CSE 374 Final Exam Sample Solution 3/15/12

Question 1. (12 points) (strings and things) Write a function `are_same` that has two parameters of type `char*` and returns:

- 0 if the two parameters point to locations holding different string values,
- 1 if the two parameters point to different locations holding the same string values
- 2 if the two parameters point to the same location.

You may assume that both parameters point to arrays of characters that form properly terminated C strings with `'\0'` at the end of each.

You should assume that any necessary standard library headers are already `#include`d and you do not need to write any `#include`s. You should use standard library functions where appropriate.

Hint: `strcmp(s,t)` and `strncmp(s,t,n)` return 0 if string `s` and `t` are the same.

```
int are_same(char * s, char * t) {
   if (s == t) {
      return 2;
   } else if (strcmp(s,t) == 0) {
      return 1;
   } else {
      return 0;
}
```

A large number of answers had extraneous `(size_t)` and `(int)` casts in the pointer comparisons. That is not needed when comparing pointers for equality and, in the case of `(int)`, could cause errors on 64-bit machines where pointer values occupy 64 bits while `int`s only use 32. However, we did not deduct any points for extra casts in either case.

Question 2. (12 points) (preprocessor) What output does the following program produce? (It does compile and execute successfully.)

Hint: Watch out. `x*y` and `y*x` might do quite different things.

```
#include <stdio.h>

#define FOO(x,y) x + y
#define BAR(x,y) y * x

int main() {
  int a = 2;
  int b = 3;
  int c = 5;
  printf("%d\n", FOO(a+b,c));
  printf("%d\n", BAR(a+b,c));
  printf("%d\n", BAR(FOO(a,c),BAR(b,b)));
  return 0;
}
```

10

13

23

Question 3. (12 points) (Memory management) Consider the following definition for linked lists of strings in C and three functions that allegedly deallocate the space occupied by a list.

```
struct node {
  char * s;
  struct snode * next;
};
void free_list_1(struct node * lst) {
  if (lst == NULL)
    return;
  free(lst);
}
void free_list_2(struct node * lst) {
  if (lst == NULL)
    return;
  free(lst->s);
  free_list_2(lst->next);
  free(lst);
}
void free_list_3(struct node * lst) {
  if (lst == NULL)
    return;
  free(lst);
  free(lst->s);
  free_list_3(lst->next);
}
```

(a) (8 points) Explain which of the three functions is best. Explain why the other two are not well-written.

`free_list_2` is the best one. `free_list_1` causes a memory leak because it only frees the first `node` struct in the list, without freeing any of the strings or remaining nodes. `free_list_3` is incorrect because it uses the dangling pointer `lst` and the pointers in the associated free node struct after that struct has been released. It may well "work" because the memory might not be reused or changed until after `free_list_3` finishes, but it is not correct.

(b) (4 points) Explain what assumption(s) are implicitly made in the best function and how the function is wrong if the assumption(s) are violated.

The main assumption is that there are no cycles in the free list and that each node points to a distinct string (`char` array). If those assumptions are violated, `free_list_3` will perform duplicate free operations on some nodes or strings.

Question 4. (8 points) (debugging) You have been assigned to debug a program that is crashing for some reason. You have narrowed the problem down to a set of C functions that implement a list of strings. The four functions involved are:

```
init();          // must call this first before any others
add(char * s)    // add s to the list
delete(char * s) // delete s from the list if present
size()           // return number of strings in the list
```

The list package requires that function `init()` be called before any of the other three can be used successfully. Your guess is that somehow one of the other functions is being called first, but your boss wants you to prove this before any more time or effort is spent on the problem.

Explain how you could use `gdb` to discover whether one of the other functions are called before `init()`. You may not make any modification(s) to any of the source files, but you may recompile code as needed.

1. **Recompile all code with `gcc -g` to ensure debugging information is available.**
2. **Start `gdb` to debug the program.**
3. **Set breakpoints at the beginning of each of the four functions `init`, `add`, `delete`, and `size`.**
4. **Run the program. If a breakpoint is reached, and if it is at the beginning of any of the functions other than `init`, you have confirmed your hypothesis that one of the other three functions was called before `init`.**

Question 5. (8 points) (svn) Suppose you are using `svn` for a group project. You decide to move some of the code in file `foo.c` into a new file `bar.c`. You have updated the `makefile` appropriately.

(a) What `svn` command(s) should you use before your next commit?

`svn add bar.c`

(b) If you forget to do the commands from your answer to part (a), who will discover your forgetfulness and when?

One of your colleagues will discover the problem when they try to build the program after performing a `svn update` operation.

Question 6. (16 points) (t9) Suppose we extended our data structure for the t9 trie to keep track of how many times each word in the trie has been accessed:

```
struct t9node {     /* t9 trie node */
   char *w;         /* word associated with this node       */
                    /*   or NULL if no word stored here.    */
   int nlookups;    /* number of times this word w has been */
                    /*   accessed. Not defined if w==NULL.  */
   struct t9node * child[10];
                    /* pointers to subtrees of this t9 node. */
                    /* child[2]-child[9] are sub-trees for   */
                    /* digits 2-9. child[0] is the # subtree */
                    /* for words with the same digits as w.  */
                    /* Pointers to empty subtrees are NULL.  */
};
```

Complete the definition of function `max_word` below so it returns a pointer to the word `w` in the trie with the maximum associated `nlookups` value. If there is more than one word with the same maximum value, return a pointer to any one of them (i.e., break ties however you wish). You should assume that any necessary standard library headers are already `#include`d and you do not need to write any `#include`s. Hint: recursion.

You may define additional function(s) if needed as part of your solution. The next page is blank if you need more room.

```
/* return word with max # lookups in trie with root r */
char * max_word(struct t9node * r) {
   return max_node(r)->w;
}
```

(Rest of answer on the next page.)

Question 6. (cont.) Additional space if needed.

```
/* Return pointer to t9node with w!=NULL and largest  */
/* nlookups value in the subtree with root r, or NULL */
/* if there are no words in the subtree with root r.  */
struct t9node * max_node(struct t9node * r) {
   struct t9node * ans;   // best known answer so far
   struct t9node * temp;
   int k;

   // if this subtree is empty return NULL
   if (r == NULL) {
      return NULL;
   }

   // Initialize ans to result of searching '#' link
   ans = max_node(r->child[0]);

   // Search subtrees for digits 2-9 for better answer
   for (k = 2; k <= 9; k++) {
      temp = max_node(r->child[k]);
      if (temp != NULL &&
            (ans==NULL || ans->nlookups < temp->nlookups)){
         ans = temp;
      }
   }

   // if this node contains a more frequent word than any
   // subtree, this node is the answer
   if (r->w != NULL &&
         (ans == NULL || r->nlookups > ans->nlookups)) {
      ans = r;
   }

   return ans;
}
```

The conditions in the two `if` statements are somewhat subtle since the ordering relies on short-circuit evaluation both to update the result when the first non-`NULL` value is found and to ensure both pointers are not `NULL` when comparing `nlookups` values.

Question 7. (14 points) (make) Suppose we have the following collection of C header and implementation files.

```
--------------------
foo.h
--------------------
#ifndef FOO_H
#define FOO_H

#include "common.h"
...
#endif

--------------------
bar.h
--------------------
#ifndef BAR_H
#define BAR_H
...
#endif

--------------------
common.h
--------------------
#ifndef COMMON_H
#define COMMON_H
...
#endif
```

```
--------------------
foo.c
--------------------
#include "foo.h"
...

--------------------
bar.c
--------------------
#include "bar.h"
#include "foo.h"
...

--------------------
thing.c
--------------------
#include "common.h"
#include "bar.h"

int main() { ... }
```

These source files are to be used to build an executable program file named `thing`, whose `main` function is in the source file `thing.c`. The program calls functions located in all three of the `.c` files above.

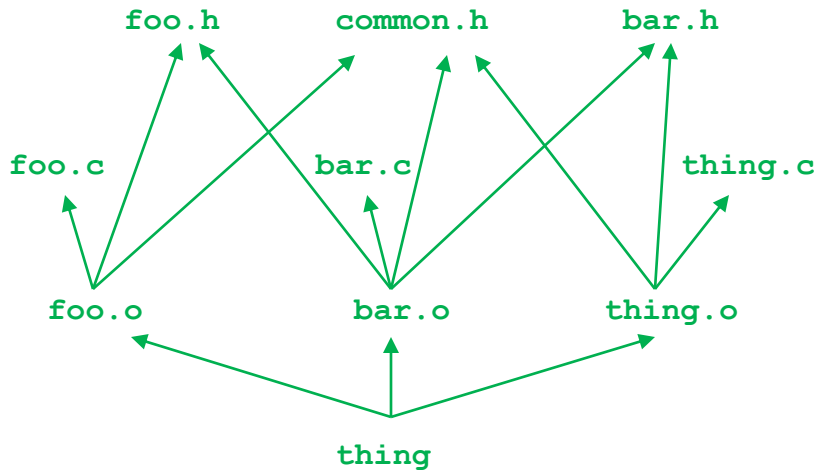
Answer the questions on the next page using the above information. You may remove this page from the exam if it makes it easier to use.

(Suggestion: sketch your answer to part (a) below or on the back of the page before you make a clean copy of it on the next page.)

x.c
↑
x.o
↑
x

Question 7. (cont.) (a) Recall that we can specify the dependencies between files in a program using a graph, where there is an arrow drawn from each file name to the file(s) it depends on. For example, the drawing to the left shows how we would diagram a program named `x` that depends on (is built from) `x.o`, which in turn depends on `x.c`.

In the space below, draw a graph (diagram) showing the dependencies between the executable program `thing` and all of the source (`.c`), header (`.h`), and compiled (`.o`) files involved in building it from the files on the previous page.



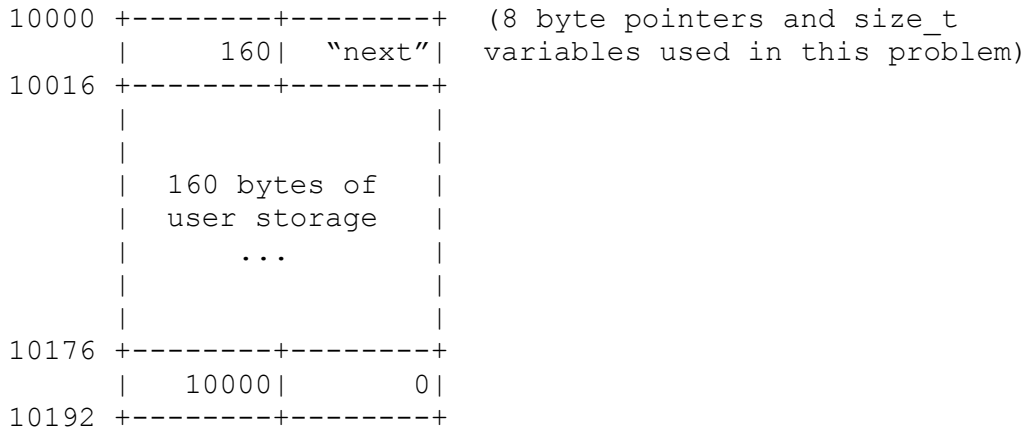
(b) Write the contents of a `Makefile` whose default target builds the program `thing`, and which only recompiles individual files as needed. Your `Makefile` should reflect the dependency graph you drew in part (a).

```
thing: foo.o bar.o thing.o
     gcc -Wall -g -o thing foo.o bar.o thing.o
foo.o: foo.c foo.h common.h
     gcc -Wall -g -c foo.c
bar.o: bar.c foo.h bar.h common.h
     gcc -Wall -g -c bar.c
thing.o: thing.c common.h bar.h
     gcc -Wall -g -c thing.c
```

Question 8. (18 points) (memory management) In the memory manager assignment, the `freemem` function had to search the free list for the proper location to add a returned block to the list and possibly merge it with other free blocks. A different way to handle this is known as the *boundry-tag* method. Here, in addition to the header at the front of every block, there is additional information in a trailer *following* each block containing a pointer back to the beginning of the block and a true/false value indicating whether the block is currently allocated. *Every* block in the heap has a header and trailer struct, whether it is currently allocated to a user program or not. Here are definitions for the header and trailer structs that surround each block:

```
struct header {          // block header:
   size_t size;          // number of data bytes in the block
                         //   without the header/trailer
   struct header* next; // next block on the free list or NULL
};
struct trailer {         // block trailer:
   struct header* hdr;   // address of header for this block
   size_t allocated;     // 1 if block allocated, 0 if free
};
```

Here is an illustration of a free block with 160 bytes of user storage whose header is at location 10000. The block plus header and trailer occupy 160+16+16 = 192 bytes.



The idea behind the boundry-tag method is that we can decide whether to merge adjacent blocks by looking at the header and trailer of adjacent blocks without having to search the free list. Given a free block, we can merge it with the block that precedes it in storage if that block has a 0 in the `allocated` field of its trailer. The trailer of the previous block is located in the 16 bytes immediately preceding the header of the current block.

Complete function `merge_with_previous` on the following page so it examines the storage immediately preceding the block whose header address is given as the parameter and, if it is also a free block, merges the two of them into a single larger block on the free list. Feel free to remove this page from the exam while you are working if you wish.

Question 8 (cont.) Complete the function below.

```
/* Given a pointer to the header of free block b, if the   */
/* block immediately preceding b in storage is also a free */
/* block, merge the two blocks into a single free block,   */
/* and update the fields in the header and trailer of the  */
/* resulting combined block as needed.                     */

void merge_with_previous(struct header * b) {
   // trailer of previous block
   struct trailer * prev_trailer;
   // header of previous block
   struct header * prev_header;
   // trailer of current block b
   struct trailer * b_trailer;

   // locate previous trailer
   prev_trailer = (struct trailer *)(((size_t) b)-16);

   // exit if previous block is not free
   if (prev_trailer->allocated == 1)
      return;

   // locate previous header and current trailer
   prev_header = prev_trailer->hdr;
   b_trailer =
      (struct trailer *) (((size_t) b) + b->size + 16);

   // set length of previous block to total length and
   // set back pointer of combined block to prev_header
   prev_header->size = prev_header->size + b->size + 32;
   b_trailer->hdr = prev_header;

   // set combined block to "free" (not required,
   // assuming both blocks were free before merging)
   b_trailer->allocated = 0;
}
```

During the exam we announced that it was okay to assume that block `b` was immediately preceded by another block, so it was unnecessary to handle the special case if it were the first block on the free list.

This answer assumes that `b` is not yet on the free list so its `next` pointer can be ignored and the free list links do not need to be updated. But solutions that changed the `next` pointers assuming that `b` was already on the free list were also okay.