CS 111: Operating System Principles
Lecture 13

Memory Allocation

3.0.0

Jon Eyolfson
November 9, 2021

Static Allocation is the Simplest Strategy

Create a fixed size allocation in your program

e.g. `char buffer[4096];`

When the program loads, the kernel sets aside that memory for you

That memory exists as long as your process does, no need to free

Dynamic Allocation is Often Required

You may only conditionally require memory

- Static allocations are sometimes wasteful

You may not know the size of the allocation

- Static allocations need to account for the maximum size

Where do you allocate memory?

- You can either allocate on the stack or on the heap

Stack Allocation is Mostly Done for You in C

Think of normal variables

e.g. `int x;`

The compiler internally inserts `alloca` calls

e.g. `int *px = (int*) alloca(4);`

Whenever the function that called `alloca` returns, it frees all the memory

This just restores the previous stack pointer

This won't work if you try to use the memory after returning

You've Used Dynamic Allocation Before

These are the `malloc` family of functions

The most flexible way to use memory, but is also the most difficult to get right

You have to properly handle your memory lifetimes, and `free` exactly once

Also, there's a new concern — fragmentation

Fragmentation is a Unique Issue for Dynamic Allocation

You allocate memory in different sized contiguous blocks

Compaction is not possible and every allocation decision is permanent

A fragment is a small contiguous block of memory that cannot handle an allocation

You can think of it as a “hole” in memory, wasting space

There are 3 requirements for fragmentation

1. Different allocation lifetimes
2. Different allocation sizes
3. Inability to relocate previous allocations

There's Internal and External Fragmentation

External fragmentation occurs when you allocate different sized blocks

There's no room for an allocation between the blocks

Internal fragmentation occurs when you allocate fixed sized blocks

There's wasted space within a block

Credit: Daniel Ritz

We Want to Minimize Fragmentation

Fragmentation is just wasted space, which we should prevent

We want to reduce the number of “holes” between blocks of memory

If we have holes, we’d like to keep them as large as possible

Our goal is to keep allocating memory without wasting space

Allocator Implementations Usually Use a Free List

They keep track of free blocks of memory by chaining them together

- Implemented with a linked list

We need to be able to handle a request of any size

For allocation, we choose a block large enough for the request

- Remove it from the free list

For deallocation, we add the block back to the free list

- If it's adjacent to another free block, we can merge them

There are 3 General Heap Allocation Strategies

Best fit: choose the smallest block that can satisfy the request

Needs to search through the whole list (unless there's an exact match)

Worst fit: choose largest block (most leftover space)

Also has to search through the list

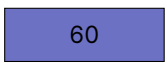
First fit: choose first block that can satisfy request

Allocating Using Best Fit (1)

Note that blocks with a blank background and a number are free

Where do we allocate this block?

Allocating Using Best Fit (2)



Where do we allocate this block?

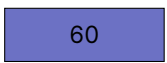
Allocating Using Best Fit (3)

The next block does not fit anywhere

Allocating Using Worst Fit (1)

Where do we allocate this block?

Allocating Using Worst Fit (2)



Where do we allocate this block?

Allocating Using Worst Fit (3)

Next block fits exactly in remaining space

Best Fit and Worst Fit are Both Slow

Best fit: tends to leave very large holes and very small holes
Small holes may be useless

Worst fit: simulation says it's the worst in terms of storage utilization

First fit: tends to leave "average" size holes

The Buddy Allocator Restricts the Problem

Typically allocation requests are of size 2^n

e.g. 2, 4, 8, 16, 32, ..., 4096, ...

Restrict allocations to be powers of 2 to enable a more efficient implementation

Split blocks into 2 until you can handle the request

We want to be able to do fast searching and merging

You Can Implement the Buddy Allocator Using Multiple Lists

We restrict the requests to be $2^k, 0 \leq k \leq N$ (round up if needed)

Our implementation would use $N + 1$ free lists of blocks for each size

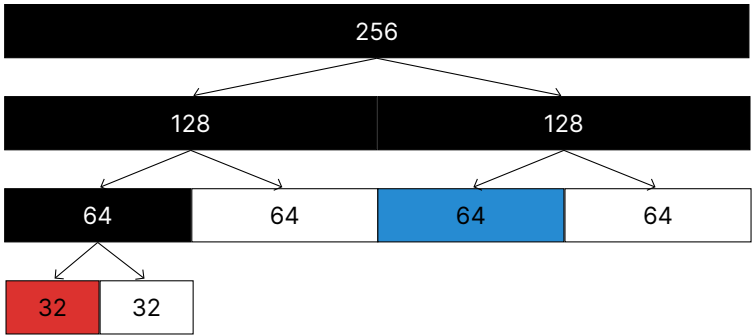
For a request of size 2^k, we search the free list until we find a big enough block

- Search $k, k + 1, k + 2, \ldots$ until we find one
  - Recursively divide the block if needed until it's the correct size
    - Insert "buddy" blocks into free lists

For deallocations, we coalesce the buddy blocks back together

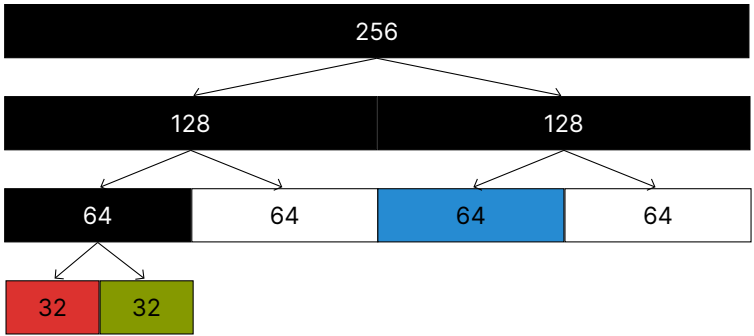
- Recursively coalesce the blocks if needed

Using the Buddy Allocator (1)



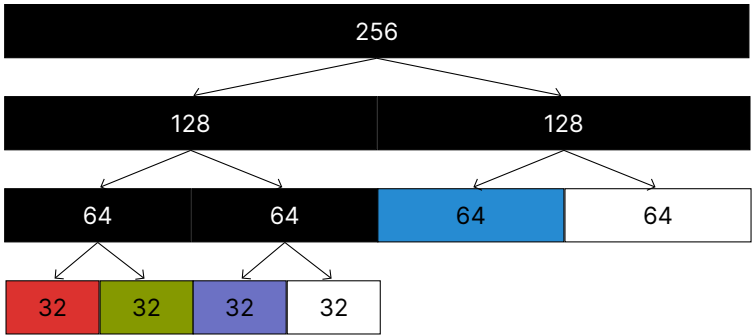
Where do we allocate a request of size 28?

Using the Buddy Allocator (2)



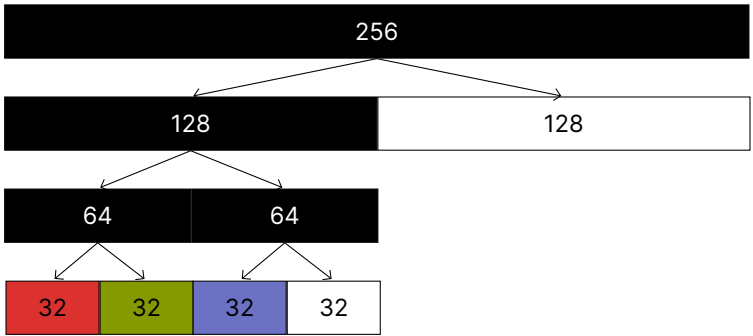
Where do we allocate a request of size 32?

Using the Buddy Allocator (3)



What happens when the we free the size 64 block?

Using the Buddy Allocator (4)



Buddy Allocators are Used in Linux

Advantages

- Fast and simple compared to general dynamic memory allocation
- Avoids external fragmentation by keeping free physical pages contiguous

Disadvantages

- There's always internal fragmentation
  - We always round up the allocation size if it's not a power of 2

Slab Allocators Take Advantage of Fixed Size Allocations

Allocate objects of same size from a dedicated pool

- All structures of the same type are the same size

Every object type has it’s own pool with blocks of the correct size

- This prevents internal fragmentation

Slab is a Cache of “Slots”

Each allocation size has a corresponding slab of slots (one slot holds one allocation)

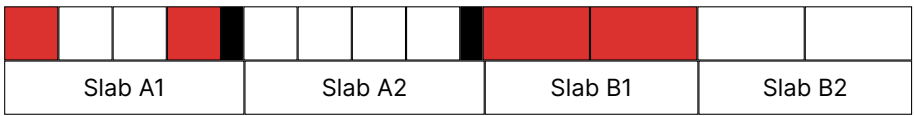
Instead of a linked list, we can use a bitmap (there’s a mapping between bit and slot)

- For allocations we set the bit and return the slot
- For deallocations we just clear the bit

The slab an be implemented on top of the buddy allocator

Each Slab Can Be Allocated using the Buddy Allocator

Consider two object sizes: A and B



We can reduce internal fragmentation if Slabs are located adjacently

In this example A has internal fragmentation (dark box)

The Kernel Has To Implement It's Own Memory Allocations

The concepts are the same for user space memory allocation
(the kernel just gives them more contiguous virtual memory pages):

- There's static and dynamic allocations
- For dynamic allocations, fragmentation is a big concern
- Dynamic allocation returns blocks of memory
  - Fragmentation between blocks is external
  - Fragmentation within a blocks is internal
- There's 3 general allocation strategies for different sized allocations
  - Best fit
  - Worst fit
  - First fit
- Buddy allocator is a real world restricted implementation
- Slab allocator takes advantage of fixed sized objects to reduce fragmentation