CSCI 347 Computer Systems II
Summer 2018
Assignment 1

Submitting Your Work

This assignment is worth 15% of the grade for the course. Save your program files (including header files and make file) in a zipped tar file and submit the file via the **Assignment 1 Submission** item on the course web site. You must submit your assignment by 2:00 pm on Monday, July 23.

Your assignments will be evaluated on correct functionality and conformance to the coding standards described at the end of this assignment specification.

Heap Management

Your task for this assignment is to write alternatives to the standard library functions malloc() and free(), used for heap management. Your functions malloc347() and free347(), declared in the supplied header file heap347.h, completely replace malloc() and free() and are to be completely responsible for heap management. You may not use malloc() and free(), but you may use sbrk() to request the kernel to allocate heap space, as needed by malloc347().

```
void *malloc347(int nbytes);
```

requests `nbytes` bytes of heap-space. If successful, malloc347() returns the address of the allocated space; if unsuccessful (not able to allocate the space), malloc347() returns NULL.

```
void free347(void *ptr);
```

releases the previously-allocated heap-space at the address specified by `ptr`. The heap manager must enable the freed space to be used for subsequent calls to malloc347().

Your heap management functions will be tested by a supplied module HeapTestEngine.o, which you must link in with the object file from your heap management routines. HeapTestEngine.o provides two functions, declared in the header file `HeapTestEngine.h`, for use by your program:

```
void init_heap_test();
void heap_test();
```

These function are to be called by your `main()` function. `init_heap_test()` must be called before `heap_test()` and performs some initialization. `heap_test()` will make many calls to `malloc347()` and `free347()` to exercise your heap management routines. You may be assured that `heap_test()` will

never make a call to `free347()` with an address that was not previously allocated by malloc347(). Parameters for the heap test engine are specified in the supplied data file `config.txt`, described below.

Since the calls from the heap test engine to `malloc347()` and `free347()` will be randomly interspersed, your heap management software needs to keep track of available heap-space. This should be done with a linked list of blocks, each having a size and a pointer to the next block in the list. When a request for space is received, through a call to `malloc347()`, your heap manager must first attempt to satisfy the request from its areas of available heap-space. If there is no block of sufficient free-space, `malloc347()` is to call `sbrk()` to request an increase in the available heap space. For this assignment, no call to `sbrk()` may request more than 1024 bytes.

There are 4 common algorithms for allocating space from the free-list:

- First-fit: store the blocks of free space in order of memory address and use the first block in the list that satisfies the request. If that block has more space than requested, break it into two areas: one being the same block of free space, but reduced in size, and the other returned in response to the request.
- Best-fit: use the block of free space which is large enough for the request but leaves minimal space left over.
- Worst-fit: use the largest block of free space, thus leaving the maximum space left over.
- Quick-fit: an alternative algorithm, developed by Weinstock and Wulf, described below and in their paper, available on the course web site.

A solution for First-Fit is supplied. Your task is to implement the Quick-Fit algorithm.

Kernighan & Ritchie, "The C Programming Language", describes a way of implementing the heap management functions for the first-fit method. That section of their book is available on the course web site for your information.

Challenges in Heap Management

There are three main challenges in heap management:

1. Aligning allocated memory with memory word boundaries appropriate for the data objects to be stored in the allocated space
2. Memory fragmentation.
3. Memory leakage.

Memory Alignment

As described by K&R, one problem to be faced in heap-space management is to ensure that storage allocated by `malloc347()` is aligned properly with memory word boundaries appropriate for the data objects that will be stored in it. This alignment must be done for the worst case which, on a 64-bit system, is for types double, long

and any pointer type, each of which requires 8 bytes. Given the block header format suggested by K&R, in using their method on a 64-bit system, we should allocate space in 16-byte units. So, for example, a request for 100 bytes would require one 16-byte unit for the header plus seven 16-byte units for the allocation: a total of eight 16-byte units or 128 bytes.

Memory Fragmentation

With random interspersing of calls to `malloc347()` and `free347()`, the available heap-space will soon become fragmented – broken into small pieces, none of which may be large enough to satisfy a subsequent request to `malloc347()`. Therefore, heap management routines must look for opportunities to coalesce neighboring blocks of free space whenever a call to `free347()` releases a block of heap-space.

Memory Leakage

Your heap management functions must avoid accidental memory leakage. Every byte allocated by `sbrk()` must be accounted for at all times, either in space allocated by `malloc347()` or on the list of free blocks.

Quick-Fit

The basis of Quick-fit is an observation by Weinstock and Wulf that in the vast majority of application programs:

- There only a few data types for which space is requested on the heap.
- The most frequently allocated types involve relatively small, fixed amounts of storage.

The algorithm uses multiple free-lists: one for each of the *regular* size blocks and one for all other *irregular* size blocks. For the purpose of this assignment, if we consider heap-space in 16-byte units and assume that the *regular* size blocks will range from 2 to 10 units each, this allows for *regular* space allocations ranging from 16 to 144 bytes (plus the overheads of the header).

If a request to malloc347() falls within the range of 2 to 10 units, we can index to the free-list of the correct size – if it is non-empty, just use the first block on that list. If the regular size free-list is empty or the request is for other than a regular size, just use first-fit from the general free-list of *irregular* size blocks.

Calls to free347() can simply add *regular* size blocks to the appropriate list. *Irregular* size blocks are added to the other list, but looking for opportunities for coalescing. You should **not** coalesce blocks on any of the *regular* size lists.

Output from the Heap Management Functions

Normally, the heap management functions would produce no output. However, you are to provide output to demonstrate that your heap management functions are operating correctly. You must display the contents of the free space list after every call to malloc347() or free347(), as already implemented in the supplied First-Fit heap management routines.

Heap Test Engine Configuration File

The size of allocation requests made by the heap test engine in its calls to malloc347() are generated at random, but determined by parameters in the configuration file. The configuration is influenced by an observation by Weinstock and Wulf that in the vast majority of application programs:

- There only a few data types for which space is requested on the heap.
- The most frequently allocated types involve relatively small, fixed amounts of storage.

The heap test engine regards those majority small allocation requests as being its "regular" requests. The parameters provided to it through the configuration file specify:

- The minimum and maximum size of those *regular* requests.
- The maximum size of the other, larger *(irregular?)* requests.
- The percentage of requests that will be regular.
- The total number of samples (calls to malloc347() or free347()) that will be generated.

Here is an example of the configuration file:

```
16          regular minimum (bytes)
144         regular maximum (bytes)
1024        irregular maximum (bytes)
80          percent of requests that are regular
10000       total number of samples
```

Saving your files in a zipped tar file

You need to submit all your C and header files, as well as your makefile. First you need to bundle them up into a single *tar* file. The term "tar" is an abbreviation of "tape archive" and goes back to the days when people would save a back-up copy of their files on magnetic tape. Nowadays, with the price of large disk drives so low, nobody uses tape anymore, but the concept of tar files survives.

1. Use the command:

   `tar -cf myfiles.tar *.c *.h Makefile`

   The -cf specifies two options for the tar command: 'c' means create and 'f' means that the name of the resulting tar file comes next in the command.

   Following the name of the tar file, we list the files to be included in the tar file. In this case, we want all the files in your current directory whose names end with `".c" or ".h",` as well as `Makefile`.

2. If you now use the command `ls` you should now see the file myfiles.tar in your directory.
3. If you use the command

   `tar -tf myfiles.tar`

   it will list the files within the tar file.

4. Now compress the tar file using the gzip program:

   ```
   gzip myfiles.tar
   ```

5. By using the `ls` command again, you should see the file myfiles.tar.gz in your directory. This is the file that you need to submit through the **Assignment 1 Submission** link in the course web site.

Coding Standards

1. Use meaningful names that give the reader a clue as to the purpose of the thing being named.
2. Avoid the repeated use of numeric constants. For any numeric constants used in your program, use #define preprocessor directives to define a macro name and then use that name wherever the value is needed.
3. Use comments at the start of the program to identify the purpose of the program, the author and the date written.
4. Use comments at the start of each function to describe the purpose of the function, the purpose of each parameter to the function, and the return value from the function.
5. Use comments at the start of each section of the program to explain what that part of the program does.
6. Use consistent indentation.