Project #3:
Memory Allocation

Justin Chen (juychen@ucsc.edu)
Derek Frank (dmfrank@ucsc.edu)
Benjamin Chow (bchow@ucsc.edu)

1. Goals

Implement a user level library that will manage memory using three different allocation methods. This user level library must be able to run on the MINIX3 kernel.

2. Available Resources

Uses 1 contiguous block of memory using malloc.

3. Design

Memory Allocation implementation

Our user-level library must be able to support the following three types of memory allocation:

- Buddy Allocation
- Slab Allocation
- Free-list Allocation

The library is called `libmem.c` and will be compiled into a static library `libmem.a`
Access to this library can be called by adding a `-lmem` option to the end of a command line

3.1 Design of each function

3.1.1 libmem.c

We have included all the code necessary to run the memory allocation in `libmem.c`

There are three major calls that can be made in this library:

```
int meminit(long n_bytes, unsigned int flags, int parm1, int *parm2)
void *memalloc (int handle, long n_bytes)
void memfree (void *region)
```

3.1.2 Major calls

`int meminit(long n_bytes, unsigned int flags, int parm1, int *parm2)`

```
     Check the number of
```

`void *memalloc (int handle, long n_bytes)`

`void memfree (void *region)`

2.1.3 Buddy Allocation

Goals:
Implement a buddy allocator to store memory of varying sizes and report the amount of memory segmentation that results.

Available Resources:
Uses 1 contiguous block of memory using malloc. All resources in the buddy allocator are used inside that block.

Design:
Our buddy allocator uses a bitmap to mark each block as used or free. It starts with level 0 with 1 block representing the entire memory, then level 1 has 2 blocks with 50%, and so on. Buddy_alloc and buddy_free aren called when the user uses memalloc or memfree, and the rest of the functions are used to implement those two.

`int getBMapBit (allocator, int level, int position)`

```
This function gets the bit from the buddy bitmap at the input level
and position.
It calculates the corresponding location in the bitmap, gets the
integer from the bitmap array, and extracts the bit value using
bitwise AND.
```

`int getBMapBuddyBit (allocator, int level, int position)`

```
This function simply calculates the buddy of the input block, and then
calls getBMapBit on the buddy to get the buddy's value.
```

`void setBMapBit (allocator, level, position, value)`

```
This function does the same thing as getBMapBit, except it sets the
value of the bit instead of getting it.
```

void* getMemoryAddress (allocator, level, position)
This function gets the actual memory address of the block of memory when given the level and position. It is used to return the address to the user after the memory is allocated.

void divide (allocator, level, position)
This function divides the memory block into two smaller memory blocks. It does this by setting the larger block's bit to 1, and setting the two children's bits to 0.

void *buddy_alloc (allocator, long n_bytes)
This function allocates the given amount of memory and returns the address to the user.

- It starts by calculating the smallest page size that will fit n_bytes.
- Then, it searches for an empty block (bit is 0 and buddy is 1, or it's the level 0 block).
  - If there are no empty blocks of that size
    Search a level down (twice as large blocks) until it finds one.
    Once it finds the block, it will divide it repeatedly until the target block is the smallest page size again.
    Set the block to 1 in the bitmap, and returns the address using getMemoryAddress.

void buddy_free (allocator, void *region)
This function takes a memory address and frees the memory from the bitmap, allowing the memory to be allocated later.

- It first calculates the corresponding level and position, and then sets the bit to 0.
- Then, it checks if the buddy is 0 as well, and merges the blocks together by setting the parent to 0 if necessary.
- The function will continue to merge until it reaches a buddy that is used, or level 0.

3.1.4 Slab Allocation

Goal:
The goal of the slab allocator is to efficiently manage memory with a minimal worst-case amount of overhead.

Available Resources:

- Will use one call to malloc() from within meminit() to attain memory to manage.
- Will be using a free-list approach to managing the free space.

Design:

The overall design is to use a free-list approach to manage the objects within slabs. Overhead will primarily consist of 4 bytes per slab for what object size is contained in the slab and 4 bytes per slab for a pointer to the first free object within the slab. The idea is to allocate a structure directly preceding the managed memory that will contain necessary overhead as well as 8 bytes inside each slab preceding all objects for slab information.

Structures:

```
struct allocator:
   NOTE: only what is used/important for slab allocator

int handle
   The handle of the current allocator.

n_bytes
   The size in bytes of memory managed by current allocator.

unsigned int flags
   Flags that specify the type of allocator being used to manage
   memory.
          Buddy:     0x01
          Slab:      0x02
          Free-list: 0x04
          First fit: 0x00
          Next fit:  0x08
          Best fit:  0x10
          Worst fit: 0x18

int parm1
    Buddy: The minimum page size in address bits
              (i.e., 12 bits --> 2^12 bytes).
    Slab: The number of pages used to make a slab.
    Free-list: The size of the segment the user wants
int *parm2
    Slab: An array specifying the possible object sizes.
void *memory
     A pointer that points to the beginning of the managed memory,
     which begins immediately after the structure.
```

VARIABLES:

```
    - Found either at the beginning 6 bytes of reserved space in a
      slab or in free objects, which are treated as free-lists.

    int objsize
      A value reserved in the first 4 bytes of each slab holding the
      current object sizes inside the slab or zero if the slab is
      free.
    void *freeptr
      A value reserved in 4 bytes directly after the objsize of each
      slab holding a pointer to the beginning of the slab's free list
```

of free objects.
void *nextfreeptr
A pointer of 4 bytes contained in each free object. Links the free-list. Null indicates the end of the free-list.

FUNCTIONS:

slab_init (long n_bytes, unsigned int flags, int parm1, int *parm2):
- The user must initialize the memory to be managed with the following.

Given:

long n_bytes
The user must give the number of bytes this slab allocator will manage.

unsigned int flags
The user must specify the use of slab allocator.

int parm1
The user must specify how many 4 KB pages are used to make slab.

int *parm2
The user must give an array of object sizes ending in 0.

Variables:

int handle
The allocator handle that will be returned.

int slabsize
The size of a slab in bytes. Must be a multiple of 4 KB.

int structsize
The size of the overhead struct in bytes (sizeof (struct allocator)).

allocator_ref myAlloc
A structure reference to the current section of managed memory.

int numslabs
The number of slabs able to fit in n_bytes minus structsize.
Found by ((n_bytes - structsize) / slabsize)

Pseudocode:

Error checking: return -1
parm1 must not be negative or zero.
n_bytes must be strictly larger than slabsize (parm1 * 4KB) to allow for overhead, i.e., at least slabsize plus structsize.
parm2 cannot have any negative numbers
parm2 cannot be infinite in length and must end in a 0
Cannot trust user to end in a 0 so, only accept up to 256 values. Max parm2 length is subject to change.
Must have at least one object size specified in parm2.
No object can exceed the slabsize minus 8 bytes, since two

    variables are maintained in the first 8 bytes of each slab.
  The smallest object must be at least 4 bytes (size of a pointer).
  Set the allocator current handle to be the next available handle.
  Allocate n_bytes of memory requested by the user for myAlloc.
  Set myAlloc->memory to point at the address directly after itself.
    This is where the first slab will begin.
  For each slab (begin at myAlloc->memory and separated by slabsize)
    Treat the first 4 bytes as and int and set it to zero to mark
    the slab as both free and containing no objects or object size.
    This "int" holds the object size in bytes of the slab when the
    slab is being used. Otherwise is zero.
  Treat the following 4 bytes as a pointer. This "pointer" will
    be set to the first address directly following itself. This
    pointer is used to locate a free object within the slab. It
    will be null when there are no available objects within the
    slab.

Returns: An integer handle of the initialized managed memory. A
    negative number is returned for errors.

slab_alloc (int handle, long n_bytes):
- Allocates and returns a specified amount of memory for use by
  the user.

Given:
  handle
    An integer handle specifying the section of managed memory.
  n_bytes
    The amount of memory in bytes to be allocated for the user.

Variables:
  int slabsize
    The size of a slab in bytes. Must be a multiple of 4 KB.
  int curobjsize
    The smallest possible object size to fit the requested n_bytes.
    Determined by scanning through parm2 and maintaining the smallest
    size that will fit n_bytes. Cannot be smaller than 4 bytes.
  void *slabptr
    A pointer to the beginning of a slab. slabptr should be
    incremented by slabsize to move between slabs.
  void *freeptr
    The pointer to a free object inside of a slab. Set when
    examining a slab as the second set of 4 bytes at the beginning
    of the slab. Also is slabptr plus 4 bytes.
  void *nextfreeptr
    If an object is free within a slab, then its first 4 bytes will
    contain a pointer either to the next free object or null if there
    are no more free objects. This pointer should be the value found
    in the free object pointed at by freeptr.
  void *objptr
    A pointer to the object being returned.

Pseudocode:

```
  Error checking: return null pointer
    n_bytes must be smaller than at least one object size in parm2
    Check if curobjsize can fit inside a used slab of objsize or
there is a free slab to use for objsize.  May require two loops.
  If there is a slab that contains objects of objsize and can fit
    another object because its freeptr is not null.
    Then objptr becomes freeptr and freeptr becomes nextfreeptr.
  Else if there is a free slab
    Set its objsize.
    Set all objects in the slab to free objects by beginning at the
slabs freeptr and incrementing by objsize.  The first 4 bytes
of each free object are used as pointers to the next free
object.  freeptr points at the first one.  The last free object
has its nextfreeptr set to null.
    objptr is set to the freeptr of this slab and freeptr is set to
the nextfreeptr within the object just released.
  Else there is no slab to put the object so return a null pointer.
  Return objptr.
```

Returns: A pointer to the beginning of the section of allocated memory. Return a null pointer for errors.

slab_free (allocator_ref myAlloc, void *region):

Frees the specified region of memory if the region exists.

Given:

```
  allocator_ref myAlloc
    A structure reference to the allocator managing the given region.
    Found by iterating through each allocator and seeing if the
    region is within its bounds.
  void *region
    A pointer that must point at the beginning of the region the
    user wishes to free.
```

Variables:

```
  int slabsize
    The size of a slab in bytes.  Must be a multiple of 4 KB.  Found
    by multiplying 4 KB by myAlloc->parm1.
  void *slabptr
    A pointer to the slab the region is contained in.  Found by
    (slabsize * floor((region - myAlloc->memory) / slabsize)).
  int objsize
    The object sizes contained in the slab pointed at by slabptr.
    Found in the slab's first 4 bytes.
  void *freeptr
    A pointer to the first free object in a linked list of free
    objects inside the slab pointed at by slabptr.  Found in the
    4 bytes directly after the objsize.
  void *nextfreeptr
    An arbitrary pointer to a free object found only inside a free
```

object. Is set to null when the object is the last free object in a slab's free-list. Must either be null or point to another free object within the same slab.

void *objptr

A pointer to the object the region is contained in. Found by (objsize * floor((region - slabptr - 8 bytes) / objsize)).

int numobjects

The number of objects in a slab of objsize. If the number of free objects in the slab is equal to numobjects, then the slab can be freed and released. Found by ((slabsize - 8 bytes) / objsize). int will take the floor of the result.

Pseudocode:

```
  Error checking: return nothing
    region must be in a slab and after the first 8 bytes of the slab.
    region must point at the beginning of an object, nowhere else so
as to prevent any sort of errors.
  If the object pointed at by objptr equals the region pointer
    Set the first 4 bytes of the object equal to the freeptr of the
    slab.
    Set the slab's freeptr to the free object/region/objptr.
    Count the number of free objects by counting the number of
dereferenced pointers starting with freeptr.  A null pointer
indicates the end of the free list.
    If the number of free objects equals numobjects
Free the slab by changing its objsize to zero and its freeptr
  to the address directly following the first 8 bytes of the
  slab.
```

Returns: Returns nothing on success or error.

3.1.4 Free-list Allocation

private int freelist_init (long n_bytes, unsigned int flags, int parm1, int *parm2)

-Check to make sure that the user has a valid allocation size. If not then return an error
-Set all the struct parameters to their respective fields
-Calculate the remaining free space (subtract the overhead)
-Store this in a 4 byte block in the beginning of the free space
-Set the pointer to free space 4 bytes after the size block
-Return this pointer

private void *firstfit_alloc (allocator_ref myAlloc, long n_bytes)

- Start at the head of the list and go down
    - Check if you find a segment of memory that is big enough to contain the user n_bytes

```
        YES: - Set the size size of the 4 byte block to the user
               n_bytes
             - Advance the pointer n_bytes (to accommodate the user
               space)
             - Set the remaining freespace size.
               Freespace block - n_bytes - 4
               We subtract 4 because that is the size needed to hold
               the freespace size.
            - Check if this is the head.
                 YES: set the head to the pointer + n_bytes location.
                  NO: set the previous to the pointer + n_bytes
       NO:  Return NULL, no space large enough is available.
```

private void *nextfit_alloc (allocator_ref myAlloc, long n_bytes)

```
 - Set global varaible void * nextfit to NULL
 - If nextfit == NULL
   YES: Set nextfit = head of free list
 - Loop through the free list from the current position
 - Check if you find a segment of memory that is big enough to
   contain the user n_bytes
        YES: - Set the size size of the 4 byte block to the user
               n_bytes
             - Advance the pointer n_bytes (to accommodate the user
               space)
             - Set the remaining freespace size.
               Freespace block - n_bytes - 4
               We subtract 4 because that is the size needed to hold
               the freespace size.
            - Check if this is the head.
                 YES: set the head to the pointer + n_bytes location.
                      set the nextfit to the pointer + n_bytes
                  NO: set the previous to the pointer + n_bytes
                     set the nextfit to the pointer + n_bytes
       NO:  Return NULL, no space large enough is available.
```

private void *bestfit_alloc (allocator_ref myAlloc, long n_bytes)

```
-Loop through the freespace list keeping track of which block is the
best fit.
- To do this, take the size of the current free space block and
subtract it from the user n_bytes, giving you the difference
- The bestfit block is stored in void * bestfit.
- Keep track of the smallest difference, every time you update the
smallest difference, set the bestfit to that current block in the list
-When you are done looping through the entire list, allocate using the
*bestfit
     To allocate:
         -Set the size of the 4 byte block to the user specificed
          n_bytes
         -Advance the pointer n_bytes (to accomodate the user
          space)
          - Set the remaining freespace size.
              Freespace block - n_bytes - 4
```

We subtract 4 because that is the size needed to hold the freespace size.
- Check if the bestfit is the head of the list
    YES: set the head to the pointer + n_bytes location.
         set the bestfit to the pointer + n_bytes
    NO: set the previous to the pointer + n_bytes
        set the bestfit to the pointer + n_bytes

private void *worstfit_alloc (allocator_ref myAlloc, long n_bytes)
-Loop through the freespace list keeping track of which block is the biggest.
- To do this, we keep track of which block is the biggest, stored into a temporary variable.
- The worstfit block is stored in void * worstfit.
- Keep track of the biggest block, every time you update the biggest block, set the worstfit to that current block in the list
-When you are done looping through the entire list, allocate using the *worstfit

To allocate:
- Set the size of the 4 byte block to the user specificed n_bytes
- Advance the pointer n_bytes (to accomodate the user space)
- Set the remaining freespace size.
  Freespace block - n_bytes - 4
  We subtract 4 because that is the size needed to hold the freespace size.
- Check if the worstfit is the head of the list
    YES: set the head to the pointer + n_bytes location.
         set the bestfit to the pointer + n_bytes
    NO: set the previous to the pointer + n_bytes
        set the bestfit to the pointer + n_bytes

private void freelist_free (allocator_ref myAlloc, void *region)
We will loop through the free list
- Keep looping until region > current block
First check the left side. Does previous + size == left?
YES: It means that the previous block is the neighbor of this region. Add the size of the region to the size of the prev block. Nothing else changes.
NO: It means that the previous block is not a neighbor.
- Store the value of the pointer in the left for the value of the pointer on this region.
- Set the value of the pointer on the left to point at the region

Now check the right side. Does region + size + 4 == right?
YES: It means that the next block is the neighbor of this region. Add the size of the region on the right to the size of the user-freed region.

```
-Store the location (that the next node was pointing) in
  region.
```

4. Testing

To test our project, we created a buddy allocator, a slab allocator, and a freelist allocator, all with the same size. Then, we ran the same allocations on each one, allocating a random amount of bytes from 1 to 100, thirty times. Then, we freed every other one, and reallocated them. This helps to test the efficiency of the allocators in filling in gaps. Finally, we displayed the amount of memory currently in use by each allocator compared to the amount that we requested, to measure the total amount of memory wasted by each allocator. We also ran this same process with bytes all of the same length, in order to compare performance when the variables all have the same size. We found that X performed the best for the first test.