Pointers

- Casting
  - <type_A>* to <type_B>*
    - Provides the same value.
    - Change the behavior when dereferenced.

  - <type_A>* to integer vise versa
    - Pointers are 8-byte numbers!
    - Wrong use will lead to headaches… lots of errors… so be careful!

Pointer Arithmetic

- Let's think of pointer + a:
  - pointer2 = pointer + (a * sizeof(type_a)) //in C code
  - lea (pointer, a, sizeof(type_a)), pointer2 //in assembly

- Examples: our machine is 64-bit
  - int *ptr = (int *) 0x12345670;
  - int * ptr2 = ptr + 1;

  - char *ptr = (char *) 0x12345670;
  - char *ptr2 = ptr + 1;

  - int * ptr = (int *)0x12345670;
  - int * ptr2 = (int *) (((char *) ptr) + 1);

Pointer Arithmetic

- Let's think of pointer + a:
  - pointer2 = pointer + (a * sizeof(type_a)) //in C code
  - lea (pointer, a, sizeof(type_a)), pointer2 //in assembly

- Examples: our machine is 64-bit
  - int *ptr = (int *) 0x12345670;
  - Int * ptr2 = ptr + 1; // ptr2 = 0x12345674

  - char *ptr = (char *) 0x12345670;
  - char *ptr2 = ptr + 1; // ptr2 = 0x12345671

  - int * ptr = (int *)0x12345670;
  - int * ptr2 = (int *) (((char *) ptr) + 1); // ptr2 = 0x12345671

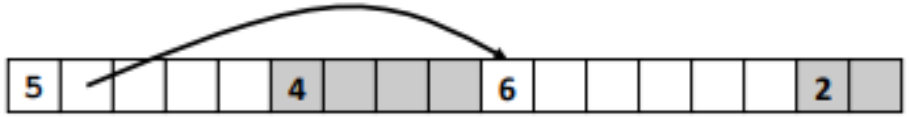
What is Malloc Lab?

- Let us design and implement our very own dynamic memory allocator
  - Get the assignment from Github Classroom
  - Update the team info struct in mm.c
  - Read README.md carefully
  - Complete the assignment
- You will be graded on efficient **memory utilization** *(space efficiency)* as well as the **throughput** *(speed efficiency)* of your malloc implementation
- You are **not allowed** to use the glibc malloc calloc realloc or free in your code
- You must complete the eseential functions:
  - int mm_init(void);
  - void *mm_malloc(size_t size); // reserve a block of memory
  - void mm_free(void *ptr); // release a chunk of memory
  - void *mm_realloc(void *ptr, size_t size); // resize a reserved chunk
- The ultimate heap memory allocator for C
  https://github.com/lattera/glibc/blob/master/malloc/malloc.c

Contents

- Implicit Free List (Chapter 9.9.12 in the textbook)
  - Putting it together: Implementing a simple allocator
  - All blocks using length-links

- Explicit Free Lists (Chapter 9.9.13 in the textbook)
  - Free blocks (not all blocks) using pointers



- Segregated Free Lists (Chapter 9.9.14 in the textbook)

General Allocator Design

- memlib.c (already implemented for you)
  - mem_init

```c
/* private variables */
char *mem_start_brk;  /* points to first byte of heap */
static char *mem_brk;        /* points to last byte of heap */
static char *mem_max_addr;   /* largest legal heap address */

/*
 * mem_init - initialize the memory system model
 */
void mem_init(void)
{
    /* allocate the storage we will use to model the available VM */
    if ((mem_start_brk = (char *)malloc(MAX_HEAP)) == NULL) {
        fprintf(stderr, "mem_init_vm: malloc error\n");
        exit(1);
    }

    mem_max_addr = mem_start_brk + MAX_HEAP;  /* max legal heap address */
    mem_brk = mem_start_brk;                  /* heap is empty initially */
}
```

  - mem_sbrk

```c
/*
 * mem_sbrk - simple model of the sbrk function. Extends the heap
 *    by incr bytes and returns the start address of the new area. In
 *    this model, the heap cannot be shrunk.
 */
void *mem_sbrk(int incr)
{
    char *old_brk = mem_brk;

    if ( (incr < 0) || ((mem_brk + incr) > mem_max_addr)) {
        errno = ENOMEM;
        fprintf(stderr, "ERROR: mem_sbrk failed. Ran out of memory...\n");
        return (void *)-1;
    }
    mem_brk += incr;
    return (void *)old_brk;
}
```

* Reference: Figure 9.41 Memory system model in the textbook

Inline Functions

- Macros for manipulating the free list

```c
static inline int MAX(int x, int y) {
  return x > y ? x : y;
}
//
// Pack a size and allocated bit into a word
// We mask of the "alloc" field to insure only
// the lower bit is used
//
static inline uint32_t PACK(uint32_t size, int alloc) {
  return ((size) | (alloc & 0x1));
}
//
// Read and write a word at address p
//
static inline uint32_t GET(void *p) { return  *(uint32_t *)p; }
static inline void PUT( void *p, uint32_t val)
{
  *((uint32_t *)p) = val;
}
//
// Read the size and allocated fields from address p
//
static inline uint32_t GET_SIZE( void *p )  {
  return GET(p) & ~0x7;
}
static inline int GET_ALLOC( void *p  ) {
  return GET(p) & 0x1;
}
```

```c
//
// Given block ptr bp, compute address of its header and footer
//
static inline void *HDRP(void *bp) {
  return ( (char *)bp) - WSIZE;
}
static inline void *FTRP(void *bp) {
  return ((char *)(bp) + GET_SIZE(HDRP(bp)) - DSIZE);
}
//
// Given block ptr bp, compute address of next and previous blocks
//
static inline void *NEXT_BLKP(void *bp) {
  return  ((char *)(bp) + GET_SIZE(((char *)(bp) - WSIZE)));
}
static inline void* PREV_BLKP(void *bp){
  return  ((char *)(bp) - GET_SIZE(((char *)(bp) - DSIZE)));
}
```

* Reference: Figure 9.43 in the textbook

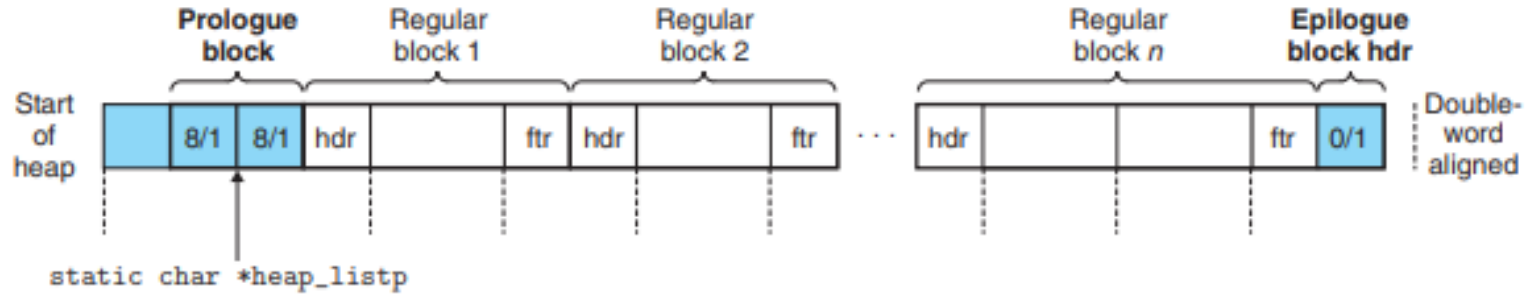
Implementing an Implicit Free Lists

- Files you need to modify / implement
  - Makefile
    - -O0: for GDB debugging
    - -O3: for your grading (enable faster execution and throughput)
  - mm.c
    1. mm_init (reference: Figure 9.44)
    2. extend_heap (reference: Figure 9.45 in the textbook)
    3. find_fit (reference: Practice problem 9.8 in the textbook)
    4. mm_free (reference: Figure 9.46 in the textbook)
    5. coalesce (reference: Figure 9.46 in the textbook)
    6. mm_malloc (reference: Figure 9.47 in the textbook)
    7. place (reference: Practice problem 9.9 in the textbook)
    8. mm_realloc (already implemented for you)
  - Other functions for debugging
    - mm_checkheap
    - printblock
    - checkblock

mm_init

- Create an initial empty heap
  - heap_listp: pointer to first block (global variable)
  - mem_sbrk(): extend heap by given size and return a new starting address
  - PUT(): assign prologue / epilogue block

- Extend the empty heap with a free block
  - extend_heap(): you need to implement



extend_heap

- This function is called in 2 cases
  1. Initialization of the heap
  2. Allocator cannot find enough space for allocation requests

- Calculate the appropriate size to maintain alignment
  - mem_sbrk() using the appropriate size

- Initialize free block header/footer and the epilogue header
  - PUT(): assign header/footer

- Merge free blocks (coalesce if the previous block is free)
  - coalesce(): you need to implement

find_fit

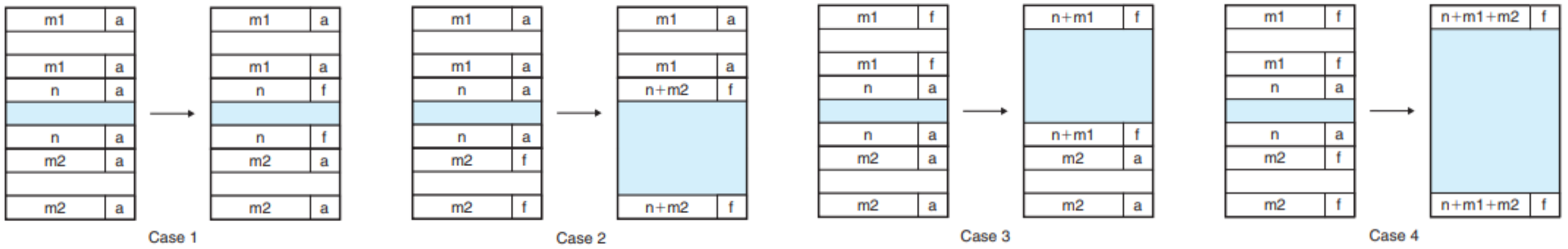
- This function directly affects the throughput of your memory allocator.
- It can be implemented in 3 different approaches.
  - First-fit (practice problem 9.8)
    - Search list from beginning, choose first free block that fits
    - It can take linear time in total number of blocks
  - Next-fit
    - Like first fit, but search list starting where previous search finished
    - Should often be faster than first fit: avoids re-scanning unhelpful blocks
  - Best-fit
    - Search the list, choose the best free block: fits, with fewest bytes left over
    - It will typically run slower than first fit

mm_free

- Read the block size from the address of pointer
  - GET_SIZE(HDRP(pointer))
- Set the “alloc” field to free
  - Header alloc field
  - Footer alloc field
- Coalesce adjacent free blocks
  - coalesce(): you need to implement

coalesce

- Check the “alloc” field of adjacent blocks (previous / next block)
  - GET_ALLOC(): return 1 if the block is allocated, otherwise return 0
- Read the size from the address of the current pointer
  - GET_SIZE(HDRP(pointer))
- Merge adjacent blocks depending on cases (4 cases)



mm_malloc

- Adjust the requested block size
  - The minimum block size is 16 bytes (including Header/Footer and aligned request)
  - Allocate a memory size rounded to a multiple of 8 bytes (to maintain the alignment)

- Search the list of available blocks and place it when a suitable block is found
  - find_fit(): you need to implement
  - place(): you need to implement

- If the search for an available block fails, extend the heap memory size
  - extend_heap(): you need to implement
  - place(): you need to implement

place

- Consider 2 cases
  1. The remaining size after allocating a request >= the minimum block size (16 bytes)
     - Set the “alloc” field of the current requested block to allocated
     - Split the remaining size of the block and set the “alloc” field of it to free

  2. Otherwise
     - Only set the “alloc” field of the current requested block to allocated

Implementing an Explicit Free Lists

- Files you need to modify / implement
  - Makefile
    - -O0: for GDB debugging
    - -O3: for your grading (enable faster execution and throughput)
  - mm.c
    1. mm_init
    2. extend_heap
    3. find_fit
    4. mm_free
    5. coalesce
    6. mm_malloc
    7. place
    8. mm_realloc
  - Additional functions
    - Structure of free block node
      - Predecessor
      - Successor
    - Insert free block node
    - Delete free block node
  - Other functions for debugging
    - mm_checkheap
    - printblock
    - checkblock

Additional Functions

- Structure of free block node
  - Two pointers for predecessor (previous block) and successor (next block)
- Inserting free block node
  - New node's successor is the predecessor's successor
  - New node's predecessor is the predecessor
  - The predecessor's successor and successor's predecessor are the new node
- Deleting free block node
  - Change the predecessor's successor to the current node's successor
  - Change the successor's predecessor to the current node's predecessor
  - Change the current node's predecessor and successor to NULL

mm_init

- Create the initial empty heap

- Initialize free list root node
  - Declare a (root) structure of free block node as a global variable
  - Initially pointing the root itself
    - predecessor = root
    - successor = root

- Extend the empty heap with a free block

extend_heap

- Calculate the appropriate size to maintain alignment
- Initialize free block header/footer and the epilogue header
- Merge free blocks (coalesce if the previous block is free)
  - You need to add a new free block, which is the result of coalesce (extended heap): Inserting free block node()

find_fit

- First fit approach
  - Set starting point to the root node’s successor
  - Searching an appropriate available free block using successors until it indicates the root
- Next fit approach
  - Set starting point to the last node’s successor
  - Searching an appropriate available free block using successors until it indicates the root
- Best fit approach

mm_free

- Read the block size from the address of pointer
- Set the “alloc” field to free
- Coalesce adjacent free blocks
  - You need to add a new free block, which is the result of coalesce (extended heap): Inserting free block node()

coalesce

- Check the “alloc” field of adjacent blocks (previous / next block)
- Read the size from the address of the current pointer
- Merge adjacent blocks depending on cases
  - You need to delete the existing free block nodes to merge adjacent free blocks: Deleting free block node()

mm_ malloc

- Adjust the requested block size
  - Check the request block size with the free block structure size before adjusting block size: MAX(size, the free block structure size)

- Search the list of available blocks and place it when a suitable block is found
  - find_fit()
  - You need to delete the existing free block nodes to use it (place): Deleting free block node()
  - place()

- If the search for an available block fails, extend the heap memory size
  - extend_heap()
  - You need to delete the existing free block nodes to use it (place): Deleting free block node()
  - place()

place

- The minimum block size is MAX(16 bytes, the free block structure size+overhead)

- If the remaining size >= the minimum size
  - Set the “alloc” field of the current requested block to allocated
  - Split the remaining size of the block and set the “alloc” field of it to free
  - Inserting the remaining free block node

- Otherwise
  - Only set the “alloc” field of the current requested block to allocated

mm_realloc

- Check the minimum block size
  - MAX(16 bytes, the free block structure size+overhead)
- Check the current block size
  - GET_SIZE(HDRP(ptr))
- According to the comparison of the current block size and the minimum size
  - If the current block size == minimum size
    - return the current pointer
  - if the current block size > minimum size
    - place() using the minimum size
  - Otherwise
    - Do the mm_realloc using the given size (already implemented for you)

Debugging using GDB

- After compiling (make), run GDB
  - gdb mdriver
- Create breakpoints for debugging
  - b mm_malloc
  - b mm_free
  - etc.
- Run mdriver with a trace file
  - run -a -f trace/short1-bal.rep
- Call mm_checkheap function with 1
  - call mm_checkheap(1)

```
jovyan@jupyter-inle7436:~/lab6-malloclab-Insoo-Lee$ gdb mdriver
GNU gdb (Ubuntu 9.2-0ubuntu1~20.04) 9.2
Copyright (C) 2020 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law.
Type "show copying" and "show warranty" for details.
This GDB was configured as "x86_64-linux-gnu".
Type "show configuration" for configuration details.
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>.
Find the GDB manual and other documentation resources online at:
    <http://www.gnu.org/software/gdb/documentation/>.

For help, type "help".
Type "apropos word" to search for commands related to "word"...
Reading symbols from mdriver...
(gdb) b mm_malloc
Breakpoint 1 at 0x413a: file mm.c, line 258.
(gdb) b mm_free
Breakpoint 2 at 0x3e3e: file mm.c, line 212.
(gdb) r -a -f traces/short1-bal.rep
Starting program: /home/jovyan/lab6-malloclab-Insoo-Lee/mdriver -a -f traces/short1-bal.rep

Breakpoint 1, mm_malloc (size=32767) at mm.c:258
258     {
(gdb) call mm_checkheap(1)
Heap (0x7fffdedc1018):
0x7fffdedc1018: header: [8:a] footer: [8:a]
0x7fffdedc1020: header: [4096:f] footer: [4096:f]
0x7fffdedc2020: EOL
(gdb)
```