CMSC 421: Principles of Operating Systems

Homework 4: Framing the Problem

This homework is due on Tuesday, October 30, at 11:59:59 PM (Eastern daylight time). You *must* use *submit* to turn in your homework like so: `submit cs421_jtang hw4 hw4.c`

Your program must be named `hw4.c`, and it will be compiled on Ubuntu 18.04 as follows:

```
  gcc --std=c99 -Wall -O2 -o hw4 hw4.c hw4_test.c -pthread
```

(Note the above is *dash*, *dash*, *"std=c99"*, and the other flags likewise are preceded by dashes.) There must not be any compilation warnings in your submission; **warnings will result in grading penalties**. In addition, your code must be properly indented and have a file header comment, as described on the coding conventions page.

In this homework, you are writing a memory allocator from scratch. You will implement a *next-fit* allocation algorithm. Furthermore, to prevent cyber attacks, you will add memory canaries to help detect buffer overflows. As proof that your code works, your program will run instructor-provided unit tests that will exercise all functions.

Part 1: Create Memory Region

Your program will simulate a machine's memory system. The machine has 10 page frames, where each frame holds 64 bytes, for 640 bytes total. Declare this "memory" as a global array. When your program starts, initialize its contents to 0.

Implement the following function with this given signature:

```
/**
 * Write to standard output information about the current memory
 * allocations.
 *
 * Display to standard output the following:
 * - Memory contents, one frame per line, 10 lines total. Display the
 *   actual bytes stored in memory. If the byte is unprintable (ASCII
 *   value less than 32 or greater than 126), then display a dot
 *   instead.
 * - Current memory allocations. For each frame, display a 'f' if the
 *   frame is free, 'R' if reserved. If the frame is the beginning of a
 *   reserved memory block, display the four hexadecimal digit canary.
 *   Otherwise, display dashes instead of the canary.
 */
void my_malloc_stats(void);
```

Part 2: Implement Memory Allocator

Next, implement a next-fit memory allocator. In a next-fit memory allocator, the first allocation begins with the first frame. The next time an allocation occurs, begin at the frame following the previous allocation (regardless if the previous allocation was freed). Continue allocating until either the end of memory is reached or there are insufficient frames remaining; at that point allocations restart at the first frame.

Use your next-fit allocator to implement these functions with these given signatures. Note the `@a` and `@c` in the comments; those are Doxygen documentation commands.

```
/**
 * Allocate and return a contiguous memory block that is within the
 * memory region.
 *
 * The size of the returned block will be at least @a size bytes,
 * rounded up to the next 64-byte increment.
 *
 * @param size Number of bytes to allocate. If @c 0, your code may do
 * whatever it wants; my_malloc() of @c 0 is "implementation defined",
 * meaning it is up to you if you want to return @c NULL, segfault,
 * whatever.
 *
 * @return Pointer to allocated memory, or @c NULL if no space could
 * be found. If out of memory, set errno to @c ENOMEM.
 */
void *my_malloc(size_t size);

/**
 * Deallocate a memory region that was returned by my_malloc().
 *
 * If @a ptr is not a pointer returned by my_malloc(), then send a
 * SIGSEGV signal to the calling process. Likewise, calling my_free()
 * on a previously freed region results in a SIGSEGV.
 *
 * @param ptr Pointer to memory region to free. If @c NULL, do
 * nothing.
 */
void my_free(void *ptr);
```

For `my_malloc()`, you are to implement a next-fit allocation strategy. Because each page frame is 64 bytes, `my_malloc()` must round up to the next 64-byte boundary when reserving space, and the returned address must be frame-aligned. Your function allocates using the space reserved in step 1 above.

For `my_free()`, your program deallocates space that was returned by `my_malloc()`. Thus, if `my_malloc()` previously allocated 5 frames and returned the address to the first frame, calling `my_free()` will deallocate all 5 frames (not just the first).

In your code, add a comment block describing how you track which frames are allocated and how large each memory block is. Specifically, describe how your code would handle these two scenarios:

1. How would a call to `my_free()` would know to deallocate 5 frames if the user had previously called `my_malloc()` for 300 bytes?
2. How does `my_free()` know it needs to send a SEGFAULT if the user tries to free a pointer that points into the middle of a memory block?

Part 3: Implement Memory Canaries

The next thing to implement is the memory canary. Canaries are special values appended to the end of memory allocation blocks, used to detect buffer overflows. Implement this feature as follows:

1. Update your program to take a single command-line argument, an **unsigned** decimal value. Call this value the *seed*. If the user does not provide an argument, default the seed to zero.
2. As part of your program startup, call `srand()`, passing the seed as the argument.
3. For every allocation (i.e., call to `my_malloc()`, silently reserve an extra two bytes to hold the canary. This may require allocating an additional frame for the canary. As part of the allocation, call `rand()`. Store the **lower 16 bits** of the random value in the canary region. Also store those 16 bits in your memory allocation table.
4. When `my_free()` is called, also check the canary. If the passed in pointer is valid, but if the canary differs from the value in the memory allocation table for that memory block, send `SIGUSR1` to the process. Deallocate the pointer even if the canary changed.

Part 4: Make Functions Thread-Safe

The final step is to make the allocator thread-safe. If a thread is executing any of the above functions, then another thread calling the function must block.

Part 5: Test Implementation

Within your Ubuntu VM, use the `wget` command to fetch the unit test code file `http://www.csee.umbc.edu/~jtang/cs421.f18/homework/hw4/hw4_test.c`. This file declares a function with the following signature:

```
/**
 * Unit test of your memory allocator implementation. This will
 * allocate and free memory regions.
 */
extern void hw4_test(void);
```

Add the above function declaration to the top of your `hw4.c`. In your `main()` function, after you have initialized your memory, call `hw4_test()`.

The grader will use a different unit test code file during grading. You may not make any assumptions about what `hw4_test()` will do, other than that its code will compile and that it will not have compilation warnings.

Note how this code implements a basic *testing framework*, that prints what it is about to test, if the test passes or fails, and the sum of passed and failed tests. Read over `hw4_test.c`; for the projects you will need to write your own unit tests. If you continue your career in software engineering, be familiar with other unit testing frameworks.

Be sure to test that your code handles threads by spawning multiple threads, with each thread allocating and deallocating space. Also write routines that intentionally corrupt memory canaries.

Sample Output

Here is a sample output from running the program using the above `hw4_test.c`, with a seed value of **421**.

```
Test 1: Display initialized memory
Memory contents:
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
Memory allocation table:
  f:---- f:---- f:---- f:---- f:---- f:---- f:---- f:---- f:---- f:----
Test 2: Simple allocations
98: PASS
101: PASS
Memory contents:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.O................................
  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  BBBBBB..........................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
Memory allocation table:
  R:8e4f R:f70b R:---- f:---- f:---- f:---- f:---- f:---- f:---- f:----
Test 3: Simple freeing
108: PASS
Memory contents:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.O................................
  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  BBBBBB..........................................................
  CCCCCCCC.......*................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
Memory allocation table:
  f:---- R:f70b R:---- R:2a1d f:---- f:---- f:---- f:---- f:---- f:----
Test 4: Out of memory condition
115: PASS
116: PASS
Test 5: Double-free
Caught signal 11: Segmentation fault!
122: PASS
Memory contents:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.O................................
  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  BBBBBB..........................................................
  CCCCCCCC.......*................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
Memory allocation table:
  f:---- f:---- f:---- R:2a1d f:---- f:---- f:---- f:---- f:---- f:----
Test 6: Threaded allocation
67: PASS
69: PASS
67: PASS
69: PASS
Memory contents:
  AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA.O................................
  BBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBBB
  BBBBBB..........................................................
  CCCCCCCC.......*................................................
  D...............................................................
  E...............:...............................................
  ................................................................
  ................................................................
  ................................................................
  ................................................................
Memory allocation table:
  f:---- f:---- f:---- R:2a1d R:c80a R:3ab5 f:---- f:---- f:---- f:----
11 tests passed, 0 tests failed.
```

Other Hints and Notes

- Ask plenty of questions on the Blackboard discussion board.
- At the top of your submission, list any help you received as well as web pages you consulted. Please do not use any URL shorteners, such as goo.gl or TinyURL. Also, do not cite shared data services, such as Pastebin, Dropbox, or Google Drive.
- Only submit `hw4.c`. The grader will ignore any submitted `hw4_test.c`.

- You may not call `malloc()`/`calloc()`/`realloc()` (or any of its ilk) or `free()`. **Using any built-in memory allocator will result in a zero for this assignment.**
- You may not change the signatures to any of the above functions.
- As that you will be performing pointer arithmetic, be aware of the differences between `int`, `size_t`, and `ptrdiff_t`.
- Having trouble rounding up to the next page alignment? See the Linux kernel's very clever `ALIGN` macro.

Extra Credit

Sorry, there is no extra credit is available for this assignment.

Adapted from a CSS design by www.mitchinson.net