Dynamic Memory: Alignment and Fragmentation

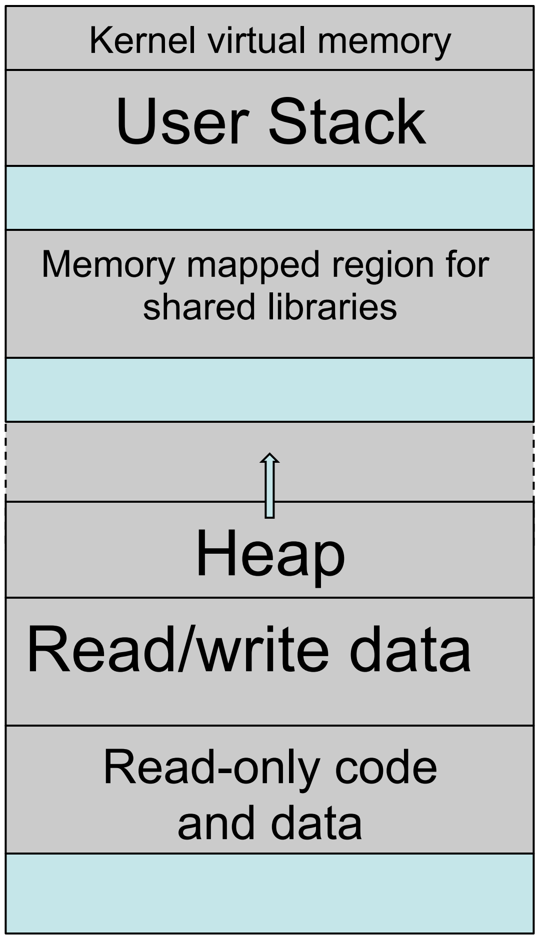
- Learning Objectives
  - Explain the purpose of dynamic memory
  - Define the terms arena, heap
  - Identify common errors involving dynamic memory
  - Explain how dynamic memory allocations are aligned
  - Explain internal and external fragmentation
  - Complete assignment 1!

Different types of memory

- In C, you can allocate variables/memory in several different ways:
  - Globals: Accessible from anywhere in your program; typically allocated in a read/write data segment of your process.
  - Global const: Global variables whose values cannot change; typically allocated in a read-only data segment.
  - Locals (regular): Variables private to a function; cannot be accessed by other functions; allocated on the stack (inside the function's call frame).
  - Local static: Like regular locals, they are private to a function, but they retain their value between invocations of the function. Since they cannot go on the stack, they go into data sections (bss if unitialized; read/write data if initialized).
  - Dynamic: Accessible to any code that has a reference to them. Allocated from the heap. The program is responsible for explicitly allocating and deallocating them.

The Heap

- Recall:



- The heap is the region of the program's memory used for dynamic allocation.
- The heap can grow (up to some limit) to accommodate user requests.
- The heap expands by calling the `sbrk` system call.

When should you use the heap?

- Variable sizes are unknown at compile time.
- The data stored may grow/shrink over the course of the program.
- The lifetime of the variable is greater than a single function, but it is not truly global.

Malloc and Friends

- I'm assuming you are familiar with `malloc, free, calloc, realloc`, but did you ever notice this (from the man page):

  The allocated memory is aligned such that it can be used for any data type.

- What does that mean exactly?
  - Each C object has an associated alignment that indicates at which addresses the variable can be placed.
    - E.g., an alignment of 2 means that the object can be placed at even addresses but not odd addresses.
    - Every type, T, has an alignment, A, such that every object of type T has addresses that are a multiple of A.
  - Since malloc doesn't know what will be stored in the allocated memory, it must always ensure that it provides memory capable of holding data of any type.

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

/*
 * We are going to assume that alignments are a power of 2, because all
 * things digital are powers of 2.  Now, let's pick a maximum power of
 * 2 that we believe the alignment must be less than -- let's say 2^10
--
 * we can't possibly require more than 1 KB alignment.
 * */
#define NUM_POWERS 11
#define NALLOCATIONS 100000

int main() {
    char *p;
    int i, j;
    int histogram[NUM_POWERS];
    size_t max, size;

    for (i = 0; i < NUM_POWERS; ++i)
        histogram[i] = 0;
```

Alignment of structures

- In class, we asked you to experiment with different arrangements of data types within structures and derive some rules about how structures are aligned.
- You should have discovered the following (T is a type):
  - sizeof(T) is a multiple of alignof(T).
  - Alignof(struct T) = max{alignof{F} for each field F in T}
  - A struct's size and alignment have to be such that you can allocate structures in an array.
  - The compiler will pad structures as necessary (add unused bytes to make field alignments work).

Select font size T T T

On a 32-bit machine, what is sizeof(struct s)?

(•) Allow Single Choice Only ( ) Allow Multiple Choices

[ ] Shuffle Answers [x] Allow Retry [ ] Limit Attempts

- 4
- 8
- 12 ✓
- 16

+ Add another answer

Preview

Terms | Privacy & cookies

```
struct s{
      char c1;
      short s;
      char c2;
      int i;
}
```

```
struct s{
    char c1;
    short s;
    char c2;
    int i;
}
```

Fragmentation

- Consider the following:

```
while {heap space left} {
    malloc(256)
    malloc(128)
}
free all 128-byte objects
```

- What will happen when we call `malloc(256)`?

External Fragmentation

- When we have enough free space to allocate something, but we are unable to do so, due to how that free space is arranged.
- Can only happen when we permit variable-sized allocations.

- How do we avoid external fragmentation?

Allow only fixed-size allocations

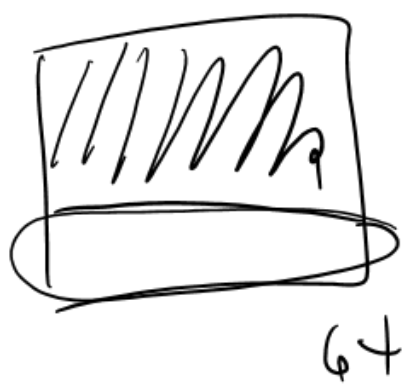
Fixed-Size and Slab Allocators

- If external fragmentation only happens when you allow variable-sized allocations, why not give only fixed size allocations?

- Slab allocators leverage this idea:
  - Allocate large chunks of memory to different slabs
  - Perform fixed-sized allocation within each slab

Internal Fragmentation

- Arises in fixed size allocators, or allocators that limit the exact sizes in which they will allocate objects.
- Refers to the space that is inside of an allocation, but unused by the requester.



Wrapping Up

- Dynamic memory allocation is a critical component of many applications.
- Allocators are required to provide aligned storage.
- Different types of allocators can produce different types of fragmentation:
  - External fragmentation: free space that you can't use because you don't have consecutive memory available.
  - Internal fragmentation: space that is allocated to an object but unused.