This is an assignment in which you will implement a binary buddy memory allocator to manage physical memory. The goal of the assignment is to reinforce the ideas of physical memory management: both in terms of the application-level requests and the management of kernel requests. In doing so, you will implement the interface for much of the high-level functionality associated with a memory management system.

Collaboration Policy

This assignment is intended to be implemented as a *pair-programming* assignment (often in groups of 3)! That means that you and your partner are working on the assignment **together**.

After the initial programming phase of the assignment, you and your partner will participate in a *code review* of another group's submission. Be sure to review the code reviewing standards from Homework 0 submit your code review on Gradescope!

Submission

You are asked to submit three deliverables for this assignment: ① the programming component, ② your code review for your partnered group, and ③ a reflection to the review of your code. In addition, in-class participation is required on **November 2nd** to both give and receive feedback for full credit.

Getting Started

This assignment is designed to run on the course virtual machine (e.g., `ssh stum2025@itbdcv-lnx04p.campus.pomona.edu`). The stencil for assignments in this class assume particular compiler versions and processor features, so using the VM will make your life easier.

Download the source for this assignment to get started! You can find all of the tasks that you will need to implement by running "`grep "TODO: implement me!" -r .`" from the "fs-stencil" repository. In the repository, you will find a `Makefile` to help you build the various sources for each part of this assignment.

Deadlines

Section	Deadline	Submission URL
Programming Component	Nov 13 (EOD)	HW4 Programming
Code Review	Nov 18 (before class)	HW4 Code Review
Code Review Reflection	Nov 20 (EOD)	HW4 Code Review Reflection
Post-Review Resubmission	Nov 25 (EOD)	HW3 Programming (late due date)

Note, this is a shorter assignment! When you complete this assignment, the final assignment of the course will be the integration of each of the components of the OS-kernel system. With the brevity of this assignment, start thinking about how you may link the various components of the assignments throughout the course together. More details to come!

1 Implementation

The goal of this assignment is to implement a *binary buddy memory allocator*[1]. By this, we mean that users will make allocation requests to the memory management system, and the system will allocate "chunks" of physical memory to map the virtual address to the physical address. The allocator finds a chunk of the appropriate size by searching a `free_list` data structure in which free memory chunks are maintained. The data structure, organizes free "chunks" in terms of their size (relative to a fixed "`PAGE_SIZE`") in which a chunk at list index i is $2^i \times$ `PAGE_SIZE` bytes.

In this part of the assignment, your job is to implement the binary buddy allocation memory management scheme. Much like the "`slab_allocator`" interface described in HW2, your memory allocator should implement the following functions:

1. `void buddy_init()`
2. `void *buddy_obj_alloc(long size)`
3. `void buddy_obj_free(long size, void *obj);`
4. `void *page_alloc_n(int npages)`
5. `void page_free_n(void *ptr, int npages)`

The `alloc` and `free` functions allocate and free space within the physical memory space by implementing and maintaining the buddy allocator data structure. `buddy_init` initializes the subsystem and allocates the associated fields.

To make the life of the allocator as easy as possible, assume that when a block is freed that it must *always* be "merged" with a buddy when appropriate!

Hints and Notes

`buddy_init` When implementing the initialization of this class, there is a circularity! To implement the subsystem, you will need to allocate space for each of the objects and methods, including the `free_list` data structure. To construct the `free_list` data structure, you need to know how much memory is available! To work around this, there are two main approaches: ① pre-compute the sizes of your structures in advance or ② use a heuristic. Either strategy is sufficient for this assignment, but do your best to maximize the usable memory in your system!

`alloc` and `free` These functions both involve fine management of the `free_list` data structure, and most bugs that can arise will relate to this management. You may find it useful to implement helper methods associated with the data structure management first, and then building the allocation and freeing functionality on top of that. This will make it easier to manage the cases when the block that you need from a particular list may or may not exist!

When reclaiming a block, a chunk should be created with a chunk that is not currently in use! This will require having some mechanism for being able to distinguish blocks that are (not) in use and having a mechanism to find these blocks.

[1]Note, this is the allocator that is used in Linux: `https://www.kernel.org/doc/gorman/html/understand/understand009.html`

2 Tests

When performing the code review (and testing the functionality of your own code), there are several tests that should implement.

1. You should put together a scenario in which the memory management object is linked into an a test where memory is allocated and freed. Such a test will emulate the behavior of allocating the address space of a process, etc... and reclaiming the memory associated with the process upon its termination.
2. You should put together a scenario in which all of memory is allocated (e.g., an allocation failure) with subsequent frees allowing next allocations to occur.
3. You should consider an allocation of a size larger than the largest "chunk" that is currently supported in by the `free_list`.
4. You should implement at least one other test as well!

3 (Optional) Extensions

3.1 User Interfaces

Suppose you wanted to implement the "`malloc`" library call. This is achieved by utilizing the `brk`/`sbrk` system call, which manages the `brk` pointer in the address space within a process. Implement a fast implementation of the `brk` system call that uses the interfaces of the memory allocation system.

3.2 Safe Concurrent Allocation

Given that several different processors may all be executing OS kernel code in parallel, it is possible that several different CPUs may be allocating memory from the memory management system. Given this, modify your implementation such that your buddy allocator is thread-safe across several processors.

3.3 Prioritization of Memory Levels

Modern systems tend to implement *Non-Uniform Memory Access* (NUMA) systems in which some memory is closer or further than others. Given a memory priority of an address space, write an allocator that performs high priority allocation/reclamation to that address space if the user so demands.