Final Project
Aditi Athavale (asa85) & Tim Han (xh87)

- Method: Implicit free list
- Overhead
    - Heap:
        - Initialized as a data structure
        - Three fields:
            - Heap_size: Size of the heap provided
            - Start: A pointer that points to the first block's head tag
            - End: Index (not pointer) of the last block's tail tag (heap end): If int* blocks = start, then blocks[end] returns the last tail tag.
        - Stored in the beginning of the heap
    - Block:
        - NOT initialized as a data structure
        - Two tags: A Header and Tail for each block. Each tag contains block size and valid bit.
        - Only one large free block after hl_init (From First Free Byte to heap end).
        - When allocating a free block, split the leftover into a new free block to prevent internal fragmentation.
        - Not stored at heap's start, but scattered all over the part of the heap after Heap_overhead.

- Search Strategy:
    - Best-fit search: Chooses the free block that minimizes leftover size.
    - Will be implemented in hl_alloc and hl_resize

- Defragmentation:
    - Implemented in hl_free (and/or hl_resize).
    - Logic: When freeing a block, check for free block immediately before and after the target block. If present, merge the free blocks.

- hl_init function implementation
    - Create the heap data struct with the overhead and store (`heap_header_t *header = (heap_header_t *)heap;`)
    - Calculate the start field for alignment. Empty buffer may inserted between heap overhead and the first block to ensure that the beginning block's pointer is 8-byte aligned.
    - Only one free block in the beginning which encompasses all the way from start (the pointer calculated above) to end of the heap. Initialize this free block by writing a head and tail tag to start and end respectively.

Heap overhead	Head	First free block	Tail

- hl_alloc function implementation
    - Start by checking the first block at address “start” of the heap_overhead field.
    - If block is not free, jump to the head tag of the next block (if present), and then continue searching
    - If block is free, check if the size is big enough.
        - If so, split the block and calculate the leftover size. Compare the size to the accumulator “frag_s”(initialized before searching). If the leftover size is smaller, store the free block’s pointer and index, and update the accumulator.
        - If not, jump to the end of the current block, and then continue searching.
    - After searching is completed, check the accumulator:
        - If accumulator is the same as before searching, no free block allocatable, so return NULL.
        - If accumulator has changed, at least one suitable free block is found, so return the latest free block pointer.

Before hl_alloc

Heap overhead	Head	First free block	Tail

hl_alloc new block after splitting

Heap overhead	Head 1	Allocated block	Tail 1	Head 2	New free block	Tail 2

- hl_release implementation
    - Go to the block pointer, check if there is another free block immediately before/after this block:
        - If so, merge them
        - If not, just turn off the use-bits of the current block

- hl_resize implementation
    - Check if the original free block has another free block immediately before/after this block
        - If so, check if merging them will give us enough space for the resize

    - If the space is enough, release the block since hl_alloc is guaranteed to find at one suitable pointer (the current block’s).
    - If not enough, don’t release the block since releasing it wouldn’t help hl_alloc finding a suitable block.
  - Run hl_alloc.
    - If a suitable block is found, release the old block if it hasn’t been released already then return the pointer.
    - If a suitable block is not found, there’s not enough space for the resize request, so return NULL.

- Spinlock/unlock: We wrote the MIPS instructions using inline assembly code
  - For hl_init, hl_alloc, and hl_release, we will acquire the lock at the beginning of the function and release the lock before each return statement.
  - For the hl_resize function, create two helper functions that are just the same as hl_alloc and hl_release except that there are no spin_lock or spin_unlock. Use these functions in hl_resize, so we only need one spin_lock in the beginning and one spin_unlock in the end.