CENG 448 — Real Time Operating Systems

Memory Management

Dr. Larry D. Pyeatt

South Dakota School of Mines and Technology

Overview

Dynamic memory allocation is very nice to have, but can cause issues in a Real Time System.

The concept is quite simple:

1. There is a pool of available memory called the heap, and
2. There are functions (e.g. `malloc`, `realloc`, and `free`) for allocating and deallocating memory from the heap at runtime.

Embedded systems developers may implement custom memory-management facilities on top of those provided by the RTOS.

The `malloc`, `realloc`, and `free` functions can be implemented in various ways, and there are trade-offs.

Some implementations may have issues with fragmentation, others may have speed issues, etc.

Common features

Many heap management schemes will have a *minimum allocation unit.*

The memory management functions must know the starting address(es) and size(es) of the memory block(s) that are available.

The memory management functions will use some of the available memory to store data structures that keep track of which portions of memory are currently in use, and which portions are available.

Implementation requirements:

- tracking block metadata
- managing free memory
- performing allocations and deallocations

Metadata

Data for tracking what memeroy is allocated and what is free.

Typical metadata includes the size and allocation status for blocks of memory, and usually pointers to the next free block or chunk of free blocks.

- usually store in a block “header”
- if the minimum allocation size is greater than 1 byte, you can use bottom bit(s) of size for storing other information.
- When a block is freed, it may be possible to merge it with the previous and/or succeeding block.

Tracking Available Memory

A very simple method is to use a bit map, where each bit in the map represents one block of memory. When memory is allocated, `malloc` allocates a little bit extra, and stores the number of blocks allocated there. When the memory is released, `free` will know how many blocks to release.

Simple, but what are the drawbacks?

Allocation Array

An array is used to represent the memory blocks. The size of a sequence of free blocks is stored in the first and last entries.

12											12

Negative numbers indicate that a block is used. When blocks are allocated, they are taken from the end:

9								9	-3		0

If two more blocks are allocated:

7						7	-2	0	-3		0

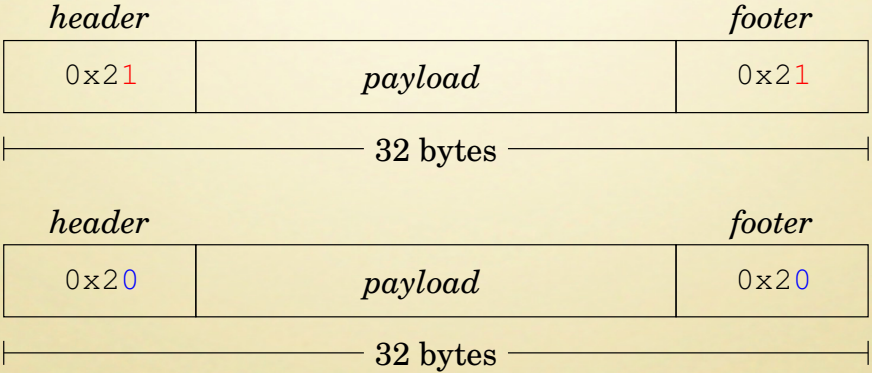
If the first block is freed:

7						7	-2	0	3		3

Allocation Array Implementation

The allocation array is a concept, but implementations can vary.

Most implementations find some clever way to store the allocation array so that overhead is minimized.



Selection Policies

Best fit: check all free blocks and find the one that is closest (but not too small) to the needed size. Expensive, but low fragmentation.

First fit: start from beginning of array, and stop as soon as you find a big enough block.

Next fit: start from point where last allocation was made, and stop as soon as you find a big enough block.

Worst fit: always allocate from the largest free block.

For Worst Fit, you can add a priority queue to store the size and starting location of each free block. Largest block at the front. The heap data structure is a good choice for this. Selecting the block is O(1), and updating when a block is released is O(1).

For Best Fit, a binary tree can be used. Selecting a block and freeing a block are both O($log_2 N$)

Managing Free Space

We need to coalesce adjacent free blocks in order to decrease fragmentation and improve performance

We have a choice of when to do this:

1. at search time: deferred coalescing
   - can be done without a footer
   - could result in a cascade of coalesces
   - indeterminate, unbounded performance
2. when freeing: immediate coalescing
   - requires a footer
   - at most two coalesces
   - bounded performance.
3. run periodic task at low priority

Additional Considerations

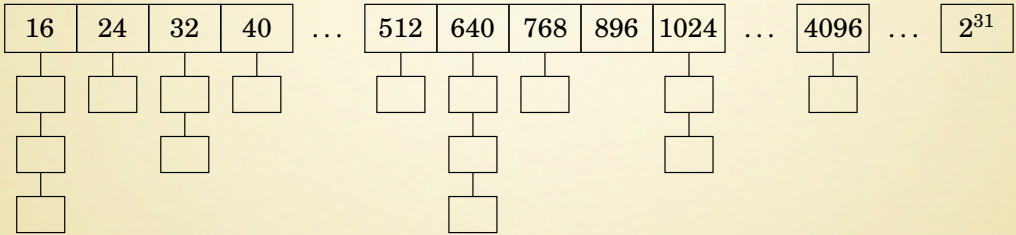
Real-world programs (that use the allocator) may exhibit exploitable patterns

- allocation ramps
- plateaus
- peaks
- common request sizes
- locality

An allocation array approach may not be best for some applications.

You can always write a custom allocator to get the performance you need.

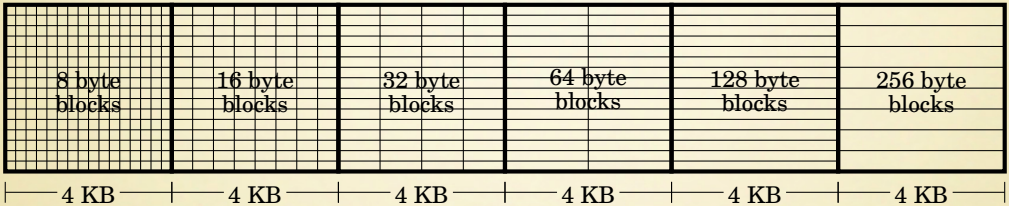
Bins (Pools) of fixed-size objects



- Can be extremely fast.
- Time to allocate/deallocate can be O(1).
- May waste memory (internal fragmentation).

Simple Segregated Storage

Pre-allocate lists of fixed block sizes. Use bins to track free blocks.



Advantages:

- Allocation and deallocation are O(1) and very predictable.
- Extremely low overhead.

Disadvantage:

- Worst case is almost 50% internal fragmentation.
- External fragmentation could be even worse.

Extension

When allocating, if the bin is empty, get a new page and break it into objects of the required size. Track unused pages with another data structure.

If entire page becomes empty, release it. Need to track how many objects are allocated from each page.

Allocation and deallocation times are less predictable.

Segregated Fits

Instead of using simple powers-of-two, actually look at the sizes you will need and provide apropriate bins, or provide a way to create bins for sizes that are used often.

For `malloc`:

1. look in appropriate bin
2. if it is not empty, return first block
3. otherwise, look in next larger bin, take a block from there, and split it, returning the leftover part to the appropriate bin.

For `free`?

Approximates best fit (i.e., good fit) with high speed by reducing search space

May choose not to coalesce (or defer coalescing) for smaller, common sizes

Hybrid Allocator

allocated block	size, status=allocated
	. . . user data space . . .
	size
free block	size, status=free
	pointer to next block in bin
	pointer to previous block in bin
	. . . unused space . . .
	size
allocated block	size, status=allocated
	. . . user data space . . .
	size
other blocks	. . .
wilderness block	size, status=free
	. . . unused space . . .
	size

Buddy Systems

Very little block overhead:

- free/allocated bit
- is block “whole” or split?
- size not needed!

Fast, bounded behavior, but fragmentation can be very bad.

Blocking vs Non-Blocking

The common implementation of malloc will return NULL if memory is not available.

Why not have it block the task until memory is available?

Hardware

There are two classes of hardware to assist in memory management:

- Memory Protection Unit (MPU)
- (Virtual) Memory Management Unit (MMU or VMMU).

Examples

Newlib `malloc` and `free`

BGET

Doug Lea's malloc