COSC 301: Operating Systems - Homework 2

Due Monday, September 23, 2013

In the `list.c` file you'll find in this repository is the beginnings of a linked list library. For this linked list, each node will contain a string (of maximum length 127 characters plus 1 for null-termination), and a pointer to the next node in the list. You can think of the linked list as containing a list of names.

A definition for `struct node` can simply be:

```
struct node {
    char name[128];
    struct node *next;
};
```

For this homework, you will write a few functions to add new functionality to a linked list library. You can use any built-in C functions in your work (e.g., strlen, strncasecmp, strlcpy, etc.)

You should write all your functions in one `.c` file, and just submit that `.c` file. To test your code, you can use the sample linked list code that we wrote (or will write) in class (and modify it a little bit reflect the fact that linked lists for this homework will have a char array as the value, rather than an int).

---

1. The first function you should write is one that takes a string and a pointer to the head of the list as parameters, and prints each item in the list to the console (stdout) that matches the first parameter. The match should be done in a case-insensitive way (look up the `strncasecmp` manual page for how to do this). The function should *not* modify the list (i.e., you should be able to call `list_print_matches` repeatedly, with the same results each time). Note that this function is very similar to one we wrote in class, except for the matching element.

The type signature of `list_print` should be:

```
void list_print_matches(const char *name, const struct node *head) {

}
```

Note that the `const struct node *` indicates to the compiler that the function will not modify what `head` points to.

2. The second function to write is one that traverses the list to find the *first* name in the list that matches the parameter `name`. The string comparison should be done in a case-insensitive way (i.e., case shouldn't matter). The function should return 0 if no nodes were deleted from the list, and 1 if a match was found an a node deleted. Don't forget to free any heap-allocated memory.

The signature of `list_delete` should be:

```
int list_delete(const char *name, struct node **head) {

}
```

3. The next function is one that should append a new node to the list with the string/name passed as the first parameter. The function shouldn't return anything.

The type signature should be:

```
void list_append(const char *name, struct node **head) {

}
```

4. Next, write a function called `list_reverse` that reverses the order of all the nodes in the list *in place*. (Hint: think about using a temporary list to accumulate a new, reversed list of nodes, then update the original head pointer.)

This function shouldn't return anything, and the type signature should be:

```
void list_reverse(struct node **head) {

}
```

5. Lastly, write a function called `list_sort` that sorts the items in the list *in place*. You do not have to be fancy! A simple O(n^2) sort is ok for this function. A similar hint as the `list_reverse` function applies: think about using a temporary list to accumulate a newly sorted list, then update the original head pointer.

The type signature for `list_sort` is just:

```
void list_sort(struct node **head) {

}
```

For 5% extra credit (and a pretty good challenge) try to implement merge sort for this problem.