Programming Project #3
CMPS 111, Winter 2014

The goal of this assignment is to implement a memory allocation library with two different allocation mechanisms. This is user-level code, but must be run under MINIX3.

Basics

This assignment doesn't require you to modify the kernel, but does require that you write code to manage memory in user space. The code you're going to write must support two different types of memory allocators, each of which we've covered in class: buddy and free list-based allocation. Your code should allocate a *single* large region from the kernel, and then manage it using one of these allocation mechanisms.

Details

Your code is going to be in a library, which means you have to create a `libmem.a` file that can be accessed on the command line with the `-lmem` option. You may implement your code in a single `.o` files, but your `Makefile` must create the `libmem.a` file for user programs to use.

In addition, you're going to report on how well `libmem.a` handles different types of memory allocation / freeing workloads. We'll suggest some workloads; you should create additional workloads, and write up the results of testing your code.

`libmem.a`

`libmem.a` will have three calls:

`int meminit(long n_bytes, unsigned int flags, int parm1, int *parm2)`
: The goal of `meminit` is to initialize your memory allocator. The *n_bytes* parameter is the number of bytes that this allocator should manage. *flags* describe the behavior of the allocator (see below for values). *parm1* and *parm2* are parameters that can be used to customize a memory allocator. `meminit()` returns a handle that can be used to identify the allocator; a single process might choose to use more than one allocator. This handle must be at least 0; a negative number is returned to indicate an error.

`void *memalloc (int handle, long n_bytes)`
: `memalloc()` takes a handle (from `meminit()` and returns a pointer to a region of *n_bytes* of data. If no memory is available, or the handle is invalid, it returns NULL (void *0).

`void memfree (void *region)`
: `memfree()` frees a previously-allocated region of memory, returning it to the pool of available memory for the appropriate handle. Note that the handle itself is *not* passed; `memfree()` must figure out how to associate the region with the appropriate handle. There is no return value; if the region isn't valid, the call has no effect.

The array of values (and length) passed to `meminit()` is used by several of the memory allocators to determine how to set up the allocator. Details on this can be found in the description of each allocator.

Buddy Allocator

The buddy allocator works in exactly the way we described in class. `parm1` is the minimum page size, in *address bits*. In other words, if the minimum page size is specified as 12, the minimum page size is actually 2^{12} bytes, or 4KB. If the region to divide up isn't a power of 2, return an error (-1). `flags` should be set to 0x1 to create this allocator.

Your allocator should start with a memory region whose size is passed (return -1 if the region size isn't a power of 2), and successively split it. You should use as little space as possible to keep track of memory in use; the typical approach is to use one bitmap per page size, with a single bit for each page of that size. In other words, if there are 128 pages of 4KB, the 4KB page bitmap would have 128 entries. For the bitmap for the smallest-size page, a 1 means the page is in use, and a 0 means the page is free or unallocated because the parent page hasn't been split. The same is true for all levels above the smallest page size. For example, a 1 entry at the 8KB level means that at least one of its (two) children must be allocated, and hence set to 1.

To allocate a block of a particular size 2^k, check the bitmap for that size block. If there are any blocks marked free whose buddy is marked in use, allocate that block. If no such block is found, there are no free blocks of size 2^k, so check blocks of size 2^{k+1} in the same way. If one is found, mark it in use (set the bitmap entry to 1) and split it, marking one child allocated (return this one) and the other one free. This continues recursively up the bitmaps until the root is reached.

When a block is freed, check its buddy to see if it's free as well. If so, merge the two blocks and mark the parent free. This goes on recursively until you reach the top level bitmap. Note that space must be allocated on 2^x byte boundaries. This fact can be used to free a block when its size is unknown. Simply start at the lowest level (smallest page size) of bitmap, and see if the corresponding entry is in use. If it is, mark it free and merge upwards. If not, try the next higher bitmap, and so on.

Free-list Based Allocator

Once again, this was discussed in class. You need to allocate objects of whatever size is passed. Here, you may need to store the size before the start of an allocated region. This is typically done by allocating a region of size *n+4*, setting the first four bytes of the region to the size, and return a pointer to the region *after* where you stored your size.

You need to implement the first-fit, next-fit, best-fit, worst-fit, and random-fit algorithms for this allocator. `flags` should be set to 0x4 to use this allocator, and you should OR in 0x0, 0x08, 0x10, 0x20, or 0x40 for first-fit, next-fit, best-fit, worst-fit, and random-fit respectively.

You can find discussions of these algorithms in either Tanenbaum text (or in just about any other operating systems text). Briefly, first fit chooses the first hole in which a segment will fit, starting its search from the beginning of the free list each time; next fit is the same as first fit, except the search starts from the last hole you picked; best fit chooses the hole that is the tightest fit; and, conversely, *worst fit* chooses the hole that is loosest fit; random fit chooses a random hole among those that the page will fit into, you can design any algorithm you would like to manage and select the random hole.

Testing and Benchmarking

Once you have your allocator working, you'll need to test it to see how well it allocates objects of identical or mixed sizes. Write some test programs to do this. You'll also want to write code that can return information about memory usage and efficiency for each allocator. For each allocator, you'll want to know how much memory of the original allocation is actually in use, both directly (for objects) and indirectly (internal fragmentation). You'll have to write some code to track this information for each allocator. However, the interface to do this is up to you—our code won't call it.

You'll next need to run some experiments. One required experiment is to compare best fit, random fit, and worst fit. Create a fixed workload and run it once using best fit, once using worst fit, and ten times using random fit. Record the number and size of holes left by each strategy after the workload terminates. Calculate the statistics: min, max, median, mean, and standard deviation of the number and size of holes.

In addition, make up one or more of your own experiments that show situations under which each allocator might do better. Here is a suggestion: Under what kind of workload does buddy allocation work well?

Write up your findings, visually represent the statistics in graphs, and turn this in as a report with your project.

Deliverables

You must turn in a compressed (`gzip`ped) `tar` file of your project directory, including your design document and your report. Please ensure that the directory contains no machine-generated files when you create the tar file; you might want to add a special target (usually called `clean` to your `makefile` to handle this task). In addition, include a `README` file (place it in the project directory that you tar up) to explain anything unusual to the teaching assistant. **Do not submit object files, assembler files, or executables.** Every file in the submit directory that could be generated automatically by the compiler or assembler will result in a 5 point deduction from your programming assignment grade. Also, the names and UCSC accounts of all group members must be in each file you modify.

Your design document should be called `DESIGN` (if in plain text), or `design.pdf` (if in Adobe PDF) and should reside in the top-level project directory. Formats other than plain text or PDF are not acceptable; please convert other formats (Word, LaTeX, HTML) to PDF. Your design should describe the design of your assignment in enough detail that a knowledgeable programmer could duplicate your work. This includes descriptions of the data structures you use, all non-trivial algorithms and formulas, and a description of each function including its purpose, inputs, outputs, and assumptions it makes about the inputs or outputs.

Your project report should be in PDF, since it should (hopefully) have graphs. Please call it `report.pdf` and put it into the top level directory in the tar file.

Project Groups

You may do this project, as well as the fourth project with a project partner of your choice. However, you can't switch partners if you already had a partner for Project #2. If you choose to work with a partner (and we encourage it), you both receive the same grade for the project. One of you should turn in a single file called `PARTNER` with the name and CATS account of your partner. The other partner should turn in files as above. Please make sure that both partners' names and accounts appear on all project files.

If a group decides to use grace days to turn in their project late, **each student on the project must use a grace day for every day the project is late (e. g., a three-student project turned in three days late requires nine grace days, three from each student).** Recall that late projects (those turned in after the due date, which may have been extended by using grace days) receive a grade of 1 point, regardless of quality.

Hints

- Go to lab and office hours for help on how to approach this assignment.
- Do your design document first. The code runs at user level, but a design will help.
- Get one allocator working at a time, and then glue them together through `meminit()`. Similarly, get multiple allocators working simultaneously after you get one working.
- Write simple user-level programs to test your code.
- **START EARLY!**