Free Space Simulator

Calvin Yong

University of Central Florida

Spring 2021

Goals

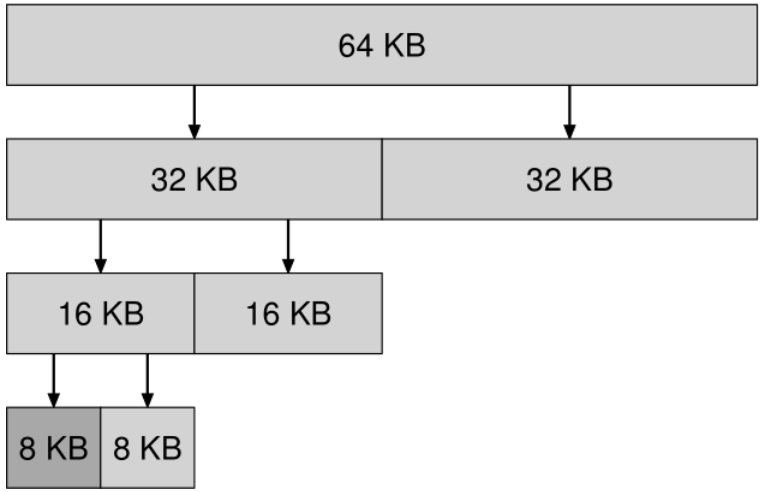
- Simulate a buddy memory allocator
- Measure its efficiency in one way by measuring its internal and external fragmentation on a variety of workloads
- Compare it to a list-based freelist (OSTEP's `malloc.py`)

Managing free space

- Allocators manage memory by using a data structure to keep track of free space
  - List-based (doubly linked list)
  - Trees
- Various ways to evaluate the quality of an allocator
  - Speed / concurrency / cache-friendly / scaling
  - Internal/External fragmentation
  - Space required for headers/metadata

Buddy Allocator

- Published 1965
- Uses a buddy system to efficiently coalesce free blocks
- Used in jemalloc (FreeBSD)

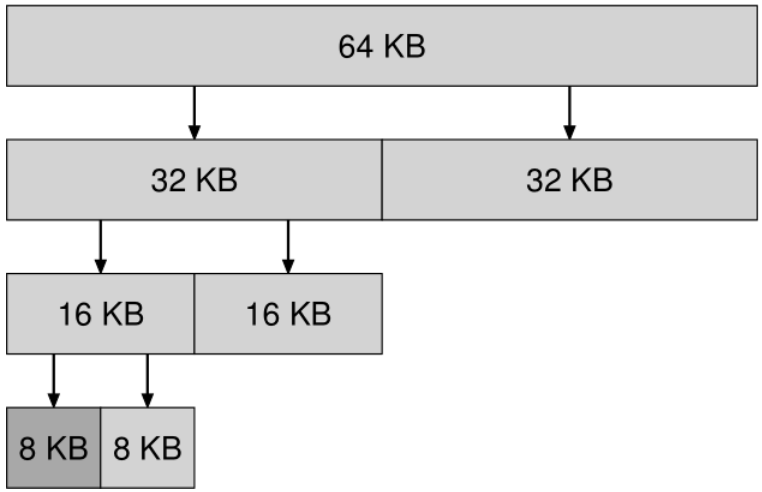


Operating Systems: Three Easy Pieces

Allocating

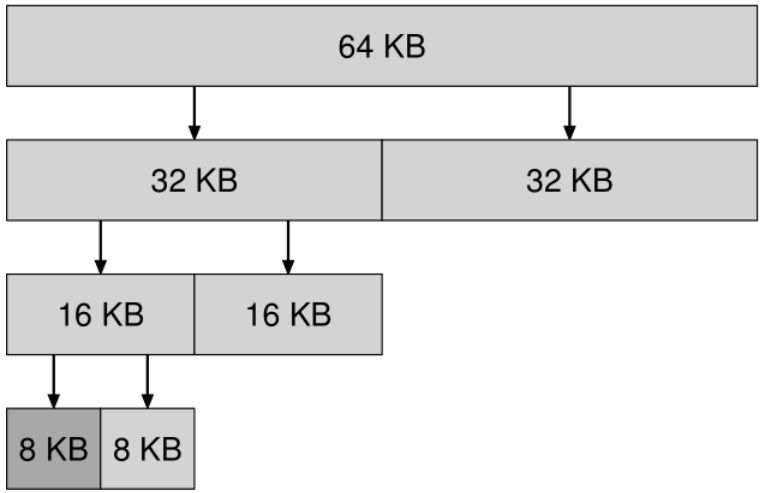
- Structure begins with a list of linked lists, all empty except the top list with one node

1. Find the smallest block that can accommodate the requested size
2. If a free block does not exists, go up levels until a free node is found.
3. Split the free node. When the smallest size is reached, allocate that node.



Freeing Memory

1. Free the block.
2. Check if buddy is free. If so, coalesce.
3. Repeat step 2 until buddy is not free, or the upper limit is freed (all memory is freed).



Strengths and Weaknesses

- Strengths
  - Fast coalescing (buddy differs by a single bit)
  - Good at reducing external fragmentation
- Weaknesses
  - Suffers from internal fragmentation since we can only give out sizes that are powers of two.

Implementation

- All allocators we simulate must implement the following methods.

```rust
pub trait Allocator {
    /// Allocate memory for the requested size. Returns None
    /// if space cannot be allocated
    fn malloc(&mut self, size: usize) -> Option<usize>;

    /// Frees the memory for the given pointer. Returns
    /// an error if the pointer doesn't exist
    fn free(&mut self, ptr: usize) -> Result<(), &str>;

    /// Get the the largest amount of memory that is
    /// possible to allocate
    fn largest_alloc(&self) -> usize;

    /// Get the total free space, which might not be possible
    /// to request due to external fragmentation
    fn free_space(&self) -> usize;

    /// Get the amount of internal fragmentation
    fn internal_frag(&self) -> usize;

    /// If there is free space, get a measure
    /// of the external fragmentation
    fn external_frag(&self) -> f32 {
        1.0 - (self.largest_alloc() as f32 / self.free_space() as f32)
    }

    /// Print the allocator. Too lazy to implement Display
    fn print(&self);
}
```

Experiments and Workloads

- Measurements (averaged across 5 runs)
  - Internal fragmentation
  - External fragmentation

- Workloads
  - Constant size memory allocations
    - `free()` only frees the memory most recently allocated
    - Resembles the operations of a stack
  - Random size memory allocations
    - `free()` can free any memory allocated

Results

- Stack

	Internal (50%)	External (50%)	Internal (80%)	External (80%)
FreeList	0	0	0	0
Buddy	0	0.4911	0	0.3841

- Random

	Internal (50%)	External (50%)	Internal (80%)	External (80%)
FreeList	736.4	0.0271	5211.8	0.9538
Buddy	516	0.4643	7950	0.2

Demo