**MALLOC Design Doc**

**Richard Greenbaum, rrg73**

**John Demouly, jjd268**

**Hl\_init:**

The heap is initialized to be 8 byte aligned. The heap\_size is decreased by the same amount that the heap pointer was increased to insure that the heap does not extend past the memory allocated for it. It is then 4 byte aligned. The block\_data struct is 4 bytes long and contains one attribute: block\_size, the size of the memory block directly preceding it. Each block of memory (allocated or not) will have a block\_data struct directly preceding it. The first block on the heap will be reserved as the reference block for the spin lock and is set to 0. Thus its value will always be either a 0 or a 1 and its sole purpose is to ensure that only one core is accessing any of the hl\_lib functions at a given time. The next block is initialized as the metadata for the single block of unallocated memory that is the rest of the heap. The last bit of the block\_size is used to signify whether the memory block preceding the metadata is free memory or allocated memory. The last block in the heap is set as a metadata with a block\_size of INT\_MAX. It is used as reference when looping through the linked list of metadata pointers as a signal that the loop has reached the end of the heap.

**Hl\_alloc:**

When hl\_alloc is called, the heap is 8 byte aligned and the block\_size attribute is 8 byte aligned minus 4, as each block will be preceded by a 4 byte metadata. The function iterates through the heap by jumping from one block\_data to the next. If the block of data is free (block\_size%2==0), the block size is large enough to fit the stored block, and the block size is the smallest one yet that fits the first two criteria, the pointer to that metadata is stored as best\_ptr. The loop ends when the block\_size evaluates to INT\_MAX. The block size of best\_ptr is incremented to signify that it is now allocated memory, a metadata block is added to denote the remaining unallocated memory after the stored block, and the value best\_ptr + the size of the metadata is returned.

**hl\_release:**

To remove a block of memory from the heap, we locate the metadata of corresponding to the given block pointer and set its least significant bit to 0. We then check whether or not the adjacent memory blocks are free and, if so, combine them to make a larger chunk of free memory.

**hl\_resize:**

To resize a block, we first check to ensure that we were not passed a null pointer if we are, then we simply act is if hl\_alloc was called with the block size to be allocated equal to the new\_block\_size argument. Then we release the block, determine the optimal location for the new sized block using the same method as in hl\_alloc, and put the data back into the block. If the pointer to the block location is unchanged, there is no need to call memcpy and we return the original block\_pointer. If there hl\_alloc attempt is unsuccessful, we unfree the block\_pointer, by adding one, and returning NULL to signify that there is no room to resize this block in the heap.

However, to get around the spin lock, we are using helper functions hl\_prealloc and hl\_prerelease. These are functionally the same as alloc and release respectively but are only called once the spinlock is already grabbed by hl\_resize. The only difference is that hl\_prealloc has a third argument, the previous block pointer. This ensures the memory is not overlapping when determining the best pointer so a call to memcpy() will not corrupt the data.

**Testing:**

Initial we tested each function with basic tests to ensure for the most part they were working. Testing init setting the headers correctly, making sure allocate returned the correct pointer. Making sure a call to release would free the block so a subsequent call to alloc would return the recently released block pointer if it was the heap was subsequently empty. Making sure resize worked correctly if there was only one block in the heap etc.

We then started testing edge cases where the functions should return NULL like when the array was full or an alloc called for something that was larger than the heap itself. Other edge cases like different circumstances on a call to resize were also tested. Like first block in the heap, making it smaller, larger or making sure if the block needed to be moved it in fact was moved. Once we were passing all of our speculation tests, we moved to stress tests where we created loops with numerous calls to alloc, release and resize in order to make sure that our specs did not break down after larger amounts of calls. When we were ready we submitted to the auto grader and were able to see that we made it to the leaderboard, thus our testing was robust enough to ensure our specifications worked.

**Spin Lock:**

Spin lock is implemented using a Mutex lock directly from the lecture slides. The lock is implemented using MIPS inline assembly code using asm volatile().

Spin\_lock takes in only an slock pointer and then uses load links and store conditionals to ensure other threads/cores are not accessing and hl functions while the current thread is trying to get the lock. Once the lock is acquired, it returns to the parent function to execute whatever hl function was called.

Spin\_unlock takes in the same pointer as spin\_lock and stores the value 0 into that memory location, freeing the lock for other threads to grab.

We tested the spin lock unlock pair with three criteria:

1. Manually setting the value of where slock points to zero, grabbing the lock, and then making sure that the value at that location is 1, making sure the lock was actually grabbed.
2. Setting the slock value manually to one, and then testing to ensure that a call to spin\_unlock sets that value to 0, showing the lock is in fact freed.
3. Manually setting the value of the slock to 1, signifying another thread has the lock. Then trying to call lock resulting in an infinite loop, since our call to spin\_lock is repeatedly trying to grab the lock.