CSci 4061: Introduction to Operating Systems

Assignment 3: Dynamic Memory Management

Due: Nov 19th at 11:55 pm. You may work in a group of 2 or 3.

Purpose: In this assignment, you will become more familiar with the issues that surround dynamic memory management: dynamic memory allocation and deallocation. Understanding these issues is important in designing memory-efficient run-time systems, an important task of the systems programming. You will make use of Unix library/system calls for memory management: {re}malloc, free, ... In addition, you will also learn about interrupt-driven programming (via alarm signals) and separately compiled functions.

Description

1. Dynamic Memory Management Functions

Dynamic memory allocation and deallocation in Unix is achieved via the `malloc` family of system calls along with `free`. However, for programs that wish to perform a great deal of allocation and deallocation, this introduces a lot of overhead due to the expense of making library/system calls. In addition, in some environments `malloc` may not be *thread-safe* or *signal-handler-safe*, meaning that dynamic memory allocation may not work correctly if called by threads or within signal handlers because of race conditions in the heap management routines.

To combat these problems, you decide to write your own dynamic memory manager and make it available to applications that wish to utilize dynamic memory. Your memory manager will "manage" a pool of dynamic memory divided into chunks of variable size.

Here is the interface of your memory manager. You must implement the following functions of the memory manager:

```
int mm_init (mm_t *MM, int mem_size);
   // allocate all memory, returns -1 on failure
void *mm_get (mm_t *MM, int sz);
   // get a chunk of memory (pointer to void) of size sz, NULL on failure
void mm_put (mm_t *MM, void *chunk);
   // give back chunk to the memory manager, don't free it though!
void mm_release (mm_t *MM);
   // release all memory back to the system
```

Note that the `mm_init` function should allocate ALL of the memory ahead of time that the manager will use for `mm_get` and `mm_put`. This creates an upper bound to the total amount of memory that this `mm_t` can give out, which is not *truly* dynamic memory allocation, but is close enough for our purposes.

The idea is that an application would declare a variable of type `mm_t` for *each separate* dynamic data structure they wished to manage. The main program would then initialize this variable by calling `mm_init`. After that, calls could be made from threads, signal handlers, or any other functions, to get or put memory chunks as needed by the program.

Prior to defining any of these functions you will need to define the type mm_t. You will need to keep track of status of memory chunks (free or taken). Within mm_t, you are not allowed to use arrays of a fixed-size length. (You cannot assume anything about the maximum number of chunks.) On a mm_get, you should return the first chunk (starting from the lowest address) that is big enough. It is important that you do this, so that we can grade your program on test cases. This will likely create a new "hole" that you should keep track of. On a mm_put, a new hole is created that *may* need to be merged with adjacent holes (to the left and/or right). We recommend that you define a doubly-linked list of chunks that can be marked taken or free (holes). This will enable easy merging. But you are free to use any data structure you like. The mm_put function will only be able to put back what we previously received from a call to mm_get.

We have provided the basic framework of these functions for you. We have left code comments for you to fill in your code in specific areas of these files. The file mm_public.c will contain your implementation of the memory manager routines and a timer that you can use. The file mm_public.h contains the definition of mm_t, which you must come up with. You will be provided with a Makefile which will compile your programs. To compile, simply type `make mm`.

Extra Credit: Is your memory manager more efficient than native malloc/free? We will provide you with a file main_malloc.c, so you can compare the performances of the two programs. This will not be easy to achieve without significant optimization to the data structures (and I'm not completely sure it is doable). It must be more efficient for an arbitrary sequence of mm_get/mm_put's (including mm_init) to get credit.

2. Using the Memory Manager

Now you will use your memory manager for a fairly sophisticated application. As stated in the introduction, the use of malloc in signal handlers may be a problem, so it may be useful to use your memory manager within a signal handler to return allocated memory.

You will implement a simple interrupt-driven message-handling capability using signals. Your application is a simple "server" that will receive a message and simply print its contents. The message is sent to the server in the form of packets (fixed-size message fragments, numbered 0, 1, ...) that arrive asynchronously. When all of the packets have arrived at your server, the message can printed out. **Note:** messages will arrive one-at-a-time for simplicity, but packets may arrive out-of-order, just like on the Internet. You must assemble the message in the proper packet order. You will use your memory manager to store the message contents. You should allocate the message contiguously using memory provided by the memory manager.

To model network communication to the server, packets will arrive asynchronously whenever an alarm goes off (SIGLARM). In reality, network communication would raise SIGIO to your process, but we will use SIGALRM for simplicity. We will provide the code that sends packets to your application. You will be manipulating pointers quite a bit including some elementary pointer arithmetic. We have provided a code-shell (packet_public.c) that requires you modify a few functions including the handler. We have also provided packet_public.h, which contains all of the packet/message structure definitions. To compile this program, type `make packet`. You can also type `make` to compile all programs at once.

We will be testing the memory manager and packet functionality with completely separate tests. When writing your code, do not assume that message types are the only kind of object that we will store in memory.

3. Deliverables

1. The files containing your code.

2. A README file.

All files should be zipped up in a .tar.gz or .zip file and submitted via Moodle. This is your official submission that we will grade. Please do not send your deliverables to the TAs. Keep in mind that the submission link will disappear right at 11:55 p.m.

4. Documentation

You must include a README file which describes your program. It must contain:

1. A list of all files included in your submission.
2. How to use the program from the shell.
3. What exactly your program does (briefly).
4. Which group member was responsible for which part of the project.

The README file should not be too long, as long as it properly describes the above points. Proper in this case means that a first-time user will be able to answer the above questions without any confusion after reading the README. Within your code you should write detailed comments, although you don't need to comment every line of your code.

At the top of your README file and main C source file, please include the following comment:

```
/* CSci4061 F2012 Assignment 3
* section: one_digit_number
* login: cselabs_login_name (usually your x500)
* date: mm/dd/yy
* names: full_name1, full_name2, ... (for partners)
* id: id_for_first_name, id_for_partner, ... (for partners)
*/
```

Grading

5% README file - tell us who did what
20% Documentation within code, coding and style
  (indentations, readability of code, use of defined constants rather than numbers)
75% Test cases
  (correctness, error handing, meeting the specifications)

Please make sure to pay attention to documentation and coding style. A perfectly working program will not receive full credit if it is undocumented and very difficult to read. We will use GCC version 4.5.2 to compile your code. The grading will be done on CSELabs machines only.