Announcements

- Midterm #2 is Tuesday
- Program #4 will be on the web soon
- No class next Tuesday
  - see you at the midterm
- Reading
  - Notes (Today)

Implementing Dynamic Allocation

- Where does memory come from?
  - make a call to OS to add memory to running process
    - On Linux, this is called sbrk(int *incr)
      - Adds at least *incr bytes to end of memory
    - Key idea, get big chunks of memory from OS
      - OS calls can be100x slower than local subroutines
      - Hand out small region on demand
- How to manage it?
  - Need to keep data structures to
    - track what memory is in use
    - where are free memory regions
  - Requests to malloc (heap) and new (gc) update structures

Heap Manager Requirements

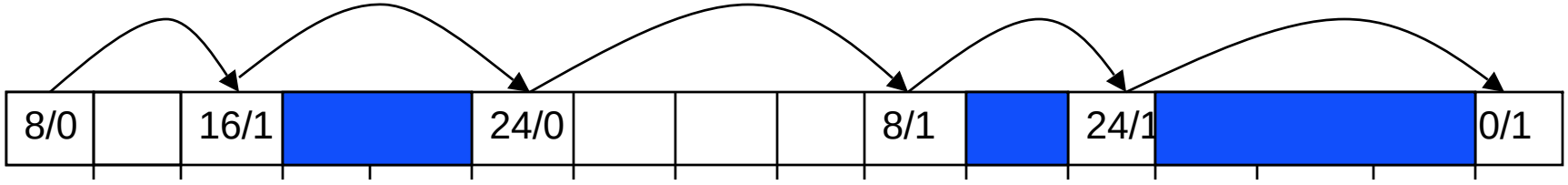
- Handle arbitrary requests
  - other than a free of an address can't occur before the corresponding malloc.
- Must respond immediately
  - can't wait for other requests
- All data it uses must be stored in the heap itself
  - no aux data structures
- Must be aligned so as to hold any data
  - roundup the the next natural alignment if needed
- Can't modify allocated blocks
  - pointers exist in program
  - must only modify free blocks of memory

Heap Manager Goals

- Maximize Throughput
  - need each request to be fast
  - often ok if average is fast (some calls might be slow)
- Minimize Wasted Space
  - What is the peak utilization of memory
  - How much space is wasted at the peak?
  - Avoid *Fragmentation* (wasted space)
    - Internal - allocated space larger than malloc request
    - External - insufficient free space due to many small free blocks

Simple Data Structure for Heap

- Divide heap into blocks
- Each Block has a 32 bite header
  - 29 bits define size of block (in 8 byte chunks)
  - 1 bit is for free/allocated (1 indicates allocated)
  - 2 bits are un-used
- Can find next block if know the current block
  - add address of current block header to size of block



Using Our Simple Heap

- Allocation (malloc)
  - Find a free block
    - start at first block
    - look for one that fits
      - first fit: stop on first block that is big enough
      - best fit: search all blocks and find best fit
  - To split or not?
    - can split free block into two blocks (one free, one used)
      - most efficient packing at first
    - can just waste space after free
      - can lead to fragmentation after multiple malloc/frees
- De-allocation (free)
  - one double word less than pointer is header
  - marks as free

Coalescing Free Blocks

- After free, may have up to three free blocks in a row
  - one before and one after
- Merge a sequence of free blocks into a larger one
- Might causing repeated split/merging
  - consider {malloc(n),free(n)}*
    - if the first malloc involves a split, rest will too
- Coalescing Implementation
  - Update first header block to be size of full free space
  - How to find the previous header?
    - search from start of list
    - include footer at end of block (also with size)
    - include explicit back pointer
      - store address of previous header in header

A Heap Implementation

```
#include <stdio.h>
#include <unistd.h>

typedef struct _header {
    unsigned int size:29;
    unsigned int unusedBits:2;
    unsigned int allocated:1;
    struct _header *prev;
} header;

#define HEAP_INCR (1024 * 1024)
static header *heap;
```

```
void mm_init() {
    header *end;
    heap = (header *) sbrk(HEAP_INCR );
    heap->size = (HEAP_INCR-
            (2*sizeof(header)))>>3;
    heap->allocated = 0;
    heap->prev = NULL;

    end = heap + heap->size - 1;
    end->allocated = 0;
    end->size = 0;
    end->prev = heap;


}
```

```
void *mm_alloc(unsigned int size) {
    header *curr, *temp, *next;
    if (!heap) mm_init();
    for (curr=heap;; curr = curr + curr->size + 1) {
        if (curr->size >= size && !curr->allocated) {
            temp = curr + ((size + 7) >> 3) + 1;
            next = curr + curr->size + 1;
            if (temp < (next - 1)) {
                /* split current node */
                temp->size = next - temp - 1;
                temp->allocated = 0;
                temp->prev = curr;
                next->prev = temp;
                curr->size = temp - curr - 1;
                curr->allocated = 1;
            }
            return ((void *)(curr + 1));
```

mm_allocate (cont)

```
} else if (!curr->size) {
        /* no room in the heap, grow heap */
        temp = (header *) sbrk(HEAP_INCR);
        if (!temp) return NULL;
        curr->size = (HEAP_INCR-  sizeof(header))>>3;
        curr->allocated = 0;
        /* update new end */
      }
    }
}
```

```
void mm_free(void *ptr) {
    header *curr, *prev, *next, *nn;
    curr = ((header *) ptr) - 1;
    prev = curr->prev;
    next = curr + curr->size + 1;
    curr->allocated = 0;
    if (!next->allocated) {  /* merge next into current */
        curr->size += next->size + 1;
        nn = next + next->size + 1;
        nn->prev = curr;
        next = nn;      /* for double merge case */
    }
    if (prev && !prev->allocated) {  /* merge current into prev */
        prev->size += curr->size + 1;
        next->prev = prev;
    }
}
```

Alternative: power of two sized space

- Maintain separate free lists of different sized objects
  - 8, 16, 32, 64, 128, 256, 512, 1024, 4096 bytes for example
- Never Split a block
- If a free list of a given size is empty
  - use sbrk to get more storage (some increment of size)
  - create a new free list of these items
- Free
  - add back to free list for target size
  - if all items on a page (increment size) are free, release to common free pool