CS 4400
Computer Systems

LECTURE 20

Dynamic memory allocation example

Explicit free lists

Review

- heap: allocated and free blocks
- explicit allocator goals: max throughput and utilization
- how are free blocks organized?
- how are free blocks placed?
- are free blocks split?
- are free blocks coalesced?

Example: Simple Allocator

- The design space for your allocator is large.
  - choices for block format, free list format, block placement, block splitting, and coalescing policies
- Simple allocator: implicit free list and immediate coalescing with boundary tags.
- `int mm_init(void)` initializes the allocator.
- `void* mm_malloc(size_t size)` same interface as `malloc`.
- `void mm_free(void* bp)` same interface as `free`.

Implicit Free List Invariant

- First word is an unused padding word.
- The *prologue block* is an 8-byte allocated block.
  - consists of only a header and a footer
  - created during initialization and never freed
- Zero or more regular blocks, created by calls to `malloc` or `free`, follow.
- The *epilogue block*, a zero-sized allocated block (header only), ends the heap.

```c
/* basic constants and macros for manipulating the free list */

#define WSIZE       4       /* word size (bytes) */
#define DSIZE       8       /* doubleword size (bytes) */
#define CHUNKSIZE  (1<<12)  /* initial heap size (bytes) */
#define OVERHEAD    8       /* overhead of header & footer (bytes) */

#define MAX(x, y) ((x) > (y)? (x) : (y))

/* Pack a size and allocated bit into a word */
#define PACK(size, alloc)  ((size) | (alloc))

/* Read and write a word at address p */
#define GET(p)       (*(size_t*)(p))
#define PUT(p, val)  (*(size_t*)(p) = (val))

/* Read the size and allocated fields from address p */
#define GET_SIZE(p)  (GET(p) & ~0x7)
#define GET_ALLOC(p) (GET(p) & 0x1)

/* Given block ptr bp, compute address of its header and footer */
#define HDRP(bp)       ((char*)(bp) - WSIZE)
#define FTRP(bp)       ((char*)(bp) + GET_SIZE(HDRP(bp)) - DSIZE)

/* Given block ptr bp, compute address of next and previous blocks */
#define NEXT_BLKP(bp)  ((char*)(bp) + GET_SIZE(((char*)(bp) - WSIZE)))
#define PREV_BLKP(bp)  ((char*)(bp) - GET_SIZE(((char*)(bp) - DSIZE)))
```

```
/* before calling mm_malloc or mm_free, the allocator must initialize
   the heap by calling mm_init */

int mm_init(void) {
    /* create the initial empty heap – four words */
    if ((heap_listp = mem_sbrk(4*WSIZE)) == NULL)
      return -1;
    PUT(heap_listp, 0);                            /* alignment padding */
    PUT(heap_listp+WSIZE, PACK(OVERHEAD, 1));  /* prologue header */
    PUT(heap_listp+DSIZE, PACK(OVERHEAD, 1));  /* prologue footer */
    PUT(heap_listp+WSIZE+DSIZE, PACK(0, 1));   /* epilogue header */
    heap_listp += DSIZE;  /* move heap_listp past prologue's header */

    /* extend the empty heap with a free block of CHUNKSIZE bytes */
    if (extend_heap(CHUNKSIZE/WSIZE) == NULL)
      return -1;

    return 0;
}

/* extend_heap requests additional heap space from the mem system,
   rounding up to the nearest multiple of two words

   called here and when malloc is unable to find a suitable fit */
```

```c
static void* coalesce(void* bp) {
    size_t prev_alloc = GET_ALLOC(FTRP(PREV_BLKP(bp)));
    size_t next_alloc = GET_ALLOC(HDRP(NEXT_BLKP(bp)));
    size_t size = GET_SIZE(HDRP(bp));

    if (prev_alloc && next_alloc) { return bp; }

    else if (prev_alloc && !next_alloc) {
      size += GET_SIZE(HDRP(NEXT_BLKP(bp)));
      PUT(HDRP(bp), PACK(size, 0));
      PUT(FTRP(bp), PACK(size ,0));
      return(bp);
    }

    else if (!prev_alloc && next_alloc) {
      size += GET_SIZE(HDRP(PREV_BLKP(bp)));
      PUT(FTRP(bp), PACK(size, 0));
      PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
      return(PREV_BLKP(bp));
    }

    else {
      size += GET_SIZE(HDRP(PREV_BLKP(bp))) +
              GET_SIZE(FTRP(NEXT_BLKP(bp)));
      PUT(HDRP(PREV_BLKP(bp)), PACK(size, 0));
      PUT(FTRP(NEXT_BLKP(bp)), PACK(size, 0));
      return(PREV_BLKP(bp));
    }
}
```

```c
void mm_free(void* bp) {
    size_t size = GET_SIZE(HDRP(bp));
    PUT(HDRP(bp), PACK(size, 0));
    PUT(FTRP(bp), PACK(size, 0));
    coalesce(bp);
}
```

```c
void* mm_malloc(size_t size) {
    size_t asize;      /* adjusted block size */
    size_t extendsize; /* amount to extend heap if no fit */
    char* bp;

    /* Ignore spurious requests */
    if (size <= 0)
      return NULL;

    /* Adjust block size to include overhead and alignment reqs */
    if (size <= DSIZE)
      asize = DSIZE + OVERHEAD;
    else
      asize = DSIZE * ((size + (OVERHEAD) + (DSIZE-1)) / DSIZE);

    /* Search the free list for a fit */
    if ((bp = find_fit(asize)) != NULL) {
      place(bp, asize);
      return bp;
    }

    /* No fit found. Get more memory and place the block */
    extendsize = MAX(asize,CHUNKSIZE);
    if ((bp = extend_heap(extendsize/WSIZE)) == NULL)
      return NULL;
    place(bp, asize);
    return bp;
}
```

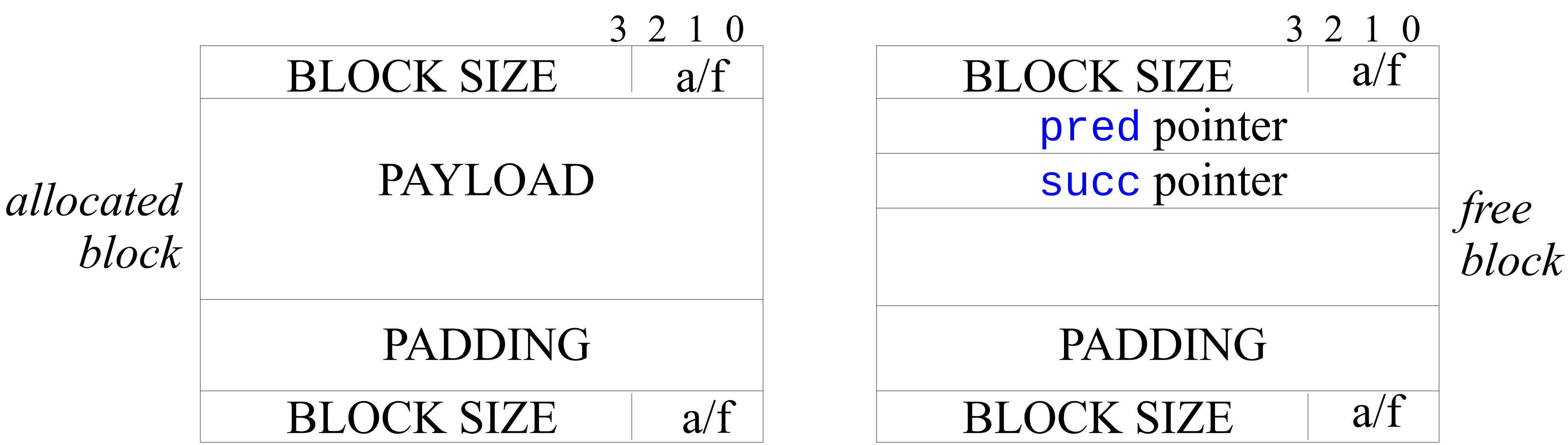
how to implement for first-fit?

how to implement for block splitting?

```c
static void* find_fit(size_t asize) {
  void* bp;

  /* first fit search */
  for(bp = heap_listp; GET_SIZE(HDRP(bp)) > 0; bp = NEXT_BLKP(bp)) {
    if(!GET_ALLOC(HDRP(bp)) && (asize <= GET_SIZE(HDRP(bp)))) {
      return bp;
    }
  }
  return NULL; /* no fit */
}


static void place(void* bp, size_t asize) {
  size_t csize = GET_SIZE(HDRP(bp));

  if((csize - asize) >= (DSIZE + OVERHEAD)) {  /* if new free block */
    PUT(HDRP(bp), PACK(asize, 1));             /* would be at least */
    PUT(FTRP(bp), PACK(asize, 1));             /* as big as min */
    bp = NEXT_BLKP(bp);                        /* block size, split */
    PUT(HDRP(bp), PACK(csize-asize, 0));
    PUT(FTRP(bp), PACK(csize-asize, 0));
  }
  else {                                      /* else, do not split */
    PUT(HDRP(bp), PACK(csize, 1));
    PUT(FTRP(bp), PACK(csize, 1));
  }
}
```

Explicit Free Lists

- With implicit free lists, block allocation is *O*(heap size).
- Better to organize free blocks into some form of explicit data structure.
- The body of a free block can be used to store pointers that implement such a data structure.



Explicit Free Lists

- First-fit allocation time is now *O*(free block count), but time to free a block depends on order in free list.
- LIFO: insert newly freed blocks at beginning of the list.
  - with first-fit, allocator inspects most recently used blocks first
  - cost of `free`?
- Address order: maintain free list such that the address of a block in the list is less than the address of its successor.
  - memory utilization better for first-fit, linear-time `free`
- When is a larger minimum block size required? (with or without boundary tags)

Segregated Free Lists

- To further reduce the allocation time, maintain multiple free lists and contain the search to just one list.
  - Each list holds blocks that are *roughly* the same size.
  - This is known as *segregated storage*.
- There are many ways to define the size classes.
  - powers of 2; e.g., {1}, {2}, {3-4}, {5-8}, {9-15}, ..., {1025-2048}, ...
  - small blocks their own size class, and large blocks by powers of 2
- The allocator maintains an array of free lists, ordered by increasing size.
  - First find list with size class that best fits.
  - Then, what if cannot find block that fits in the list?

Simple Segregated Storage

- Each list holds same-sized blocks.
  - E.g., size class {17-32} all size 32.
- Simply allocate first free block in the appropriate list.
- Do not split or coalesce.
- If a list is empty, request more memory, divide, and link to form list.
- To free, simply insert block at front of appropriate list.
- *Pros*: `malloc` and `free` both O(1), very little block overhead
- *Cons*: susceptible to internal and external fragmentation

Segregated Fits

- Each list holds potentially different-sized blocks with sizes that fit the size class.
  - E.g., block sizes 32 and 40 are in the same list {32-63}.
- Simply allocate first free block in the appropriate list.
  - Split and insert the remaining free block in the appropriate list.
- If a fit is not found in the list, search the next list, and so on.
- If no list has a block that fits, request more memory, allocate the block, and insert remaining free block in appropriate list.
- To free, coalesce and insert resulting block in appropriate list.

`realloc`

```
void* mm_realloc(void *ptr, size_t size);
```

- If `ptr` = NULL, equivalent to `mm_malloc(size)`.
- If `size` = 0, equivalent to `mm_free(ptr)`.
- If `ptr` != NULL, it must have been returned by an earlier call to `mm_malloc` or `mm_realloc`.
  - changes size of the memory block pointed to by `ptr` (the old block) to `size` bytes and returns the address of the new block
  - address of the new block may or may not be the same as the old
  - contents of the new block are the same as those of the old `ptr` block, up to the minimum of the old and new sizes

Notes on Lab 6

- Read and understand every word of section 9.9.
- The code requires error-prone casting and pointer arithmetic.
  - use a debugger to help isolate out-of-bounds memory refs
  - encapsulate pointer casting and arithmetic in macros
- Create and use a heap consistency checker (style points).
- Work in stages.
  - leave `realloc` until the end—only 2 traces require `realloc`
  - build `realloc` on top of existing `malloc` and `free`, then try a stand-alone version for better performance