  - Explicit policy decisions (e.g. return a big block to satisfy a small request)
- Easy to measure because only depends on past requests



Peer Instruction Question

- Which of the following statements is FALSE?
  - Vote at http://PollEv.com/justinh

  A. Temporary arrays should *not* be allocated on the Heap

  B. `malloc` returns an address filled with garbage

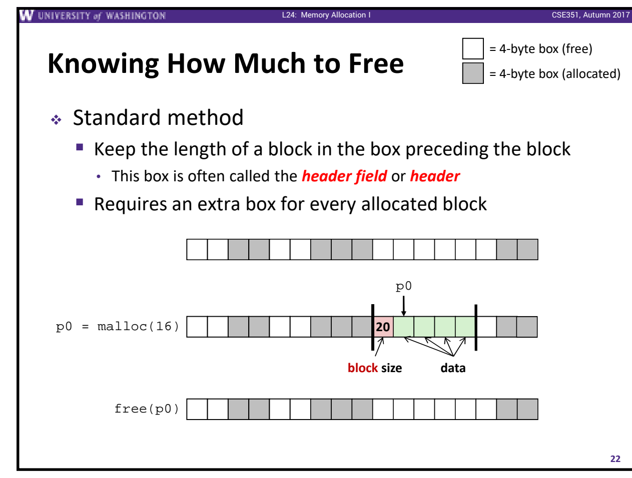
  C. Peak memory utilization is a measure of both internal and external fragmentation

  D. An allocation failure will cause your program to stop

  E. We're lost…

Implementation Issues

- How do we know how much memory to free given just a pointer?
- How do we keep track of the free blocks?
- How do we pick a block to use for allocation (when many might fit)?
- 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 reinsert a freed block into the heap?



Keeping Track of Free Blocks

= 4-byte box (free)
= 4-byte box (allocated)

1) *Implicit free list* using length – links all blocks using math
   - No actual pointers, and must check each block if allocated or free

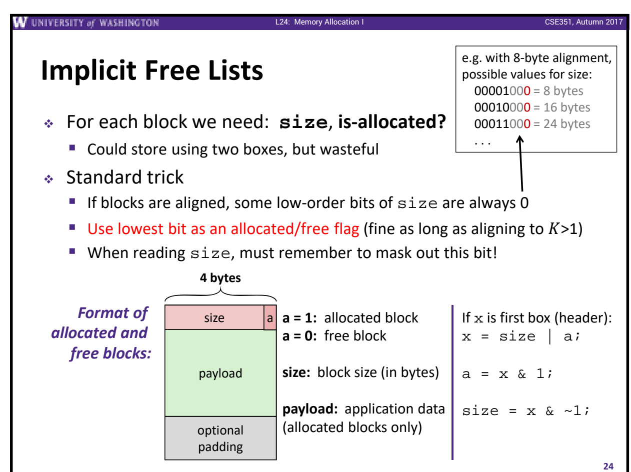
   20 | 16 | 24 | 8

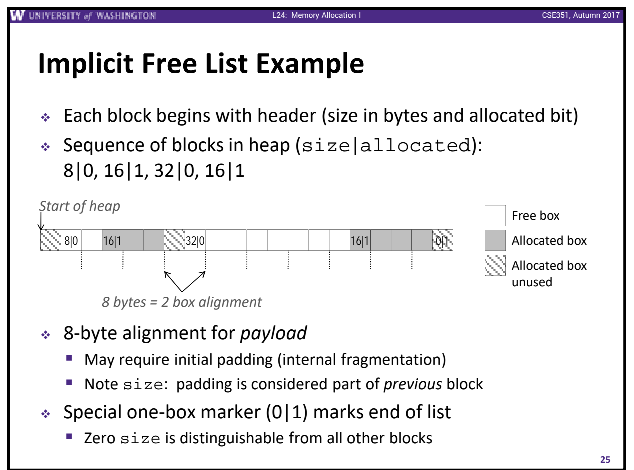
2) *Explicit free list* among only the free blocks, using pointers

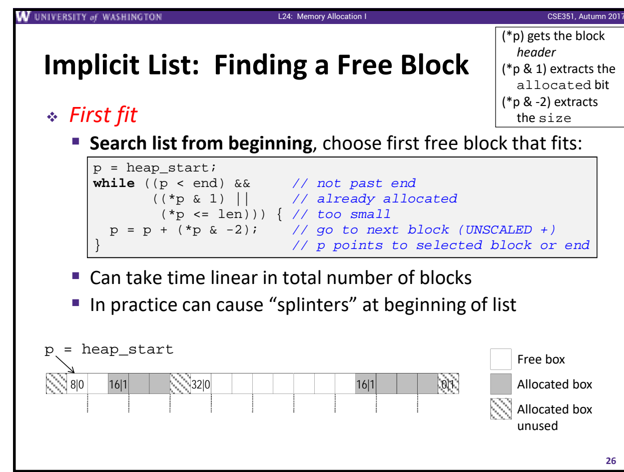
   20 | 16 | 24 | 8

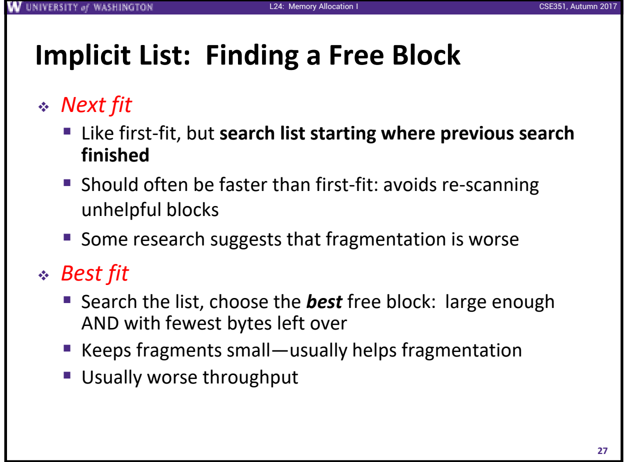
3) *Segregated free list*
   - Different free lists for different size "classes"

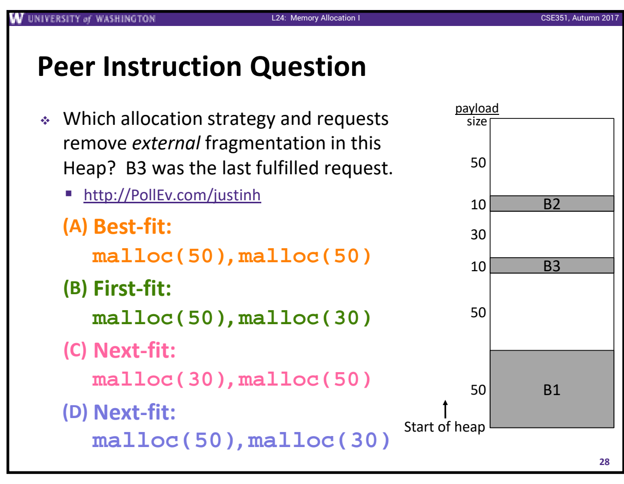
4) *Blocks sorted by size*
   - Can use a balanced binary tree (e.g. red-black tree) with pointers within each free block, and the length used as a key




UNIVERSITY of WASHINGTON
L24: Memory Allocation I
CSE351, Autumn 2017
Implicit Free List Example
Each block begins with header (size in bytes and allocated bit)
Sequence of blocks in heap (size|allocated):
8|0, 16|1, 32|0, 16|1
Start of heap
8|0
16|1
32|0
16|1
0|1
Free box
Allocated box
Allocated box unused
8 bytes = 2 box alignment
8-byte alignment for payload
May require initial padding (internal fragmentation)
Note size: padding is considered part of previous block
Special one-box marker (0|1) marks end of list
Zero size is distinguishable from all other blocks
25


UNIVERSITY of WASHINGTON
L24: Memory Allocation I
CSE351, Autumn 2017
Implicit List: Finding a Free Block
(*p) gets the block header
(*p & 1) extracts the allocated bit
(*p & -2) extracts the size
First fit
Search list from beginning, choose first free block that fits:
p = heap_start;
while ((p < end) && // not past end
((*p & 1) || // already allocated
(*p <= len))) { // too small
p = p + (*p & -2); // go to next block (UNSCALED +)
} // p points to selected block or end
Can take time linear in total number of blocks
In practice can cause "splinters" at beginning of list
p = heap_start
8|0
16|1
32|0
16|1
0|1
Free box
Allocated box
Allocated box unused
26


UNIVERSITY of WASHINGTON
L24: Memory Allocation I
CSE351, Autumn 2017
Implicit List: Finding a Free Block
Next fit
Like first-fit, but search list starting where previous search finished
Should often be faster than first-fit: avoids re-scanning unhelpful blocks
Some research suggests that fragmentation is worse
Best fit
Search the list, choose the best free block: large enough AND with fewest bytes left over
Keeps fragments small—usually helps fragmentation
Usually worse throughput
27


UNIVERSITY of WASHINGTON
L24: Memory Allocation I
CSE351, Autumn 2017
Peer Instruction Question
Which allocation strategy and requests remove external fragmentation in this Heap? B3 was the last fulfilled request.
http://PollEv.com/justinh
(A) Best-fit:
malloc(50),malloc(50)
(B) First-fit:
malloc(50),malloc(30)
(C) Next-fit:
malloc(30),malloc(50)
(D) Next-fit:
malloc(50),malloc(30)
payload size
50
10
B2
30
10
B3
50
50
B1
Start of heap
28