Implementation of the Buddy Allocator

My allocator is built around a set of recursive and iterative algorithms that manage a pre-defined array of free lists, `free_list_[]`, where each index corresponds to a block order. My algorithm allocates pages by first checking the free list for the exact order requested, then iteratively searching upward until it finds a non-empty list (or until `LastOrder` is reached). If a block at a higher order is found, my algorithm then repeatedly calls the `split_block` function, which removes and divides the block, then inserts each half into the next lowest order. This process continues until the requested order has been reached.

When pages are freed via the `free_pages` function, my algorithm will simply call the `merge_buddies` function, which handles all logic. This function will check if the block's buddy is free, and recursively merge blocks until either `LastOrder` has been reached, or no more free buddies are found. This recursive process ensures that blocks are automatically coalesced to the largest possible size upon being freed, which is the main mechanism used by the buddy system to prevent memory fragmentation.

Assumptions

My implementation relies on the main assumption that the `page*` structs are contiguous in memory, and their addresses directly correlate with the physical PFNs they represent.

Additional Implementations

In a real-world scenario, it is likely that multi-core kernels may request for pages simultaneously. Keeping this in mind, I decided to make use of stacsos' `spinlock_irq` lock to acquire a spinlock and also disable local interrupts on the current core to enforce thread-safety in my implementation. I applied this locking mechanism on `allocate_pages` and `free_pages`, since the other functions would be called by these two main functions. I also made sure that the lock is always released so that resources are always freed, even in error paths.

Notable Challenges

A major challenge for me was implementing a safe check for a buddy's fee status. I initially approached this problem by reading the `page_metadata` of the calculated buddy. However, if the buddy was not free, that memory would be in use, and reading it as if it were metadata would cause unpredictable memory corruption. As a result, the kernel often triple faulted or simply froze when I was initially developing my implementation. Debugging this was difficult as well, since a triple fault provides no information on what went wrong. I eventually solved this by using a find_free_block helper function that iterates through the free list to check for the presence of a buddy block.

Insights on Buddy Allocator

My safe-check solution is a linear search, which means that allocation and freeing could potentially take up to O(kn) time, where n is the highest order, and k is the number of free blocks in `free_list_`. If I had more freedom to implement my algorithm, I might think to use a map of some sort to store all physical pages, allowing for constant time checks to see if a buddy is free.